SPARK 2014 LRM

Release 0.3

Altran and AdaCore

June 17, 2013

CONTENTS

Date generated: June 17, 2013

CHAPTER

ONE

INTRODUCTION

SPARK 2014 is a programming language and a set of verification tools designed to meet the needs of high-assurance software development. SPARK 2014 is based on Ada 2012, 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, formal verification.

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

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 2014 is the support for a mixture of proof and other verification methods such as testing, which facilitates the use of unit proof in place of unit testing; an approach now formalized in DO-178C and the DO-333 formal methods supplement. Certain units may be formally proven and other units validated through testing.

The new aspects defined for SPARK 2014 all have equivalent pragmas which allows a SPARK 2014 program to be compiled by and executed by any Ada implementation; for instance an Ada 95 compiler provided the use of Ada 2005 and Ada 2012 specific features are avoided. The SPARK 2014 attributes Update and Loop_Entry can be used only if the Ada implementation supports them. Additionally the attribute Old can be used in only a postcondition and not in a pragma.

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 2014 reference manual. The GNAT implementation is one such compiler.

1.1 Structure of Introduction

This introduction contains the following sections:

- Section *Lifecycle of this Document* describes how this document will evolve up to and beyond the first formal release of the SPARK 2014 language and toolset.
- 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 2014.
- Section *Formal Analysis* gives a brief overview of the formal analysis to which SPARK 2014 programs are amenable.
- Section *Dynamic Semantics of SPARK 2014 Programs* gives details on the dynamic semantics of SPARK 2014.
- Section *SPARK 2014 Strategic Requirements* defines the overall goals to be met by the SPARK 2014 language and toolset.

- Section *Explaining the Strategic Requirements* provides expanded detail on the main strategic requirements.
- Section *Notes on the Current Draft* provides some brief detail on the current status and contents of this document.

1.2 Lifecycle of this Document

This document will be developed incrementally towards a number of milestones – this version of the document represents Milestone 3 – culminating in Release 1 of the document that matches the first formal release of the toolset. Subsequent releases of the document will follow, associated with subsequent formal releases of the toolset. Hence, where inclusion of particular scope is deferred, it may be deferred to:

- A specified milestone, meaning that the feature will be included in the first formal release of the toolset.
- A release subsequent to Release 1, meaning that the feature will be implemented *after* the first formal release of the toolset.

Note that the content currently in scope for the current draft of this document will only be regarded as definitive when the Release 1 version of the document is ready, and so may be subject to change.

1.3 How to Read and Interpret this Manual

This RM (reference manual) is *not* a tutorial guide to SPARK 2014. 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 2012 RM, so knowledge of Ada 2012 is assumed. Chapters 2 through 13 mirror the structure of the Ada 2012 RM. Chapter 14 covers all the annexes of the Ada 2012 RM. Moreover, this manual should be interpreted as an extension of the Ada 2012 RM (that is, SPARK 2014 is fully defined by this document taken together with the Ada 2012 RM).

SPARK 2014 introduces a number of aspects. The language rules are written as if all the SPARK 2014 specific aspects are present but minimum requirements are placed on a tool which analyzes SPARK 2014 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 2014 Aspects*). An equivalent pragma is available for each of the new aspects but these are not covered explicitly in the language rules either.

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

1.4 Method of Description

In expressing the aspects, pragmas, attributes and rules of SPARK 2014, the following chapters of this document follow the notational conventions of the Ada 2012 RM (section 1.1.4).

The following sections are given for each new language feature introduced for SPARK 2014, following the Ada 2012 RM (other than *Verification Rules*, which is specific to SPARK 2014):

1. Syntax: this section gives the format of any SPARK 2014 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 2014 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 2014 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.

Todo

We need to increase the number of examples given. To be completed in the Milestone 4 version of this document.

1.5 Formal Analysis

SPARK 2014 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 2012 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.

Todo

Consider adding a glossary, defining terms such as flow analysis and formal verification. To be completed in the Milestone 4 version of this document.

1.5.1 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 2012 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 2012 RM 4.1.1(7)).

[1] In the case of some bounded errors a check and any resulting exception only *may* be required.

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 2012 RM (11.4.2)); the one exception to this rule is pragma `Assume` (see *Proof Pragmas*).

In addition, the generation of proof obligations 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 proof obligations.

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

Note that formal verification of a program must take acount of the machine on which that program is executed and the properties of the tools used to compile and build it. In such cases it must be possible to represent the dependencies as explicit inputs to the formal verification process.

1.6 Dynamic Semantics of SPARK 2014 Programs

Every valid SPARK 2014 program is also a valid Ada 2012 program. The SPARK 2014 dynamic semantics are the same as Ada 2012 with the exception of some new aspects, pragmas and attributes which have dynamic semantics. Additionally, the new dynamic semantics only affect assertion expressions so if assertion expressions are ignored then the dynamic semantics of an Ada 2012 program are the same as a SPARK 2014 program.

SPARK 2014 programs that have failed their static analysis checks can still be valid Ada 2012 programs. An incorrect SPARK 2014 program with, say, flow analysis anomalies or undischarged proof obligations 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 2014 may make use of certain aspects, attributes and pragmas which are not defined in the Ada 2012 reference manual. Ada 2012 explicitly permits implementations to provide implementation-defined aspects, attributes and pragmas. If a SPARK 2014 program uses one of these aspects (e.g., Global), or attributes (e.g., Update) 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 2014 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 2014 program may be compiled and executed by any Ada implementation (whether or not it recognizes the SPARK 2014 pragmas). Ada specifies that unrecognized pragmas are ignored: an Ada compiler that ignores the pragma is correctly implementing the dynamic semantics of SPARK 2014 and the SPARK 2014 tools will still be able to undertake all their static checks and proofs.

Todo

The pragmas equivalent to the new aspects need to be added to this document. To be added in the Milestone 4 version of this document.

1.7 Main Program

In SPARK 2005, a dedicated annotation was used to identify the main program. There is no corresponding aspect in SPARK 2014 and instead it is expected that any implementation of SPARK 2014 will have its own mechanism to allow the tools to identify the main program (albeit not within the language itself).

1.8 SPARK 2014 Strategic Requirements

The following requirements give the principal goals to be met by SPARK 2014. Some are expanded in subsequent sections within this chapter.

- The SPARK 2014 language subset shall embody the largest subset of Ada 2012 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 2012 preconditions, postconditions and other assertions dictates that SPARK 2014 shall have executable semantics for assertion expressions. Such expressions may be executed, proven or both. To avoid having to consider potential numeric overflows when defining an assertion expression SPARK 2014 mandates a mode whereby extended or infinite integer arithmetic is supported for assertion expressions. The way in which this mode is selected is tool dependent and shall be described in the user guide for the tool. If this mode is not active, proof obligations to demonstrate the absence of overflow in assertion expressions will be present.
- SPARK 2014 shall provide for mixing of verification evidence generated by formal analysis [for code written in the SPARK 2014 subset] and evidence generated by testing or other traditional means [for code written outside of the core SPARK 2014 language, including legacy Ada code, or code written in the SPARK 2014 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 written in SPARK 2014 making it potentially entirely provable largely using automatic proof tools.
- SPARK 2014 shall support *constructive*, modular development which allows contracts to be specified on the declaration of program units and allows analysis and verification to be perfomed based on these contracts as early as possible in the development lifecycle, even before 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 2014 specific aspects.
- A SPARK 2014 analysis tool is required to synthesize at least some of the SPARK 2014 specific aspects, used to specify the contract of a program unit, if a contract is not explicitly specified, for instance the *Global Aspect* and the *Depends Aspect* from the implementation of the unit if it exists. The minimum requirements are given in *Synthesis of SPARK 2014 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 2014 specific aspects facilitates different development strategies and the analysis of pre-existing code (see section *Synthesis of SPARK 2014 Aspects*).
- Although a goal of SPARK 2014 is to provide a language that supports as many Ada 2012 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 by *Code Policies* that shall be allowed that reduce the subset of SPARK 2014 that may be used in line with specific goals such as domain needs or certification requirements. The use of a code policy may also have the effect of simplifying proof or analysis. See section *Code Policies* for further details.
- SPARK 2014 shall allow the mixing of code written in the SPARK 2014 subset with code written in full Ada 2012. See section *In and Out of SPARK 2014* for further details.
- Many systems are not written in a single programming language. SPARK 2014 shall support the development, analysis and verification of programs which are only partly in SPARK 2014, with other parts in another language, for instance, C. SPARK 2014 specific aspects manually specified at unit level will form the boundary interface between the SPARK 2014 and other parts of the program. *No further detail is given in the current draft of this document on mixing SPARK 2014 code with non-Ada code.*

Todo

Complete detail on mixing SPARK 2014 with non-Ada code. To be completed in the Milestone 4 version of this document.

- SPARK 2014 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 *Ghost Functions*.
- SPARK 2014 shall provide counterparts of all language features and analysis modes provided in SPARK 83/95/2005, unless it has been identified that customers do not find them useful.
- Enhanced support for specifying and verifying properties of secure systems shall be provided (over what is available in SPARK 2005). [The features to provide this enhanced support are not yet fully defined and will not be implemented until after release 1 of the SPARK 2014 tools.]
- SPARK 2014 shall support the analysis of external communication channels, which might be volatile variables, typically either an input or an output. See section *External 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 implementation-defined and partially-specified features may be outside of SPARK 2014 by definition, though their use could be allowed and a warning or error generated for the user. See section *In and Out of SPARK 2014* for further details.] *Note that the current draft of this document does not necessarily define all restrictions necessary to guarantee an unambiguous semantics.*
- SPARK 2014 shall support provision of "formal analysis" as defined by DO-333, 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. *Note that the current draft of this document does not necessarily define all restrictions necessary to guarantee soundness.*

Todo

Ensure that all strategic requirements have been implemented. To be completed in the Milestone 4 version of this document.

Todo

Where Ada 2012 language features are designated as not in SPARK 2014 in subsequent chapters of this document, add tracing back to the strategic requirement that motivates that designation. To be completed in the Milestone 4 version of this document.

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 2014 enforces a number of global restrictions to Ada 2012. While these are covered in more detail in the remaining chapters of this document, the most notable restrictions are:

- The use of access types and allocators is not permitted.
- 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 relation ship 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.
- The goto statement is not permitted.
- The use of controlled types is not currently permitted.
- Tasking is not currently permitted (it is intended that this will be included in Release 2 of the SPARK 2014 language and tools).
- 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 2014.
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 2014, then additional information may be provided in the code – possibly at the boundary – to indicate this (see section *In and Out of SPARK 2014* 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 toolset that proposes a combination of formal verification and testing for SPARK 2014 should provide a detailed process for doing so, including any necessary additional testing of proof assumptions.

Restrictions that Apply to the Tested Code

There are two two sources of restriction that apply to the tested code:

1. The need to validate a partial proof that relies on code that is not itself proven but is only tested.
2. The need to validate the assumptions on which a proof is based when proven code is combined with tested code.

The specific details of the restrictions to be applied to tested code – which will typically be non-SPARK 2014 – code will be given in a subsequent draft of this document.

No further detail is given in the current draft of this document on Combining Formal Verification and Testing, or on providing what it needs. Further detail will be provided at least in part under TN LC10-020.

Todo

Add detail on restrictions to be applied to tested code, making clear that the burden is on the user to get this right, and not getting it right can invalidate the assumptions on which proof is based. To be completed in the Milestone 4 version of this document.

Todo

Complete detail on combining formal verification and testing. To be completed in the Milestone 4 version of this document.

1.9.3 Code Policies

The restrictions imposed on the subset of Ada that could be used in writing SPARK 2005 programs were not simply derived from what was or is amenable to formal verification. In particular, those restrictions stemmed partly from good programming practice guidelines and the need to impose certain restrictions when working in certain domains or against certain safety standards. Hence, we want to allow such restrictions to be applied by users in a systematic and tool-checked way despite the goal that SPARK 2014 embodies the largest subset of Ada 2012 that is practical to formally verify.

Since SPARK 2014 will allow use of as large a subset of Ada 2012 as possible, this allows for the definition of multiple *Code Policies* that allow different language subsets to be used as opposed to the single subset given by SPARK 2005. Each of these code policies can be targeted to meeting a specific user need, and where a user has multiple needs then multiple policies may be enforced. Needs could be driven by:

- Application domains - for example, server-class information systems,

- Standards - for example, DO-178C Level A,
- Technical requirements - for example, systems requiring software that is compatible with a "zero footprint" run-time library.

As an example, a user developing an air traffic control system against DO-178C might impose two code policies, one for the domain of interest and one for the standard of interest.

These capabilities will be handled outside of the language since the need is not specific to SPARK, and has already been resolved either by Ada 2012 (pragma Restrictions and pragma Profile), or GNAT (pragma Restriction_Warnings) or by coding standard checkers (e.g. gnatcheck).

Todo

Complete detail on Code Policies. To be completed in the Milestone 4 version of this document. Consider referencing the User's Guide for details of the various profiles.

1.9.4 Ghost Functions

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 should be restricted to places where they do not affect the functionality of the program. Complete removal of *ghost* entities has no functional impact on the program.

SPARK 2014 currently supports ghost functions but not ghost types or variables. Ghost functions may be executable or non-executable. Non-executable ghost functions 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 2014 language and toolset and therefore cannot be checked by the tools. An unsound definition 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 2014 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.

1.9.5 Synthesis of SPARK 2014 Aspects

SPARK 2014 supports a *constructive* analysis style where all program units require contracts specified by SPARK 2014 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 2014 analysis tool shall be able to synthesize a conservative approximation of at least a minimum of SPARK 2014 specific aspects from the source code of a unit.

Synthesis of SPARK 2014 aspects is fundamental to the analysis of pre-existing code where no SPARK 2014 specific aspects are provided.

The mandatory requirements of a SPARK 2014 analysis tool is that it shall be capable of synthesizing at least a basic, conservative *Global Aspect*, *Depends Aspect* , *Refined_Global Aspect*, *Refined_Depends Aspect*, *Abstract_State Aspect*, *Refined_State Aspect* and *Initializes Aspect* from either the implementation code or from other SPARK 2014 aspects as follows:

- if 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 all of the `global_items` that have a `mode_selector` of Output or In_Out are `outputs`, those that have a `mode_selector` of Input or In_Out are `inputs` of the `dependency_relation` and that each `output` is dependent on every `input`. 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 Initializes aspect is specified for a package but it declares persistent variables which are initialized then an Initializes aspect shall be synthesized stating the visible variables that are initialized and the state abstractions representing the hidden variables that are initialized.

The syntheses described above do not include all of the SPARK 2014 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 2014 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 2014 specific aspects. If the aspects are present, the synthesized aspects may be compared with the explicit ones and auto correction used to update the aspects if the changes are acceptable. If there are aspects missing they are automatically synthesized for analysis purposes. This is also regarded as generative analysis.
- Legacy code is analyzed which has no or incomplete SPARK 2014 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 2014 and non-SPARK 2014 code (and so there is an interaction with the detail presented in section *In and Out of SPARK 2014*). This leads to two additional process steps that might be necessary:
  - An automatic identification of what code is in SPARK 2014 and what is not.
  - Manual definition of the boundary between the SPARK 2014 and non-SPARK 2014 code by explicitly specifying accurate and truthful contracts using SPARK 2014 specific aspects on the declarations of non-SPARK 2014 program units.

1.9.6 In and Out of SPARK 2014

There are various reasons why it may be necessary to combine SPARK 2014 and non-SPARK 2014 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 2014.
- Need to analyze legacy code that was not developed as SPARK 2014.

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 2014.

The default is to assume all of the program text is in SPARK 2014, although this could 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 not in SPARK and should not be analyzed. If just the body is not in SPARK 2014 a SPARK 2014 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 2014 parts of the program and – if this is not the case – the assumptions on which the analysis of the SPARK 2014 code relies may be invalidated.

In general a definition may be in SPARK 2014 but its completion need not be.

A finer grain of mixing SPARK 2014 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 2014 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 2014. If the declaration generates a warning, or if the error is justified, then the unit is considered to be in SPARK 2014 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 unsupressible error. The body of a unit causing the unsuppressible (or declaration if this is the cause) will need to be marked as not in SPARK 2014 to prevent its future analysis.

Hence, SPARK 2014 and non-SPARK 2014 code may mix at a fine level of granularity. The following combinations may be typical:

- Package specification in SPARK 2014. Package body entirely not in SPARK 2014.
- Visible part of package specification in SPARK 2014. Private part and body not in SPARK 2014.
- Package specification in SPARK 2014. Package body almost entirely in SPARK 2014, with a small number of subprogram bodies not in SPARK 2014.
- Package specification in SPARK 2014, with all bodies imported from another language.
- Package specification contains a mixture of declarations which are in SPARK 2014 and not in SPARK 2014. A client of the package may be in SPARK 2014 if it only references SPARK 2014 declarations; the presence of non-SPARK 2014 constructs in a referenced package specification does not by itself mean that a client is not in SPARK 2014.

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 (reference TBD).

Todo

We need to consider what might need to be levied on the non-SPARK 2014 code in order for flow analysis on the SPARK 2014 code to be carried out. To be completed in the Milestone 4 version of this document.

Todo

Complete detail on mixing code that is in and out of SPARK 2014. In particular, where subheadings such as Legality Rules or Static Semantics are used to classify the language rules given for new language features, any rules given to restrict the Ada subset being used need to be classified in some way (for example, as Subset Rules) and so given under a corresponding heading. In addition, the inconsistency between the headings used for statements and exceptions needs to be addressed. To be completed in the Milestone 4 version of this document.

1.9.7 External 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 2014 follows the Ada convention that a read of a volatile variable has a possible side effect of updating the variable. SPARK 2014 extends this notion to cover updates of a volatile variable such that an update of a volatile variable also has a side effect of reading the variable. SPARK 2014 further extends these principles to apply to state abstractions also using the Input_Only and Output_Only options in the declaration of a state abstraction (see section *External State*).

1.10 Notes on the Current Draft

The aim of this draft of the document is to fully define the main features of the SPARK 2014 language. Subsequent updates for release 1 of the tools are only expected to fix problems arising during implementation of the tools and correct any errors in the document.

There are two areas of the language where there is on-going significant discussion and so are likely to change. These areas are "Externals" and "Refined_Pre and Refined_Post".

CHAPTER

TWO

LEXICAL ELEMENTS

SPARK 2014 supports the full Ada 2012 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 2014 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

SPARK 2014 does not add any declarations or types to Ada 2012, but it restricts their usage.

3.1 Declarations

The view of an entity is in SPARK 2014 if and only if the corresponding declaration is in SPARK 2014. When clear from the context, we say *entity* instead of using the more formal term *view of an entity*.

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

- a scalar type with a specified Default_Value; 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 or type extension 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.

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

3.2 Types and Subtypes

The view of an entity introduced by a `private_type_declaration` is in SPARK 2014 if the types of any visible discriminants are in SPARK 2014, even if the entity declared by the corresponding `full_type_declaration` is not in SPARK 2014.

For a type or subtype to be in SPARK 2014, all predicate specifications that apply to the (sub)type must be in SPARK 2014. Notwithstanding any rule to the contrary, a (sub)type is never in SPARK 2014 if its applicable predicate is not in SPARK 2014.

Subtypes that are not preelaborable are not subject to flow analysis. [Users may write programs using such subtypes and those programs can be subject to formal verification. However, flow analysis will ignore the use of such a subtype and will instead raise a warning to indicate that its use has not been analysed.]

Todo

Lift restriction that non-preelaborable subtypes are not subject to flow analysis. To be completed in a post-Release 1 version of this document.

3.2.1 Classification of Operations

No restrictions or extensions.

3.2.2 Subtype Predicates

The Static_Predicate aspect is in SPARK 2014. The Dynamic_Predicate aspect is not in SPARK 2014.

[Eventually SPARK 2014 may include uses of the Dynamic_Predicate aspect, subject to the restriction that the predicate expression cannot take any variables as inputs. This is needed to ensure that if a value belonged to a subtype in the past, then the value will still belong to the subtype in the future. Predicates for composite types might also be restricted to disallow dependencies on non-discriminant components (but allow dependencies on discriminants and array bounds) in order to avoid cases where modifying a subcomponent can violate the subtype predicate of an enclosing object.]

Todo

Add the Dynamic_Predicate aspect to SPARK 2014. To be completed in a post-Release 1 version of this document.

3.3 Objects and Named Numbers

The entity declared by an `object_declaration` is in SPARK 2014 if its declaration does not contain the reserved word **aliased**, its type is in SPARK 2014, and its *initialization*`_expression`, if any, is in SPARK 2014.

3.3.1 Object Declarations

No extensions or restrictions.

3.3.2 Number Declarations

Constants that are not preelaborable are not subject to flow analysis. [Users may write programs using such constants and those programs can be subject to formal verification. However, flow analysis will ignore the use of such a constant and will instead raise a warning to indicate that its use has not been analysed.]

Todo

Lift restriction that non-preelaborable constants are not subject to flow analysis. To be completed in a post-Release 1 version of this document.

3.4 Derived Types and Classes

An entity declared by a `derived_type` declaration is in SPARK 2014 if its parent type is in SPARK 2014, and if the declaration contains an `interface_list` or a `record_part` these must also contain entities that are in SPARK 2014.

3.5 Scalar Types

No extensions or restrictions.

3.6 Array Types

An entity declared by a `array_type_definition` is in SPARK 2014 if its components are in SPARK 2014 and default initialization is in SPARK 2014.

3.7 Discriminants

A `discriminant_specification` is in SPARK 2014 if its type is discrete and it does not occur as part of a derived type declaration whose parent type is discriminated. [In other words, inherited discriminants shall not be hidden.]

3.8 Record Types

SPARK 2014 does not permit partial default initialization of record objects. More specifically, if at least one non-discriminant component (either explicitly declared or inherited) of a record type or type extension either is of a type which defines default initialization or is declared by a component_declaration which includes a Default_Value, then the record type or type extension shall define full default initialization.

[The enforcement of this rule requires looking at the `full_type_declaration` of a `private_type` declaration to determine whether the private type has a default initialization. A future version of SPARK 2014 may introduce extra features to avoid having to do this.]

3.9 Tagged Types and Type Extensions

Use of the 'Class attribute is not permitted in SPARK 2014.

[This restriction may be relaxed at some point in the future. As a consequence of this restriction, dispatching calls are not currently in SPARK 2014 but are planned for a future release.]

Todo

Add 'Class attribute to SPARK 2014. To be completed in a post-Release 1 version of this document.

3.9.1 Type Extensions

A type extension declared within a subprogram body, block statement, or generic body which does not also enclose the declaration of each of its ancestor types is not in SPARK 2014.

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

Use of `interface_type_definition` is not permitted in SPARK 2014.

Todo

Include interface types in SPARK 2014. To be completed in a post-Release 1 version of this document.

3.10 Access Types

Access types allow the creation of aliased data structures and objects, which notably complicate the specification and verification of a program's behavior. Therefore, all forms of access type declaration are excluded from SPARK 2014.

The attribute `Access` is not in SPARK 2014.

Finally, as they are based on access discriminants, user-defined references and user-defined indexing are not in SPARK 2014.

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 2014.

4.1 Names

A name that denotes an entity is in SPARK 2014 if and only if the entity is in SPARK 2014. Neither `explicit_dereference` nor `implicit_dereference` are in SPARK 2014.

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 2014 if this property can be determined prior to formal program verification.

In particular, 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

The `attribute_designator` Access is not allowed in SPARK 2014.

Todo

Are there any other language defined attributes which will not be supported? To be completed in the Milestone 4 version of this document.

Todo

What do we do about Gnat defined attributes, a useful one is: For a prefix X that denotes an object, the GNAT-defined attribute `X'Valid_Scalars` is defined in SPARK 2014. This Boolean-valued attribute is equal to the conjunction of the `Valid` attributes of all of the scalar parts of X.

[If X has no volatile parts, `X'Valid_Scalars` implies that each scalar subcomponent of X has a value belonging to its subtype. Unlike the Ada-defined `Valid` attribute, the `Valid_Scalars` attribute is defined for all objects, not just scalar objects.]

Perhaps we should list which ones are supported in an appendix? Or should they be part of the main language definition?

It would be possible to use such attributes in assertion expressions but not generally in Ada code in a non-Gnat compiler.

To be completed in the Milestone 4 version of this document. Note that as language-defined attributes form Appendix K of the Ada RM, any GNAT-defined attributes supported in SPARK 2014 will be presented in an appendix.

4.1.5 User-Defined References

User-defined references are not allowed in SPARK 2014 and so the aspect Implicit_Dereference is not in SPARK 2014.

4.1.6 User-Defined Indexing

User-defined indexing is not allowed in SPARK 2014 and so the aspects Constant_Indexing and Variable_Indexing are not in SPARK 2014.

4.2 Literals

The literal **null** representing an access value is not allowed in SPARK 2014.

4.3 Aggregates

Legality Rules

The box symbol, <>, may only be used in an aggregate if the type(s) of the corresponding component(s) 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.]

4.3.1 Update Expressions

Todo

Detail on Update Expressions needs to be put into the standard format. To be completed in the Milestone 4 version of this document.

The Update attribute provides a way of overwriting specified components of a copy of a given composite value. For a prefix X that denotes an object of a nonlimited record type or record extension T, the attribute

```
X'Update ( record_component_association_list )
```

is defined and yields a value of type T. The record_component_association_list shall have one or more record_component_associations, each of which shall have a non-**others** component_choice_list and an expression.

Each selector_name of each record_component_name shall denote a distinct non discriminant component of T. Each record_component_association's associated components shall all be of the same type. The expected type and applicable index constraint of the expression is defined as for a record_component_association occurring within a record aggregate.

In all cases (i.e., whether T is a record type, a record extension type, or an array type - see below), evaluation of X'Update begins with the creation of an anonymous object of type T which is initialized to the value of X in the same way as for an occurrence of X'Old (except that the object is constrained by its initial value but not constant). Next, components of this object are updated as described below. The attribute reference then denotes a constant view of this updated object. The master and accessibility level of this object are defined as for the anonymous object of an aggregate. The assignments to components of the result object described below are assignment operations and include performance of any checks associated with evaluation of the target component name or with implicit conversion of the source value to the component subtype.

If T is a record type or record extension then the component updating referenced above proceeds as follows. For each component for which an expression is provided, the expression value is assigned to the corresponding component of the result object. The order in which the components are updated is unspecified.

For a prefix X that denotes an object of a nonlimited one dimensional array type T, the attribute

```
X'Update ( array_component_association {, array_component_association} )
```

is defined and yields a value of type T.

Each array_component_association of the attribute reference shall have one or more array_component_associations, each of which shall have an expression. The expected type and applicable index constraint of the expression is defined as for an array_component_association occurring within an array aggregate of type T. The expected type for each discrete_choice is the index type of T. The reserved word **others** shall not occur as a discrete_choice of an array_component_association of the attribute_reference.

For a prefix X that denotes an object of a nonlimited multidimensional array type T, the attribute

```
X'Update ( multidimensional_array_component_association
        {, multidimensional_array_component_association} )
```

is defined with associated syntax

```
multidimensional_array_component_association ::=
  index_expression_list_list => expression
index_expression_list_list ::=
  index_expression_list { | index_expression_list }
index_expression_list ::=
  ( expression {, expression} )
```

and yields an object of type T.

The expected type and applicable index constraint of the expression of a multidimensional_array_component_association are defined as for the expression of an array_component_association occurring within an array aggregate of type T. The length of each

index_expression_list shall equal the dimensionality of T. The expected type for each expression in an index_expression_list is the corresponding index type of T.

If T is a one-dimensional array type then the component updating referenced above proceeds as follows. The discrete choices and array component expressions are evaluated. Each array component expression is evaluated once for each associated component, as for an array aggregate. For each such associated component of the result object, the expression value is assigned to the component. Evaluations and updates are performed in the order in which the array_component_associations are given; within a single array_component_association, in the order of the discrete_choice_list; and within the range of a single discrete_choice, in ascending order.

If T is a multidimensional type then the component updating referenced above proceeds as follows. For each multidimensional_array_component association (in the order in which they are given) and for each index_expression_list (in the order in which they are given), the index values of the index_expression_list and the expression are evaluated (in unspecified order) and the expression value is assigned to the component of the result object indexed by the given index values. Each array component expression is evaluated once for each associated index_expression_list.

Note: the Update attribute for an array object allows multiple assignments to the same component, as in either

```
Some_Array'Update (1 .. 10 => True, 5 => False)
```

or

```
Some_Array'Update (Param_1'Range => True, Param_2'Range => False)
-- ok even if the two ranges overlap
```

This is different from the Update attribute of a record

```
Some_Record'Update
  (Field_1 => ... ,
   Field_2 => ... ,
   Field_1 => ... ); -- illegal; components not distinct
```

for which the order of component updates is unspecified.

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.

An expression is in SPARK 2014 only if its type is in SPARK 2014 and the expression is side-effect free.

An expression (or range) in SPARK 2014 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;
- the default_expression of a discriminant_specification or component_declaration;
- a Dynamic_Predicate aspect specification;
- an indexing expresssion of an indexed_component or the discrete_range of a slice in an object renaming declaration which renames part of that index or slice.

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

[The Dynamic_Predicate rule is redundant because no use of the Dynamic_Predicate is currently in SPARK 2014. This rule is added in anticipation of the possible relaxation of that restriction.]

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 2012 RM 4.5). SPARK 2014 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.

[The GNAT Ada 2012 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

The use of allocators is not allowed in SPARK 2014.

4.9 Static Expressions and Static Subtypes

No extensions or restrictions.

CHAPTER

FIVE

STATEMENTS

SPARK 2014 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 2014 restricts statements that complicate verification, and excludes statements related to tasking and synchronization.

Extended Legality Rules

1. A `simple_statement` shall not be a `goto_statement`, an `entry_call_statement`, a `requeue_statement`, a `delay_statement`, an `abort_statement`, or a `code_statement`.
2. A `compound_statement` shall not be an `accept_statement` or a `select_statement`.

A statement is only in SPARK 2014 if all the constructs used in the statement are in SPARK 2014.

[A future release of SPARK 2014 is planned to support the Ravenscar multi-tasking profile and then some of the tasking statements such as `entry_call_statement`, and `delay_statement` will be permitted.]

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

SPARK 2014 currently does not support the implementation of user-defined iterator types.

Todo

Need to consider further the support for iterators and whether the application of constant iterators could be supported.

5.5.2 Generalized Loop Iteration

SPARK 2014 currently does not support generalized loop iteration.

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 ::= loop_variant_item {, loop_variant_item}
loop_variant_item       ::= change_direction => discrete_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

1. Loop_Invariant is 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.
2. Loop_Variant 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.
3. 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.
4. 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`.

[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.]

5. The expression of a `loop_variant_item` shall be of any discrete type.

Dynamic Semantics

1. 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].
2. The elaboration of an Checked Loop_Variant pragma begins by evaluating the `discrete_expressions` in textual order. 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].

Examples

The following example illustrates some pragmas of this section

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

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)]

1. The value of X'Loop_Entry [(loop_name] is the value of X on entry to the loop that is denoted by `loop_name`. If the optional `loop_name` parameter is not provided, the closest enclosing loop is the default.

Legality Rules

1. 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`.
2. 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 denote an entity;
   - the prefix of the `attribute_reference` shall not contain a Loop_Entry `attribute_reference`.
3. A Loop_Entry `attribute_reference` shall occur within a Loop_Variant or Loop_Invariant pragma.
4. 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
     ((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].
5. The prefix of a Loop_Entry `attribute_reference` shall statically denote 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 denote an entity and have 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.]

Dynamic Semantics

1. 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.

2. 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));
         ...; -- 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

The goto statement is not permitted in SPARK 2014.

5.9 Proof Pragmas

This section discusses the pragmas Assert_And_Cut and Assume.

Two SPARK 2014 pragmas are defined, Assert_And_Cut and Assume. Each has a single Boolean parameter and may be used wherever pragma Assert is allowed.

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 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 the same as a pragma Assert except that there is no proof obligation 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

1. 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

1. The verification rules for pragma Assume are significantly different to that 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 proof obligation. 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.]

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 2014 even if the implementation view is not.

Rules are imposed in SPARK 2014 to ensure that the execution of a function call does not modify any variables declared outside of the function. It follows as a consequence of these rules that the evaluation of any SPARK 2014 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 *Abstraction of State*). Global items are presented in Global aspects (see *Global Aspect*).

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.

Static Semantics

1. The *exit* value of a global item or parameter of a subprogram is its value immediately following the successful 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 may be updated by a call to the subprogram. The result of a function is also an output.
4. An *input* of a subprogram is a global item or parameter whose initial value may be used in determining the exit value of an output of the subprogram. 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 Aspect*).

Verification Rules

1. A function declaration 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 for which no explicit declaration is given.

6.1.1 Preconditions and Postconditions

As indicated by the `aspect_specification` being part of a `subprogram_declaration`, a subprogram is in SPARK 2014 only if its specific contract expressions (introduced by Pre and Post), if any, are in SPARK 2014.

For an `expression_function_declaration`, F, without an explicit Postcondition, the expression, E, implementing the function acts as its Postcondition, that is the default postcondition is F'Result = E.

In general the expression, E, of a postcondition of a function may be used as the expression of an `expression_function_declaration` instead making E both the implementation of the function and the expression of its [default] postcondition.

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 2014 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, these aspects are allowed in the same contexts as a Pre or Post aspect.

See section *Contract Cases* for further detail on Contract_Case aspects, section *Global Aspect* for further detail on Global aspects and section *Depends Aspect* 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 must 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

1. 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

1. 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.

Verification Rules

1. Each `condition` in a Contract_Cases aspect has to be proven to be mutually exclusive, that is only one `condition` can be True with any set of inputs conformant with the formal parameters and satisfying the specific precondition.
2. At the point of call a check that a single `condition` of the Contract_Cases aspect is True has to be proven, or if no `condition` is True then the Contract_Cases aspect must have an **others** `contract_case`.

3. For every `contract_case`, when its `condition` is True, or the **others** `contract_case` when none of the conditions are True, the implementation of the body of the subprogram must be proven to satisfy the `consequence` of the `contract_case`.

Todo

(TJJ 29/11/12) Do we need this verification rule? Could it be captured as part of the general statement about proof? To be completed in milestone 4 version of this document.

6.1.4 Global Aspect

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 may only be specified for the initial declaration of a subprogram (which may be a declaration, a body or a body stub). The implementation of a subprogram body must be consistent with the subprogram's Global aspect.

Note that a Refined_Global aspect may be applied to a subprogram body when using state abstraction; see section *Refined_Global Aspect* 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`

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.

Name Resolution Rules

1. 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

1. For a subprogram that has a `global_specification`, an entire object that is outside the subprogram, can only be referenced within if it is a `global_item` in the `global_specification`.
2. A `global_item` shall not denote a constant object other than a formal parameter [of an enclosing subprogram] of mode **in**. [This restriction may be relaxed in some way at some point in the future.]
3. 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).
4. 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)].
5. Each `mode_selector` shall occur at most once in a single Global aspect.
6. A function subprogram shall not have a `mode_selector` of Output or In_Out in its Global aspect.
7. The `global_items` in a single Global aspect specification shall denote distinct entities.
8. A `global_item` occurring in a Global aspect specification of a subprogram shall not denote a formal parameter of the subprogram.
9. 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, 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.

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

1. A `global_item` shall occur in a Global aspect of a subprogram if and only if it denotes an entity that is referenced by the subprogram.
2. 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` that denotes an output but not an input and is always fully initialized as a result of any successful execution of a call of the subprogram has a `mode_selector` of Output;
   - otherwise the `global_item` denotes both an input and an output, is 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 any part of the output are ignored. Similarly, for purposes of determining whether an entity is "fully initialized as a result of any successful execution of the call", only nondiscriminant parts are 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.]

1. 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.

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.
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 Aspect

A Depends aspect defines a *dependency relation* for a subprogram which may be given in the `aspect_specification` of the subprogram. The dependency relation is used in information flow analysis.

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 exit value of output, X, on the completion of the subprogram is at least partly determined from the entry value of input, *Y* and 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**.

The functional behavior of a subprogram is not specified by the Depends aspect but, unlike a postcondition, the Depends aspect has to be complete in the sense that every input and output of the subprogram must appear in the Depends aspect.

The Depends aspect may only be specified for the initial declaration of a subprogram (which may be a declaration, a body or a body stub). The implementation of a subprogram body must be consistent with the subprogram's Depends Aspect.

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

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

1. 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).
2. 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].
3. 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.
4. The *input set* of a subprogram is the *explicit input set* of the subprogram augmented with those formal parameters of mode **out** 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** which are of an unconstrained array subtype, an unconstrained discriminated subtype, a tagged type, or a type having a subcomponent of an unconstrained discriminated subtype. [Tagged types are mentioned in this rule in anticipation of a later version of SPARK 2014 in which the current restriction on uses of the 'Class attribute is relaxed; currently there is no way to read or otherwise depend on the underlying tag of an **out** mode formal parameter of a tagged type.]
5. 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`.
6. The entity denoted by each `input` of a `dependency_relation` of a subprogram shall be a member of the input set of the subprogram.
7. 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.
8. The entity denoted by each `output` of a `dependency_relation` of a subprogram shall be a member of the output set of the subprogram.
9. Every member of the output set of a subprogram shall be denoted by exactly one `output` in the `dependency_relation` of the subprogram.
10. 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.
11. In a `dependency_relation` there can be at most one `dependency_clause` which is a `null_dependency_clause` and if it exists it must be the last `dependency_clause` in the `dependency_relation`.
12. An entity denoted by an `input` which is in an `input_list` of a **null** `output_list` shall not be denoted by an `input` in another `input_list` of the same `dependency_relation`.
13. The `inputs` in a single `input_list` shall denote distinct entities.
14. A `null_dependency_clause` shall not have an `input_list` of **null**.

Static Semantics

1. 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)).]
2. 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)).]
3. 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 subrogram (other than itself, if the `output_list` =>+ **null** self-dependency syntax is used).
4. 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.
5. 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).]
6. [A function without an explicit Depends aspect specification is assumed to have the `dependency_relation` that its result is dependent on all of its inputs. Generally an explicit Depends aspect is not required for functions.]
7. [A subprogram which has an explicit Depends aspect specification and lacks an explicit Global aspect specification is assumed to have the [unique] Global aspect specification that is consistent with the subprogram's Depends aspect.]
8. [A subprogram which has an explicit Global aspect specification but lacks an explicit Depends aspect specification and, as yet, has no implementation of its body is assumed to have the conservative `dependency_relation` that each member of the output set is dependent on every member of the input set.]

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

1. Each entity denoted by an `output` given in the Depends aspect of a subprogram must be an output in the implementation of the subprogram body and the output must depend on all, but only, the entities denoted by the `inputs` given in the `input_list` associated with the `output`.
2. Each output of the implementation of the subprogram body is denoted by an `output` in the Depends aspect of the subprogram.
3. [Each input of the implementation of a subprogram body is denoted by an `input` of the Depends aspect of the subprogram.]

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 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 Ghost Functions

Ghost functions are intended for use in discharging proof obligations and in making it easier to express assertions about a program. The essential property of ghost functions is that they have no effect on the dynamic behavior of a valid SPARK program other than, depending on the assertion policy, the execution of known to be true assertion expressions. More specifically, if one were to take a valid SPARK program and remove all ghost function declarations from it and all assertions containing references to those functions, then the resulting program might no longer be a valid SPARK program (e.g., it might no longer be possible to discharge all the program's proof obligations) but its dynamic semantics (when viewed as an Ada program) should be unaffected by this transformation other than evaluating fewer known to be true assertion expressions.

The rules below are in given in general terms in relation to "ghost entities" since in future it is intended that ghost types and ghost variables will be allowed. Currently, however, only ghost functions are allowed and so an additional legality rule is provided that allows only functions to be explicitly declared as a ghost (though entities declared within a ghost function are regarded implicitly as ghost entities). When the full scope of ghost entities is allowed, the rules given in this section may be moved to other sections as appropriate, since they will refer to more than just subprograms.

Todo

Add ghost types and ghost variables to SPARK 2014. To be completed in a post-Release 1 version of this document.

Static Semantics

1. SPARK 2014 defines the `convention_identifier` Ghost. An entity (e.g., a subprogram or an object) whose Convention aspect is specified to have the value Ghost is said to be a ghost entity (e.g., a ghost function or a ghost variable).

2. The Convention aspect of an entity declared inside of a ghost entity (e.g., within the body of a ghost function) is defined to be Ghost.
3. The Link_Name aspect of an imported ghost entity is defined to be a name that cannot be resolved in the external environment.

Legality Rules

1. Only functions can be explicitly declared with the Convention aspect Ghost. [This means that the scope of the following rules is restricted to functions, even though they are stated in more general terms.]
2. A ghost entity shall only be referenced:
   - from within an assertion expression; or
   - within or as part of the declaration or completion of a ghost entity (e.g., from within the body of a ghost function); or
   - within a statement which does not contain (and is not itself) either an assignment statement targeting a non-ghost variable or a procedure call which passes a non-ghost variable as an out or in out mode actual parameter.
3. Within a ghost procedure, the view of any non-ghost variable is a constant view. Within a ghost procedure, a volatile object shall not be read. [In a ghost procedure we do not want to allow assignments to non-ghosts either via assignment statements or procedure calls.]
4. A ghost entity shall not be referenced from within the expression of a predicate specification of a non-ghost subtype [because such predicates participate in determining the outcome of a membership test].
5. All subcomponents of a ghost object shall be initialized by the elaboration of the declaration of the object.

Todo

Make worst-case assumptions about private types for this rule, or blast through privacy? To be completed in milestone 4 version of this document.

---

6. A ghost instantiation shall not be an instantiation of a non-ghost generic package. [This is a conservative rule until we have more precise rules about the side effects of elaborating an instance of a generic package. We will need the general rule that the elaboration of a ghost declaration of any kind cannot modify non-ghost state.]
7. The Link_Name or External_Name aspects of an imported ghost entity shall not be specified. A Convention aspect specification for an entity declared inside of a ghost entity shall be confirming [(in other words, the specified Convention shall be Ghost)].
8. Ghost tagged types are disallowed. [This is because just the existence of a ghost tagged type (even if it is never referenced) changes the behavior of Ada.Tags operations. Note overriding is not a problem because Convention participates in conformance checks (so ghost can't override non-ghost and vice versa).]
9. The Convention aspect of an External entity shall not be Ghost.

[We are ignoring interactions between ghostliness and freezing. Adding a ghost variable, for example, could change the freezing point of a non-ghost type. It appears that this is ok; that is, this does not violate the ghosts-have-no-effect-on-program-behavior rule.

Todo

Can a ghost variable be a constituent of a non-ghost state abstraction, or would this somehow allow unwanted dependencies? If not, then we presumably need to allow ghost state abstractions or else it would be illegal for a library level package body to declare a ghost variable. To be completed in a post-Release 1 version of this document.

Todo

Do we want an implicit Ghost convention for an entity declared within a statement whose execution depends on a ghost value? To be completed in a post-Release 1 version of this document.

Dynamic Semantics

1. The effects of specifying a convention of Ghost on the runtime representation, calling conventions, and other such dynamic properties of an entity are the same as if a convention of Ada had been specified.

   [If it is intended that a ghost entity should not have any runtime representation (e.g., if the entity is used only in discharging proof obligations and is not referenced (directly or indirectly) in any enabled (e.g., via an Assertion_Policy pragma) assertions), then the Import aspect of the entity may be specified to be True.]

Verification Rules

1. A non-ghost output shall not depend on a ghost input.
2. A ghost entity shall not be referenced
   - within a call to a procedure which has a non-ghost output; or
   - within a control flow expression (e.g., the condition of an if statement, the selecting expression of a case statement, the bounds of a for loop) of a compound statement which contains such a procedure call. [The case of a non-ghost-updating assignment statement is handled by a legality rule; this rule is needed to prevent a call to a procedure which updates a non-ghost via an up-level reference, as opposed to updating a parameter.]

     [This rule is intended to ensure an update of a non-ghost entity shall not have a control flow dependency on a ghost entity.]
3. A ghost procedure shall not have a non-ghost output.

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_Function'Result in Lo .. Hi,
        Convention => Ghost;

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

6.2 Formal Parameter Modes

No extensions or restrictions.

Todo

The modes of a subprogram in Ada are not as strict as S2005 and there is a difference in interpretation of the modes as viewed by flow analysis. For instance in Ada a formal parameter of mode out of a composite type need only be partially updated, but in flow analysis this would have mode in out. Similarly an Ada formal parameter may have mode in out but not be an input. In flow analysis it would be regarded as an input and give rise to flow errors.

In deciding whether a parameter is only partially updated, discriminants (including discriminants of subcomponents) are ignored. For example, given an *out* mode parameter of a type with defaulted discriminants, a subprogram might or might not modify those discriminants (if it does, there will of course be an associated proof obligation to show that the parameter's 'Constrained attribute is False in that path).

Perhaps we need an aspect to describe the strict view of a parameter if it is different from the specified Ada mode of the formal parameter? To be completed in a post-Release 1 version of this document.

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

A call is in SPARK 2014 only if it resolves statically to a subprogram whose declaration view is in SPARK 2014.

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.

A common place for aliasing to be introduced is through the actual parameters and between actual parameters and global variables in a procedure call. Extra verification rules are given that avoid the possibility of aliasing through actual parameters and global variables. A function is not allowed to have side-effects and cannot update an actual parameter or global variable. Therefore, function calls cannot introduce aliasing and are excluded from the anti-aliasing rules given below for procedure calls.

Syntax

No extra syntax is associated with anti-aliasing.

Legality Rules

No extra legality rules are associated with anti-aliasing.

Static Semantics

1. Objects are assumed to have overlapping locations if it cannot be established statically that they do not. [This definition of overlapping is necessary since these anti-aliasing checks will initially be implemented by flow analysis; in a future tool release it is intended that these checks will be implemented by the proof engine and so the static checking may be suppressed.]

Dynamic Semantics

No extra dynamic semantics are associated with anti-aliasing.

Verification Rules

1. A procedure call shall not pass actual parameters which denote objects with overlapping locations, when at least one of the corresponding formal parameters is of mode **out** or **in out**, unless the other corresponding formal parameter is of mode **in** and is of a by-copy type.
2. A procedure call shall not pass an actual parameter, whose corresponding formal parameter is mode **out** or **in out**, that denotes an object which overlaps with any `global_item` referenced by the subprogram.
3. A procedure call shall not pass an actual parameter which denotes an object which overlaps a `global_item` of mode **out** or **in out** of the subprogram, unless the corresponding formal parameter is of mode **in** and by-copy.
4. 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.

6.5 Return Statements

No extensions or restrictions.

6.5.1 Nonreturning Procedures

Syntax

There is no additional syntax associated with nonreturning procedures in SPARK 2014.

Legality Rules

1. For a call to a nonreturning procedure to be in SPARK 2014, it must be immediately enclosed by an if statement which encloses no other statement.

Static Semantics

There are no additional static semantics associated with nonreturning procedures in SPARK 2014.

Dynamic Semantics

There are no additional dynamic semantics associated with nonreturning procedures in SPARK 2014.

Verification Rules

1. A call to a nonreturning procedure introduces an obligation to prove that the statement will not be executed, much like the proof obligation associated with

   ```
   pragma Assert (False);
   ```

   [In other words, the proof obligations introduced for a call to a nonreturning procedure are the same as those introduced for a runtime check which fails unconditionally. See also section *Exceptions*, where a similar restriction is imposed on `raise_statements`.]

6.6 Overloading of Operators

No extensions or restrictions.

6.7 Null Procedures

No extensions or restrictions.

6.8 Expression Functions

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*).

CHAPTER

SEVEN

PACKAGES

Verification Rules

1. In SPARK 2014 the elaboration of a package shall only update, directly or indirectly, variables declared immediately within the package.

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 for flow analysis and proof. *State abstraction* is the means by which this hidden state is managed for flow analysis and proof. A state abstraction represents one or more declarations which are part of the hidden state of a package.

SPARK 2014 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 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 declared immediately in the private part or body of P; and
   - the state declared in the visible part of any packages declared immediately within the private part or body of P.

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 2014 program.

External state may be specified as *output only* or *input only*. The update of output only external state is considered to be read by some external reader of the state and so the update is not ineffective even though the program itself cannot read the updated state. Input only external state is considered to be updated by some external writer and so successive reads of the input only external state may not give the same value even though the program itself cannot update the state. In other words the update of output only and the read of input only external state both have a side effect.

Output only and input only external states are treated specially in Global and Depends aspects as described below and cannot be denoted as actual parameters or be `global_items` of a function as they would introduce side-effects.

External state that is not specified as output only or input only behaves as normal (non-volatile) state and may be read or updated by the program and has no special treatment for Global and Depends aspects.

SPARK 2014 aspects are defined for specifying a variable as an input only or an output only [Ada aspects Volatile, Import and Export are used for specifying whether a variable is external] see *Input_Only and Output_Only Aspects*). When a state abstraction is declared by an Abstract_State aspect (see *Abstract_State Aspect*) it may be specified as external, in which case it may be also specified as either input only or output only.

Static Semantics

Static semantics are given individually for external variables and external state abstractions.

Legality Rules

1. External state which is specified as input only shall not be denoted in a Global aspect with a `mode_selector` of In_Out or Output. [Nor shall it be denoted as an `output` of a Depends aspect.]
2. External state which is specified as output only shall not be denoted in a Global aspect with a `mode_selector` of Input or In_Out. [Nor shall not be denoted as an `input` of a Depends aspect.]
3. A `global_item` of a function shall not denote input only or output only external state.
4. An actual parameter in a function call shall not denote output only or input only external state.
5. Since output only external state shall never be read by the program and input only external state may never be updated by the program neither of these sorts of external state shall be denoted by a `name` of an `initialization_item` of an Initializes aspect (see *Initializes Aspect*).

7.1.3 Input_Only and Output_Only Aspects

A variable which represents a communication channel with an external entity, for instance a transducer, subsystem, or program is considered an *external variable*. A variable is external if it is Volatile or is declared with an Ada Address, Import, or Export specification (either using an aspect or a pragma).

If a variable is volatile it has to be specified as an input only or an output only external state. The Boolean aspects Input_Only and Output_Only are used for this specification.

Static Semantics

1. A variable which is Volatile or has one of the Ada aspects Import or Export, or the Ada aspect Address specified in its declaration is an external variable.

Legality Rules

1. The declaration of Volatile variable shall have exactly one of an Input_Only or Output_Only aspect specified as True. A variable with a True Input_Only specification is an *external input*; a variable with a True Output_Only specification is an *external output*. [The rule that a volatile variable shall be either an input or an output only may be relaxed in a future version of SPARK.]

2. A variable which is not Volatile shall not have an Input_Only or Output_Only aspect specified as True.
3. The Boolean expression of the aspect definitions of the Input_Only or Output_Only aspects shall be static.
4. Contrary to the general SPARK 2014 rule that expression evaluation cannot have side effects, a read of an external input is considered to have side effects. To reconcile this discrepancy, a name denoting an external input shall only occur in the following contexts:
   - as the [right hand side] expression of an assignment statement; or
   - as the expression of an initialization expression of an object declaration that is not specified as volatile; or
   - as an actual parameter in a call to an instance of Unchecked_Conversion whose result is renamed [in an object renaming declaration]; or
   - as an actual parameter in a procedure call of which the corresponding formal parameter is mode **in** and is of a non-scalar volatile type.

   [This rule means that an external input cannot be updated directly by the program.]
5. A name denoting an external output shall only occur in the following contexts:
   - as the name on the left-hand side of an assignment statement; or
   - as an actual parameter in a procedure call of which the mode of the corresponding formal parameter is **out** and is of a non-scalar volatile type.

   [This rule means that an external output cannot be directly read by the program.]
6. See section on volatile variables for rules concerning their use in SPARK 2014 (*Shared Variable Control (Annex C.6)*).

Dynamic Semantics

There are no dynamic semantics associated with these aspects.

Verification Rules

There are no extra verification rules.

Examples

```
with System.Storage_Units;
package Input_Port
is
   Sensor : Integer
      with Volatile,
           Input_Only,
           Address => System.Storage_Units.To_Address (16#ACECAFE#);
end Input_Port;


with System.Storage_Units;
package Multiple_Ports
is
   type Volatile_Type is record
      I : Integer
   end record with Volatile;

   -- Read_Port may only be called with an actual parameter for Port
   -- which is an external input only
   procedure Read_Port (Port : in Volatile_Type; Value : out Integer)
      with Depends => (Value => Port); -- Port is an external input only
```

```
   -- Write_Port may only be called with an actual parameter for Port
   -- which is an external output only
   procedure Write_Port (Port : out Volatile_Type; Value : in Integer)
      with Depends => (Port => Value); -- Port is external output only

   -- The following declarations are all external input only variables
   V_In_1 : Volatile_Type
      with Input_Only,
           Address => System.Storage_Units.To_Address (16#A1CAFE#);

   V_In_2 : Integer
      with Volatile,
           Input_Only,
           Address => System.Storage_Units.To_Address (16#ABCCAFE#);

   -- The following declarations are all external output only variables
   V_Out_1 : Volatile_Type
      with Output_Only,
           Address => System.Storage_Units.To_Address (16#BBCCAFE#);

   V_Out_2 : Integer
      with Volatile,
           Output_Only,
           Address => System.Storage_Units.To_Address (16#ADACAFE#);

   -- The following is a declaration of a non-volatile external variable
   V_Non_Volatile : Integer
      with Address => System.Storage_Units.To_Address (16#BEECAFE#);

end Multiple_Ports;
```

Todo

Add support for more complex models of external state. To be completed in a post-Release 1 version of this document.

7.1.4 Abstract_State Aspect

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 2014 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 a unit which is more visible than itself.

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              ::= External
                             | Input_Only
                             | Output_Only
name_value_option          ::= Part_Of => abstract_state
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 at most one of Input_Only or Output_Only `options` shall be specified in the `option_list`. The Input_Only and Output_Only options shall not be specified in an `option_list` without an External `option`.
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` shall have a completion [(require a body)] if it contains a non-null Abstract_State aspect specification. If a package declaration has a non-null Abstract_State aspect but in Ada a body is not required, a pragma Elaborate_Body shall be stated within the package declaration to make it required in Ada.
5. A subprogram declaration that overloads a state abstraction has an implicit Global aspect denoting the state abstraction with a `mode_selector` of Input. An explicit Global aspect may be specified which replaces the implicit one.

Static Semantics

1. 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.

Note: (SB) Making these implicit declarations overloadable allows declaring a subprogram with the same fully qualified name as a state abstraction; to make this scenario work, rules of the form "... shall denote a state abstraction" need to be name resolution rules, not just legality rules.

---

2. [A state abstraction shall only be named in contexts where this is explicitly permitted (e.g., as part of a Global aspect specification), but this is not a name resolution rule. Thus, the declaration of a state abstraction has the same visibility as any other declaration. 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.]

3. A **null** `abstract_state_list` specifies that a package contains no hidden state.
4. An External state abstraction is one declared with an `option_list` that includes the External `option` (see *External State*).
5. A state abstraction which is declared with an `option_list` that includes a Part_Of `name_value_option` indicates that it is a constituent (see *State Refinement*) exclusively of the state abstraction denoted by the `abstract_state` of the `name_value_option` (see *Abstract_State, Package Hierarchy and Part_Of*).

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 Q
   with Abstract_State => State          -- Declaration of abstract state named State
                                         -- representing internal state of Q.
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 Q.
      with Global => (Output => State),  -- State may be used in a global aspect.
           Post   => Is_Ready;

   procedure Op_1 (V : Integer)          -- Another procedure providing some operation on State
      with Global => (In_Out => State),
           Pre    => Is_Ready,
           Post   => Is_Ready;
end Q;

package X
   with Abstract_State => (A, B, (C with External, Input_Only))
        -- Three abstract state names are declared A, B & C.
        -- A and B are internal abstract states
        -- C is specified as external state which is input only.
is
   ...
end X;

package Mileage
   with Abstract_State => (Trip,   -- number of miles so far on this trip
                                   -- (can be reset to 0).
                           Total)  -- total mileage of vehicle since last factory-reset.
is
   function Trip  return Natural;  -- Has an implicit Global => Trip.
   function Total return Natural;  -- Has an implicit Global => Total.

   procedure Zero_Trip
      with Global  => (Output => Trip),  -- In the Global and Depends aspects
           Depends => (Trip => null),    -- Trip denotes the state abstraction.
           Post    => Trip = 0;          -- In the Post condition Trip denotes
                                         -- the function.
   procedure Inc
      with Global  => (In_Out => (Trip, Total)),
           Depends => ((Trip, Total) =>+ null),
```

```
            Post    => Trip = Trip'Old + 1 and Total = Total'Old + 1;

   -- Trip and Old in the Post conditions denote functions but these
   -- represent the state abstractions in Global and Depends specifications.

end Mileage;
```

7.1.5 Initializes Aspect

The Initializes aspect specifies the visible variables and state abstractions of a package that are initialized by the elaboration of the package. In SPARK 2014 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 Initializes aspect shall follow the Abstract_State aspect if one is present.
3. 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 variable declared immediately within the visible part of the package.
4. Each `name` in the `input_list` shall denote an entire variable or a state abstraction but shall not denote an entity declared in the package with the `aspect_specification` containing the Initializes aspect.
5. Each entity in a single `input_list` shall be distinct.

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. If an `initialization_item` has an `input_list` then the `names` in the list denote entities 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

1. 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.
2. If an `initialization_item` has an `input_list` then the entities denoted in the input list shall be used in determining the initialized value of the entity denoted by the `name` of the `initialization_item`.

Examples

```
package Q
   with Abstract_State => State,   -- Declaration of abstract state name State
        Initializes    => State    -- Indicates that State will be initialized
                                   -- during the elaboration of Q.
is
   ...
end Q;

package Y
   with Abstract_State => (A, B, (C with External, Input_Only)),
        -- Three abstract state names are declared A, B & C.
        Initializes    => A
        -- A is initialized during the elaboration of Y.
        -- C is specified as external input only state
        -- 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 Aspect

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`.
2. The Initial_Condition aspect shall follow the Abstract_State aspect and Initializes aspect if they are present.
3. Each variable or state abstraction denoted 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.

Static Semantics

1. An Initial_Condition aspect is a sort of postcondition for the elaboration of both the specification and body of a package. If present on a package, then its *Boolean*_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. 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

1. 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. This equivalence 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

1. [The Initial_Condition aspect gives a proof obligation to show that the implementation of the package_specification and its body satisfy the predicate given in the Initial_Condition aspect.]

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. 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 or absence 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 the visible part 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 visible part 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_Global, Refined_Depends, Refined_Pre and Refined_Post aspects are provided to express the contracts of a refined view of a subprogram.

7.2.2 Refined_State Aspect

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 } )
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

1. 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.]
2. If a package_specification has a non-null Abstract_State aspect its body shall have a Refined_State aspect.
3. If a package_specification does not have an Abstract_State aspect, then the corresponding package_body shall not have a Refined_State aspect.
4. Each constituent shall be either a variable or a state abstraction.
5. An object which is a constituent shall be an entire object.
6. A constituent shall denote an entity of the hidden state of a package or an entity which has a Part_Of option or aspect associated with its declaration.
7. Each *abstract*_state_name declared in the package specification shall be denoted as the state_name of a refinement_clause in the Refined_State aspect of the body of the package.
8. 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 are either variables declared in the private part or body of the package, or the declarations from the visible part of nested packages declared immediately therein.]

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

Static Semantics

1. 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`.
2. A **null** `constituent_list` indicates that the named abstract state has no constituents. 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

There are no verification rules associated with Refined_State aspect.

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;

-- 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 Abstract_State, Package Hierarchy and Part_Of

Each item of state declared in the visible part of a private library unit (and any descendants thereof) must be connected, directly or indirectly, to an *encapsulating* state abstraction of some public library unit. This is done using the Part_Of `option` or aspect associated with each declaration of the visible state of the private unit.

The unit declaring the encapsulating state abstraction identified by the Part_Of `option` or aspect needs not be its parent, but it must be a unit whose body has visibility of the private library unit, while being *more visible* than the original unit. Furthermore, the unit declaring the encapsulating state abstraction must denote the corresponding item of visible state in its Refined_State aspect to indicate that it includes this part of the visible state of the private unit.

That is, the two specifications, one in the private unit, and one in the body of the (typically) public unit, must match one another.

Hidden state declared in the private part of a unit also requires a Part_Of `option` or aspect, but it must be connected to an encapsulating state abstraction of the same unit.

The `option` or aspect Part_Of is used to specify the encapsulating state abstraction of the (typically) public unit with which a private unit's visible state item is associated.

To support multi-level hierarchies of private units, a private unit may connect its visible state to the state abstraction of another private unit, so long as eventually the state gets connected to the state abstraction of a public unit through a chain of connections. However, as indicated above, the unit through which the state is *exposed* must be more visible.

If a private library unit has visible state, this state might be read or updated as a side effect of calling a visible operation of a public library unit. This visible state may be referenced, either separately or as part of the state abstraction of some other public library unit. The following scenario gives rise to aliasing between the state abstraction and its constituents:

- a state abstraction is visible; and
- an object (or another state abstraction) is visible which is a constituent of the state abstraction; and
- it is not apparent that the object (or other state) is a constituent of the state abstraction - there are effectively two entities representing part or all of the state abstraction.

To resolve such aliasing, rules are imposed to ensure such a scenario can never occur. In particular, it is always known what state abstraction a constituent is part of and a state abstraction always knows all of its constituents.

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 applied to a variable declaration or a Part_Of aspect applied to a generic package instantiation. The Part_Of indicator shall denote the encapsulating state abstraction of which the declaration is a constituent.
2. A unit is more visible than another if it has less private ancestors.

Legality Rules

1. Every private unit and each of its descendants, that have visible state shall for each declaration in the visible state:
   - connect the declaration to an encapsulating state abstraction by associating a Part_Of indicator with the declaration; and
   - name an encapsulating state abstraction in its Part_Of indicator if and only if the unit declaring the state abstraction is strictly more visible than the unit containing the declaration.

   [Each state abstraction which has a Part_Of indicator, the unit in which it is declared and its encapsulating state is noted by any tool analyzing SPARK 2014.]
2. Each item of hidden state declared in the private part of a unit shall have a Part_Of indicator associated with the declaration which shall denote an encapsulating state abstraction of the same unit.
3. No other declarations shall have a Part_Of indicator.
4. The body of a unit whose specification declares a state abstraction named as an encapsulating state abstraction of a Part_Of indicator shall:
   - have a `with_clause` naming each unit, excluding itself, containing such a Part_Of indicator; and
   - in its Refined_State aspect, denote each declaration associated with such a Part_Of indicator as a `constituent` exclusively of the encapsulating state abstraction.

   [The state abstractions with a Part_Of indicator, the unit in which they have been declared and their encapsulating state have been noted as described previously and these records are used to check this rule.]

5. If both a state abstraction and one or more of its `constituents` are visible in a private package specification or in the package specification of a non-private descendant of a private package, then either the state abstraction or its `constituents` may be denoted but not within the same Global aspect or Depends aspect. The denotation must also be consistent between the Global and Depends aspects of a subprogram.
6. In a public package specification entities that are Part_Of an encapsulating state abstraction shall not be denoted; such entities may be represented by denoting their encapsulating state abstraction which is not Part_Of a more visible state abstraction. [This rule is applied recursively, if an entity is Part_Of a state abstraction which itself a Part_Of another encapsulating state abstraction, then it must be represented by the encapsulating state abstraction]. The exclusion to this rule is that for private parts of a package given below.
7. In the private part of a package a state abstraction declared by the package shall not be denoted other than for specifying it as the encapsulating state in the Part_Of indicator. The state abstraction's `constituents` declared in the private part shall be denoted.
8. In the body of a package, a state abstraction whose refinement is visible shall not be denoted except as an encapsulating state in a Part_Of indicator. Only its `constituents` may be denoted.
9. Within a package body where a state abstraction is visible, its refinement is not visible, but one or more of its `constituents` are visible, then the following rules apply:
   - either the state abstraction or its `constituents` may be denoted but not within the same Global aspect or Depends aspect. The denotation must also be consistent between the Global and Depends aspects of a subprogram.
   - a state abstraction denoted in a Global or Depends aspect is not refined into its constituents in a Refined_Global or Refined_Depends aspect [because the refinement of the state abstraction is not visible].

Verification Rules

1. In a package body of a public child when a state abstraction is visible, its refinement is not but one or more of its constituents are visible then if a subprogram declared in the visible part of the package, directly or indirectly:
   - reads a `constituent` of a state abstraction then, this shall be regarded as a read of the most visible encapsulating state abstraction of the `constituent` and shall be represented by this encapsulating state in the Global and Depends aspects of the subprogram; or
   - updates a `constituent` of a state abstraction then, this shall be regarded as an update of the most visible encapsulation state abstraction of the `constituent` and shall be represented by this encapsulating state with a `mode_selector` of In_Out in the Global aspect of the subprogram and as both an `input` and an `output` in the Depends aspect of the subprogram. [The reason for this is that it is not known whether the entire state abstraction is updated or only some of its constituents.] This rule does not apply when the most visible encapsulating state abstraction is External Input_Only or Output_Only. In this case the state abstraction shall have a `mode_selector` of Input for Input_Only states and Output for Output_Only states. Similarly in the Depends aspect Input_Only states shall be denoted only as `inputs` and Output_Only states shall be denoted only as `outputs`.

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

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 part of state abstraction 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;


package Outer
   with Abstract_State => (A1, A2)
is
   procedure Init_A1
      with Global  => (Output => A1),
           Depends => (A1 => null);

   procedure Init_A2
      with Global  => (Output => A2),
           Depends => (A2 => null);

private
   -- A variable declared in the private part must have a Part_Of aspect
   Hidden_State : Integer
      with Part_Of => A2;

   package Inner
      with Abstract_state => (B1 with Part_Of => Outer.A1)
                     -- State abstraction declared in the private
                     -- part must have a Part_Of option
                     -- A1 cannot be denoted in the private part.
   is
      procedure Init_B1
         with Global  => (Output => B1),
              Depends => (B1 => null);

      procedure Init_A2
         -- A2 cannot be denoted in the private part but
         -- Outer.Hidden_State, which is Part_Of A2, may be denoted.
         with Global  => (Output => Outer.Hidden_State),
              Depends => (Outer.Hidden_State => null);

   end Inner;
end Outer;
```

```
package body Outer
   with Refined_State => (A1 => Inner.B1,
                          A2 => Hidden_State)
                          -- Outer.A1 and Outer.A2 cannot be denoted in the
                          -- body of Outer because their refinements are visible.
is
   package body Inner
      with Refined_State => (B1 => null)  -- Oh, there isn't any state after all
   is
      procedure Init_B1
         with Refined_Global  => null,  -- Refined_Global and Refined_Depends of a null refinement
              Refined_Depends => null
      is
      begin
         null;
      end Init_B1;

      procedure Init_A2
         -- Refined_Global and Refined_Depends aspects not required
         -- because there is no refinement of Outer.Hidden_State.
      is
      begin
         Outer.Hidden_State := 0;
      end Init_A2;

   end Inner;

   procedure Init_A1
      with Refined_Global  => (Output => B1),
           Refined_Depends => (B1 => null)
   is
   begin
      Inner.Init_B1;
   end Init_A1;

   procedure Init_A2
      with Refined_Global  => (Output => Hidden_State),
           Refined_Depends => (Hidden_State => null)
   is
   begin
      Inner.Init_A2;
   end Init_A2;

end Outer;

package Q
   with Abstract_State => (Q1, Q2)
is
   -- Q1 and Q2 may be denoted here
   procedure Init_Q1
      with Global  => (Output => Q1),
           Depends => (Q1 => null);

   procedure Init_Q2
      with Global  => (Output => Q2),
           Depends => (Q2 => null);

private
```

```
   -- Q1 and Q2 may only be denoted as the encapsulating state abstraction
   Hidden_State : Integer
      with Part_Of => Q2;
end Q;

private package Q.Child
   with Abstract_State => (C1 with Part_Of => Q.Q1)
is
   -- Only constituents of Q1 and Q2 may be denoted here
   procedure Init_Q1
      with Global  => (Output => C1),
           Depends => (C1 => null);

   procedure Init_Q2
      with Global  => (Output => Q.Hidden_State),
           Depends => (Q.Hidden_State => null);
end Q.Child;

with Q;
package body Q.Child
   with Refined_State => (C1 => Actual_State)
is
   -- C1 shall not be denoted here - only Actual_State
   -- but Q.Hidden_State may be denoted.
   Actual_State : Integer;

   procedure Init_Q1
      with Refined_Global  => (Output => Actual_State),
           Refined_Depends => (Actual_State => null)
   is
   begin
      Actual_State := 0;
   end Init_Q1;

   procedure Init_Q2
   is
   begin
      Q.Hidden_State := 0;
   end Init_Q2;

end Q.Child;

with Q.Child;
package body Q
   with Refined_State => (Q1 => Q.Child.C1,
                          Q2 => Hidden_State)
is
   -- Q1 and Q2 shall not be denoted here but the constituents
   -- Q.Child.C1 and Hidden_State may be.

   procedure Init_Q1
      with Refined_Global  => (Output => Q.Child.C1),
           Refined_Depends => (Q.Child.C1 => null)
   is
   begin
      Q.Child.Init_Q1;
   end Init_Q1;
```

```
   procedure Init_Q2
      with Refined_Global  => (Output => Hidden_State),
           Refined_Depends => (Hidden_State => null)
   is
   begin
      Q.Child.Init_Q2;
   end Init_Q2;

end Q;
```

7.2.4 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] by their declaring package. [It follows that such constituents will appear in the initialization clause of the declaring unit unless they are external states.]

7.2.5 Refined_Global Aspect

A subprogram declared in the visible part 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.

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 Aspect*.

Static Semantics

The static semantics are equivalent to those given for the Global aspect in *Global Aspect*.

Legality Rules

1. A Refined_Global aspect shall be specified on a body_stub (if one is present) or subprogram body if and only if it has a declaration in the visible part of an enclosing package, the declaration has a Global aspect which denotes a state abstraction declared by the package and the refinement of the state abstraction is visible.
2. A Refined_Global aspect specification shall *refine* the subprogram's Global aspect as follows:
   - 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.

- 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, the Refined_Global specification shall be omitted, or if required by the syntax of a `global_specification` replaced by a **null** in the Refined_Global aspect.
- For each `global_item` in the Global aspect which does not denote such a 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.
- No other `global_items` shall be included in the Refined_Global aspect specification.

3. `Global_items` in a Refined_Global `aspect_specification` shall denote distinct entities.
4. The mode of each `global_item` in a Refined_Global aspect shall match that of the corresponding `global_item` in the Global aspect unless: the `mode_selector` specified in the Global aspect is In_Out; the corresponding `global_item` of Global aspect shall denote a state abstraction whose refinement is visible; and the `global_item` in the Refined_Global aspect is a `constituent` of the state abstraction.

   For this special case when the `mode_selector` is In_Out, the Refined_Global aspect may denote individual `constituents` of the state abstraction as Input, Output, or In_Out (given that the constituent itself may have any of these `mode_selectors`) so long as one or more of the following conditions are satisfied:

   - at least one of the `constituents` has a `mode_selector` of In_Out; or
   - there is at least one of each of a `constituent` with a `mode_selector` of Input and of Output; or
   - the Refined_Global aspect does not denote all of the `constituents` of the state abstraction but denotes at least one `constituent` that has a `mode_selector` of Output.

   [This rule ensures that a state abstraction with the `mode_selector` In_Out cannot be refined onto a set of `constituents` that are Output or Input only. The last condition satisfies this requirement because not all of the `constituents` are updated, some are preserved, that is the state abstraction has a self-dependency.]
5. If the Global aspect specification references a state abstraction with a `mode_selector` of Output, whose refinement is visible, then every `constituent` of that state abstraction shall be referenced in the Refined_Global aspect specification.
6. The legality rules for *Global Aspect* 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

1. If a subprogram has a Refined_Global aspect it is used in the analysis of the subprogram body rather than its Global aspect.
2. The verification rules given for *Global Aspect* also apply.

7.2.6 Refined_Depends Aspect

A subprogram declared in the visible part 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.

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 Aspect*.

Static Semantics

The static semantics are equivalent to those given for the Depends aspect in *Depends Aspect*.

Legality Rules

1. A Refined_Depends aspect shall be specified on a body_stub (if one is present) or subprogram body if and only if it has a declaration in the visible part 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.

2. 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 is defined by creating a new `dependency_relation` from the original given in the Depends aspect as follows:

   - 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.]

   - 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.

     If the `output` in the Depends_Aspect denotes a state abstraction which is not also an `input`, then all of the `constituents` [for a non-**null** refinement] of the state abstraction shall be denoted as `outputs` of the partially refined dependency relation.

     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` shall denote the same `inputs` that appear in the `input_list` of the original `output`.

   - 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.

   - The partially refined dependency relation is completed by replacing the `inputs` which are state abstractions, whose refinements are visible at the point of the Refined_Depends aspect, by zero or more `inputs`. It shall be zero only for a **null** refinement, otherwise each of the `inputs` shall denote a `constituent` of the state abstraction. The completed dependency relation is the `dependency_relation` of the Refined_Depends aspect.

3. 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.

4. 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.

5. [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` from the original `input_lists`.]

6. The rules for *Depends Aspect* 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

1. If a subprogram has a Refined_Depends aspect it is used in the analysis of the subprogram body rather than its Depends Aspect.
2. The verification rules given for *Depends Aspect* also apply.

7.2.7 Refined Precondition Aspect

A subprogram declared in the visible part of a package may have a Refined Precondition aspect applied to its body or body stub. The Refined Precondition may be used to restate a precondition 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 Precondition aspect is introduced by an `aspect_specification` where the `aspect_mark` is "Refined_Pre" and the `aspect_definition` shall be a Boolean `expression`.

Legality Rules

1. A Refined_Pre aspect may appear only on a body_stub (if one is present) or the body (if no stub is present) of subprogram if the subprogram is declared in the visible part of a package, its abstract view. If the subprogram declaration in the visible part has no explicit precondition, a precondition of True is assumed for its abstract view.
2. At the point of call of a subprogram, both its precondition and the expression of its Refined_Post aspect shall evaluate to True.
3. The same legality rules apply to a Refined Precondition as for a precondition.

Static Semantics

1. A Refined Precondition of a subprogram defines a *refinement* of the precondition of the subprogram.
2. The static semantics are otherwise as for a precondition.

Dynamic Semantics

1. When a subprogram with a Refined Precondition is called; first the precondition is evaluated as defined in the Ada RM. If the precondition evaluates to True, then the Refined Precondition is evaluated. If either precondition or Refined Precondition do not evaluate to True an exception is raised.

Verification Rules

1. The precondition of the abstract view of the subprogram shall imply its Refined_Precondition.

7.2.8 Refined Postcondition Aspect

A subprogram declared in the visible part of a package may have a Refined Postcondition aspect applied to its body or body stub. The Refined Postcondition may be used to restate a postcondition given on the declaration of a subprogram in terms the full view of a private type or the `constituents` of a refined `state_name`.

The Refined Postcondition 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 which is declared in the visible part of a package, its abstract view. If the subprogram declaration in the visible part has no explicit postcondition, a postcondition of True is assumed for the abstract view.
2. The same legality rules apply to a Refined Postcondition as for a postcondition.

Static Semantics

1. A Refined Postcondition of a subprogram defines a *refinement* of the postcondition of the subprogram.
2. 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.
3. The default Refined_Post for an expression function, F, is F'Result = `expression`, where `expression` is the expression defining the body of the function.
4. The static semantics are otherwise as for a postcondition.

Dynamic Semantics

1. When a subprogram with a Refined Postcondition is called; first the subprogram is evaluated. 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 True then the postcondition is evaluated as described in the Ada RM. If either the Refined Postcondition or the postcondition do not evaluate to True then the exception Assertions.Assertion_Error is raised.

Verification Rules

1. The precondition of a subprogram declaration with the Refined Precondition of its body or body stub and its Refined Postcondition together imply the postcondition of the declaration, that is:

   (Precondition and Refined Precondition and Refined Postcondition) -> Postcondition

---

Todo

refined contract_cases. To be completed in a post-Release 1 version of this document.

7.2.9 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, Input_Only state abstraction shall have only `constituents` that are External, Input_Only states.
3. An External, Output_Only state abstraction shall have only `constituents` that are External, Output_Only states.
4. A state abstraction that is specified as just External state, referred to as a *plain External state* may have `constituents` of any sort of External state and, or, non External states.
5. A subprogram declaration that has a Global aspect denoting a plain External state abstraction with a `mode_selector` other than In_Out, and the refinement of the state abstraction is visible at the point of the Refined_Global aspect, shall not denote a Volatile `constituent` of the state abstraction, in its Refined_Global aspect.

6. All other rules for Refined_State, Refined_Global and Refined_Depends aspect also apply.

Examples

```
package Externals
   with Abstract_State => ((Combined_Inputs with External, Input_Only),
                           (Displays with External, Output_Only),
                           (Complex_Device with External)),
        Initializes => Complex_Device
is
   procedure Read (Combined_Value : out Integer)
      with Global  => Combined_Inputs,  -- Combined_Inputs is an Input_Only
                                        -- External state; it can only be 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_Only
                                            -- External state; it can only be an
                                            -- Output in Global and Depends
                                            -- aspects.
           Depends => (Displays => (D_Main, D_Secondary));

   function Last_Value_Sent return Integer
      with Global => Complex_Device;  -- Complex_Device is a Plain External
                                      -- state.  It can be an Input and
                                      -- be a global to a function provided
                                      -- the Refined_Global aspect only
                                      -- refers to non-volatile or non-external
                                      -- constituents.

   procedure Output_Value (Value : in Integer)
      with Global  => (In_Out => Complex_Device),
           Depends => (Complex_Device => (Complex_Device, Value));
      -- If the refined Global Aspect refers to constituents which
      -- are volatile then the mode_selector for Complex_Device must
      -- be In_Out and it is both an input and an output.
      -- The subprogram must be a procedure.

end Externals;

private package Externals.Temperature
   with Abstract_State => (State with External, Input_Only,
                           Part_Of => Externals.Combined_Inputs)
is
   ...
end Externals.Temperature;

private package Externals.Pressure
   with Abstract_State => (State with External, Input_Only,
                           Part_Of => Externals.Combined_Inputs)
is
   ...
end Externals.Pressure;

private package Externals.Main_Display
   with Abstract_State => (State with External, Output_Only,
                           Part_Of => Externals.Displays)
is
```

```
   ...
end Externals.Main_Display;

private package Externals.Secondary_Display
   with Abstract_State => (State with External, Output_Only,
                           Part_Of => Externals.Displays)
is
  ...
end Externals.Secondary_Display;


with Externals.Temperature,
     Externals.Pressure,
     Externals.Main_Display,
     Externals.Secondary_Display;
package body Externals
   with Refined_State => (Combined_Inputs => (Externals.Temperature,
                                              Externals.Pressure),
                          -- Input_Only external state so both Temperature and
                          -- Pressure must be Input_Only.

                          Displays => (Externals.Main_Display,
                                       Externals.Secondary_Display),
                          -- Output_Only external state so both Main_Display and
                          -- Secondary_Display must be Output_Only.

                          Complex_Device => (Saved_Value,
                                             Out_Reg,
                                             In_Reg))
                          -- Complex_Device is a Plain External and may be
                          -- mapped to any sort of constituent.
is
   Saved_Value : Integer := 0;  -- Initialized as required.

   Out_Reg : Integer
      with Volatile,
           Output_Only,
           Address  => System.Storage_Units.To_Address (16#ACECAFE#);

   In_Reg : Integer
      with Volatile,
           Input_Only,
           Address  => System.Storage_Units.To_Address (16#A11CAFE#);

   function Last_Value_Sent return Integer
      with Refined_Global => Saved_Value -- Refined_Global aspect only
                                         -- refers to non external state
                                         -- as an Input.
   is
   begin
      return Saved_Value;
   end Last_Value_Sent;

   procedure Output_Value (Value : in Integer)
      with Refined_Global  => (Input  => In_Reg,
                               Output => Out_Reg,
                               In_Out => Saved_Value),
           -- Refined_Global aspect refers to both volatile
           -- state and non external state.
```

```
         Refined_Depends => ((Out_Reg,
                              Saved_Value) => (Saved_Value,
                                               Value),
                             null => In_Reg)
   is
      Ready  : constant Integer := 42;
      Status : Integer;
   begin
      if Saved_Value /= Value then
         loop
            Status := In_Reg;  -- In_Reg is Input_Only external state
                               -- and may appear on RHS of assignment
                               -- but not in a condition.
            exit when Status = Ready;
         end loop;

         Out_Reg := Value;  -- Out_Reg is an Output_Only external
                            -- state. Its value cannot be read.
         Saved_Value := Value;
      end if;
   end Output_Value;

   ...

end Externals;
```

7.3 Private Types and Private Extensions

The partial view of a private type or private extension may be in SPARK 2014 even if its full view is not in SPARK 2014. The usual rule applies here, so a private type without discriminants is in SPARK 2014, while a private type with discriminants is in SPARK 2014 only if its discriminants are in SPARK 2014.

7.3.1 Private Operations

No extensions or restrictions.

7.3.2 Type Invariants

The `aspect_specification` Type_Invariant is not permitted in SPARK 2014. [Type invariants are not currently supported in SPARK 2014 but are intended to be introduced in a future release.]

Todo

Add support for type invariants in SPARK 2014. To be completed in a post-Release 1 version of this document.

7.4 Deferred Constants

The view of an entity introduced by a `deferred_constant_declaration` is in SPARK 2014, even if the *initialization*`_expression` in the corresponding completion is not in SPARK 2014.

7.5 Limited Types

No extensions or restrictions.

7.6 Assignment and Finalization

Controlled types are not permitted in SPARK 2014.

7.7 Elaboration Issues

SPARK 2014 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 proof obligation 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.]

Legality Rules

1. SPARK 2014 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.
2. 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 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.

[TBD: it would be possible to relax this rule by defining a less-restrictive notion of preelaborability which allows, for example,

```
type Rec is record F1, F2 : Integer; end record;
X : constant Rec := (123, 456);  -- not preelaborable
```

while still disallowing the things that need to be disallowed and then defining the above rules in terms of this new notion instead of preelaborability. The only disadvantage of this is the added complexity of defining this new notion.]

3. 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.]

4. 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;

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"].

5. In the case of a dispatching call, the subprogram_body mentioned in the above rules is that (if any) of the statically denoted callee.

6. 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.]

7. For purposes of defining the early call region, the spec and body of a library unit package which has an Elaborate_Body pragma 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.

   [This example is in SPARK 2014, but would not be without the Elaborate_Body pragma (because of the tagged type rule).

```
with Other_Pkg;
package Pkg is
   pragma Elaborate_Body;
   type T is new Other_Pkg.Some_Tagged_Type with null record;
   overriding procedure Op (X : T);
   -- freezing point of T is here
end;

package body Pkg is
   ... ; -- only preelaborable constructs here
   procedure Op (X : T) is ... ;
end Pkg;
```

   An elaboration check failure would be possible if a call to Op (simple or via a dispatching call to an ancestor) were attempted between the elaboration of the spec and body of Pkg. The Elaborate_Body pragma prevents this from occurring. A library unit package spec which declares a tagged type will typically require an Elaborate_Body pragma.]

8. For the inter-compilation_unit case, SPARK 2014 enforces the follwing static elaboration order rule:

   - If a unit has elaboration code that can directly or indirectly make a call to a subprogram in a with'd unit, or instantiate a generic package in a with'd unit, then if the with'd unit does not have pragma Pure or Preelaborate, then the client should have a pragma Elaborate_All for the with'd unit. For generic subprogram instantiations, the rule can be relaxed to require only a pragma Elaborate. [This rule is the same as the GNAT static elaboration order rule as given in the GNAT Pro User's Guide.]

   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 body is in the current compilation_unit, or it is not. In the latter case, we require an Elaborate_All pragma as described above (the pragma must be given explicitly; it is not supplied implicitly).

   [Corner case notes: These rules correctly prohibit the following example:

```
package P is
   function F return Boolean;
   Flag : Boolean := F; -- would fail elab check
end;
```

The following dispatching-call-during-elaboration example would be problematic if the Elaborate_Body pragma were not required; with the pragma, the problem is solved because the elaboration order constraints are unsatisfiable:

```
package Pkg1 is
   type T1 is abstract tagged null record;
   function Op (X1 : T1) return Boolean is abstract;
end Pkg1;

with Pkg1;
package Pkg2 is
   pragma Elaborate_Body;
   type T2 is new Pkg1.T1 with null record;
   function Op (X2 : T2) return Boolean;
end Pkg2;

with Pkg1, Pkg2;
package Pkg3 is
   X : Pkg2.T2;
   Flag : Boolean := Pkg1.Op (Pkg1.T1'Class (X));
     -- dispatching call during elaboration fails check
     -- Note 'Class is not currently permitted.
end Pkg3;

with Pkg3;
package body Pkg2 is
   function Op (X2 : T2) return Boolean is
   begin return True; end;
end Pkg2;
```

9. For an instantiation of a generic which does not occur in the same compilation unit as the generic body, the rules are as described in the GNAT RM passage quoted above.

7.7.1 Use of Initial_Condition and Initializes Aspects

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 2014 which have the following consequences:

- 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
  - the variable (or its state abstraction) is mentioned in the Initializes aspect of its enclosing package; and
  - an Elaborate (not necessarily an Elaborate_All) pragma ensures that the body of that package has been elaborated.
- 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). 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 which applies an Elaborate pragma to the library unit in question (note: an Initial_Condition which depends on no variable inputs can also be assumed to be true throughout the execution of the main subprogram).
- 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. 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

1. 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 the compilation unit containing the read shall apply an Elaborate (not necessarily Elaborate_All) pragma to the unit declaring the variable or state abstraction. 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.]
2. 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 in output) declared outside of the package. The implicit write associated with a read of an external input only state 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

1. A package body shall include Elaborate pragmas for all of the other library units [(typically private children)] which provide constituents for state abstraction refinements occurring in the given package body. [This rule could be relaxed to apply only to constituents of an abstraction which is mentioned in an Initializes aspect.]

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.

8.4 Use Clauses

Use clauses are always in SPARK 2014, even if the unit mentioned is not completely in SPARK 2014.

8.5 Renaming Declarations

8.5.1 Object Renaming Declarations

An `object_renaming_declaration` for an entire object or a component of a record introduces a static alias of the renamed object. As the alias is static, in SPARK 2014 analysis it is replaced by the renamed object. This scheme works over multiple levels of renaming.

In an `object_renaming_declaration` which renames the result of a function the name of the declaration denotes a read only variable which is assigned the value of the function result from the elaboration of the `object_renaming_declaration`. This read only variable is used in SPARK 2014 analysis.

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

Syntax

There is no additional syntax associated with subprogram renaming declarations in SPARK 2014.

Legality Rules

1. The `aspect_specification` on a `subprogram_renaming_declaration` shall not include any of the SPARK 2014-defined aspects introduced in this document. [This restriction may be relaxed in the future.]

Static Semantics

There are no additional static semantics associated with subprogram renaming declarations in SPARK 2014.

Dynamic Semantics

There are no additional dynamic semantics associated with subprogram renaming declarations in SPARK 2014.

Verification Rules

There are no additional verification rules associated with subprogram renaming declarations in SPARK 2014.

[Note that, 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.]

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

Concurrent programs require the use of different specification and verification techniques from sequential programs. For this reason, tasks, protected units and objects, and synchronization features are currently excluded from SPARK 2014.

Todo

RCC: The above text implies that SPARK 2014 does not support Ada.Calendar, which is specified in RM 9.6. SPARK 2005 supports and prefers Ada.Real_Time and models the passage of time as an external "in" mode protected own variable. Should we use the same approach in SPARK 2014? Discussion under TN [LB07-024]. To be completed in the Milestone 4 version of this document.

Todo

Add Tasking. To be completed in a post-Release 1 version of this document.

CHAPTER

TEN

PROGRAM STRUCTURE AND COMPILATION ISSUES

SPARK 2014 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 needs 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 2014 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

A program unit cannot be a task unit, a protected unit or a protected entry.

10.1.1 Compilation Units

No restrictions or extensions.

10.1.2 Context Clauses - With Clauses

State abstractions are visible in the limited view of packages in SPARK 2014. The notion of an *abstract view* of a variable declaration is also introduced, and the limited view of a package includes the abstract view of any variables declared in the visible part of that package. The only allowed uses of an abstract view of a variable are where the use of a state abstraction would be allowed (for example, in a Global `aspect_specification`).

Syntax

There is no additional syntax associated with limited package views in SPARK 2014.

Legality Rules

1. A name denoting the abstract view of a variable shall occur only:
   - as a `global_item` in a Global or Refined_Global aspect specification; or
   - as an `input` or `output` in a Depends or Refined_Depends aspect specification.

Static Semantics

1. Any state abstractions declared within a given package are present in the limited view of the package. [This means that, for example, a Globals `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*.]
2. For every variable 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.
3. The abstract view of a volatile variable is external.

Dynamic Semantics

There are no additional dynamic semantics associated with limited package views in SPARK 2014.

Verification Rules

There are no verification rules associated with limited package views in SPARK 2014.

Note: (SB) No need to allow such a name in other contexts where a name denoting a state abstraction could be legal. In particular, in an Initializes aspect spec or in any of the various refinement aspect specifications. Initializes aspect specs do not refer to variables in other packages. Refinements occur in bodies and bodies don't need limited withs.

Note: (SB) Is the rule about volatility needed? I think this is needed in order to prevent a function's Global specification from mentioning an abstract view of a volatile variable, but I'm not sure because I don't understand what prevents a function's Global specification from mentioning the "concrete" view of a volatile variable. This problem is briefly mentioned at the beginning of the peculiarly numbered subsection 7.2 (package bodies) of section 7.2.4 (volatile variables).

With clauses are always in SPARK 2014, even if the unit mentioned is not completely in SPARK 2014.

10.1.3 Subunits of Compilation Units

No restrictions or extensions.

10.1.4 The Compilation Process

The analysis process in SPARK 2014 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 2014 analyses do not involve program execution. However, SPARK 2014 programs are executable including those new language defined aspects and pragmas where they have dynamic semantics given.

10.2.1 Elaboration Control

No extenstions or restrictions.

CHAPTER

ELEVEN

EXCEPTIONS

Syntax

There is no additional syntax associated with exceptions in SPARK 2014.

Legality Rules

1. Exception handlers are not in SPARK 2014. [Exception declarations (including exception renamings) are in SPARK 2014. Raise statements are in SPARK 2014, but must (as described below) be provably never executed.]
2. Raise expressions are not in SPARK 2014; for a raise statement to be in SPARK 2014, it must be immediately enclosed by an if statement which encloses no other statement. [It is intended that these two rules will be relaxed at some point in the future (this is why raise expressions are mentioned in the Verification Rules section below).]

Static Semantics

There are no additional static semantics associated with exceptions in SPARK 2014.

Dynamic Semantics

There are no additional dynamic semantics associated with exceptions in SPARK 2014.

Verification Rules

1. A `raise_statement` introduces an obligation to prove that the statement will not be executed, much like the proof obligation associated with

   ```
   pragma Assert (False);
   ```

   [In other words, the proof obligations introduced for a raise statement are the same as those introduced for a runtime check which fails unconditionally. A raise expression (see Ada AI12-0022 for details) introduces a similar obligation to prove that the expression will not be evaluated.]
2. The pragmas `Assertion_Policy`, `Suppress`, and `Unsuppress` are allowed in SPARK 2014, but have no effect on the generation of proof obligations. [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.]

CHAPTER

TWELVE

GENERIC UNITS

Any generic unit is in SPARK 2014, regardless of whether it contains constructs that are not normally in SPARK 2014. [Information flow analysis is not performed on a generic unit; a generic unit generates no proof obligations].

An instantiation of a generic is or is not in SPARK 2014 depending on whether the instance declaration and the instance body (described in section 12.3 of the Ada reference manual) are in SPARK 2014 [(i.e., when considered as a package (or, in the case of an instance of a generic subprogram, as a subprogram)].

[For example, a generic which takes a formal limited private type would be in SPARK 2014. An instantiation which passes in a task type as the actual type would not be in SPARK 2014; another instantiation of the same generic which passes in, for example, Standard.Integer, might be in SPARK 2014.]

[Ada has a rule that legality rules are not enforced in an instance body. No such rule applies to the restrictions defining which Ada constructs are in SPARK 2014. For example, a goto statement in an instance body would cause the instantiation to not be in SPARK 2014.]

[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. Other solutions involving "generative mode" (where flow-related aspect specifications are omitted in the source and generated implicitly by the tools) may also be available, but are outside of the scope of this document.]

[At some point in the future, a more sophisticated treatment of generics may be defined, allowing a generic to be "proven" and eliminating the need separately verify the correctness of each instantiation. That is not today's approach.]

[TBD: discsuss LSP-ish rules for globals, similar to the compatibility rules for Global/Depends aspects of a subprogram which overrides a dispatching operation. OK, for example, if a subprogram reads fewer inputs than it said it would.]

Todo

Update SPARK 2014 to allow prove once/use many approach to generics. To be completed in a post-Release 1 version of this document.

CHAPTER

THIRTEEN

REPRESENTATION ISSUES

Todo

Provide full detail on Representation Issues. To be completed in a post-Release 1 version of this document.

Todo

This statement was originally in this chapter "Pragma or aspect `Unchecked_Union` is not in SPARK 2014" this needs to be recorded in the list of unsupported aspects and pragmas. To be completed in the Milestone 4 version of this document.

13.1 Operational and Representation Aspects

No restrictions or additions.

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

The use of the operators defined for type Address are not permitted in SPARK 2014 except for use within representation clauses.

13.8 Machine Code Insertions

Machine code insertions are not in SPARK 2014.

13.9 Unchecked Type Conversions

An instantiation of an Unchecked_Conversion may have Refined_Pre and Refined_Post aspects specified.

Todo

Provide a detailed semantics for Refined_Pre and Refined_Post aspects on Unchecked_Conversion. To be completed in a post-Release 1 version of this document.

13.9.1 Data Validity

[Currently SPARK 2014 does not check for data validity as it analyses code, though this may be changed in a future release. It is therefore up to users to ensure that data read from external sources is valid.]

Todo

Need to put some words in here to describe the precautions that may be taken to avoid invalid data. To be completed in the Milestone 4 version of this document.

Todo

Introduce checks for data validity into the proof model as necessary. To be completed in a post-Release 1 version of this document.

13.10 Unchecked Access Value Creation

As access types are not supported in SPARK 2014, neither is this attribute.

13.11 Storage Management

These features are related to access types and not in SPARK 2014.

13.12 Pragma Restrictions and Pragma Profile

Restrictions and Profiles will be available with SPARK 2014 to provide profiles suitable for different application environments.

13.13 Streams

Stream types and operations are not in SPARK 2014.

13.14 Freezing Rules

No restrictions or additions.

CHAPTER

FOURTEEN

SHARED VARIABLE CONTROL (ANNEX C.6)

The following restrictions are applied to the declaration of volatile types and objects in SPARK 2014:

Legality Rules

1. A volatile representation aspect may only be applied to an `object_declaration` or a `full_type_declaration`.
2. A component of a non-volatile type declaration shall not be volatile.

   [This may require determining whether a private type is volatile.]

   [The above two rules may be relaxed in a future version.]
3. A discriminant shall not be of a volatile type.
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. A volatile variable shall not be declared within the body of a function, directly or indirectly.

CHAPTER

FIFTEEN

THE STANDARD LIBRARIES

This chapter describes how SPARK 2014 treats the Ada predefined language environment and standard libraries, corresponding to appendices A through H of the Ada RM. The goal is that SPARK 2014 programs are able to use as much as possible of the the Ada predefined language environment and standard libraries.

Todo

Provide detail on Standard Libraries. To be completed in a post-Release 1 version of this document. This targeting applies to all ToDos in this chapter.

Todo

In particular, it is intended that predefined container generics suitable for use in SPARK 2014 will be provided. These will have specifications as similar as possible to those of Ada's bounded containers (i.e., Ada.Containers.Bounded_*), but with constructs removed or modified as needed in order to maintain the language invariants that SPARK 2014 relies upon in providing formal program verification.

15.1 Predefined Language Environment

15.1.1 The Package Standard

SPARK 2014 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.

15.1.2 The Package Ada

Todo

Should we say here which packages are supported in SPARK 2014 or which ones aren't supported? How much of the standard library will be available, and in which run-time profiles?

15.2 Interface to Other Languages

This section describes features for mixed-language programming in SPARK 2014, covering facilities offered by Ada's Annex B.

Todo

How much to say here? S95 supports a subset of Interfaces, and a very small subset of Interfaces.C but that's about it.

Todo

What is status of supported for pragma `Unchecked_Union` in GNATProve at present?

15.3 Systems Programming

tbd.

15.4 Real-Time Systems

This section describes features for real-time programming in SPARK 2014, covering facilities offered by Ada's Annex D.

Todo

RCC: Need to think about Ada.Real_Time. It's important for all S95 customers, to get at monotonic clock, even if not using RavenSPARK. It does depend on support for external variables, though, since Ada.Real_Time.Clock is most definitely Volatile. TN [LB07-024] raised to discuss this.

15.5 Distributed Systems

TBD.

15.6 Information Systems

TBD.

15.7 Numerics

This section describes features for numerical programming in SPARK 2014, covering facilities offered by Ada's Annex G.

Todo

How much here can be supported? Most S95 customers want Ada.Numerics.Generic_Elementary_Functions plus its predefined instantiation for Float, Long_Float and so on. How far should we go?

15.8 High Integrity Systems

SPARK 2014 fully supports the requirements of Ada's Annex H.

APPENDIX

A

SPARK 2005 TO SPARK 2014 MAPPING SPECIFICATION

This appendix defines the mapping between SPARK 2005 and SPARK 2014. It is intended as both a completeness check for the SPARK 2014 language design, and as a guide for projects upgrading from SPARK 2005 to SPARK 2014.

A.1 Subprogram patterns

A.1.1 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:

```
1  package Swap_Add_05
2  --# own X, Y: Integer;
3  is
4
5     procedure Swap;
6     --# global in out X, Y;
7     --# derives X from Y &
8     --#         Y from X;
9
10    function Add return Integer;
11    --# global in X, Y;
12
13 end Swap_Add_05;
```

Specification in SPARK 2014:

```
1  package Swap_Add_14
2     with Abstract_State => (X, Y)
3  is
4     procedure Swap
5        with Global  => (In_Out => (X, Y)),
6             Depends => (X => Y,    -- to be read as "X depends on Y"
7                         Y => X);   -- to be read as "Y depends on X"
8
9     function Add return Integer
```

```
10          with Global  => (Input => (X, Y));
11    end Swap_Add_14;
```

A.1.2 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 they have been augmented by 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 globals. *Divide* returns the division of the two globals 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. For the same reasons as in the previous example, the bodies are not included.

Specification in SPARK 2005:

```
1   package Swap_Add_Max_05
2   --# own X, Y: Integer;
3   is
4
5      subtype Index      is Integer range 1..100;
6      type    Array_Type is array (Index) of Integer;
7
8      procedure Swap;
9      --# global in out X, Y;
10     --# derives X from Y &
11     --#         Y from X;
12     --# post X = Y~ and Y = X~;
13
14     function Add return Integer;
15     --# global in X, Y;
16     --# return X + Y;
17
18     function Max return Integer;
19     --# global in X, Y;
20     --# return Z => (X >= Y -> Z = X) and
21     --#              (Y >  X -> Z = Y);
22
23     function Divide return Float;
24     --# global in X, Y;
25     --# pre Y /= 0;
26     --# return Float(X / Y);
27
28     procedure Swap_Array_Elements(A: in out Array_Type);
29     --# global in X, Y;
30     --# derives A from A, X, Y;
31     --# pre  X in Index and Y in Index;
32     --# post A = A~[X => A~(Y); Y => A~(X)];
33
34  end Swap_Add_Max_05;
```

Specification in SPARK 2014:

```
1   package Swap_Add_Max_14
2      with Abstract_State => (X, Y)
3   is
4      subtype Index      is Integer range 1..100;
5      type    Array_Type is array (Index) of Integer;
```

```
6
7     procedure Swap
8        with Global  => (In_Out => (X, Y)),
9             Depends => (X => Y,
10                        Y => X),
11            Post    => (X = Y'Old and Y = X'Old);
12
13    function Add return Integer
14       with Global  => (Input => (X, Y)),
15            Post    => Add'Result = X + Y;
16
17    function Max return Integer
18       with Global  => (Input => (X, Y)),
19            Post    => (if X >= Y then Max'Result = X
20                        else Max'Result = Y);
21
22    function Divide return Float
23       with Global  => (Input => (X, Y)),
24            Pre     => Y /= 0,
25            Post    => Divide'Result = Float(X / Y);
26
27    procedure Swap_Array_Elements(A: in out Array_Type)
28       with Global  => (Input => (X, Y)),
29            Depends => (A => (A, X, Y)),
30            Pre     => X in Index and Y in Index,
31            Post    => A = A'Old'Update (X => A'Old (Y), Y => A'Old (X));
32 end Swap_Add_Max_14;
```

A.1.3 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 analyser also needs to be smart about this, since it knows the X'First and X'Last are well-defined in the body, even though the content of X is not.

Specification in SPARK 2005:

```
1  package P
2  is
3     type A is array (Positive range <>) of Integer;
4
5     -- Shows that X'First and X'Last _can_ be used in
6     -- precondition here, even though X is mode "out"...
7     procedure Init (X : out A);
8     --# derives X from ;
9     --# pre X'First <= 2  and
10    --#     X'Last  >= 20;
11    --# post for all I in Positive range X'Range => (X (I) = 0);
12
13 end P;
```

Body in SPARK 2005:

```
1  package body P is
2
3     procedure Init (X : out A) is
4     begin
5        --# accept F,  23, X,    "OK" &
6        --#        F, 602, X, X, "OK";
```

```
      for I in Positive range X'Range loop
         X (I) := 0;
         --# assert for all J in Positive range X'First .. I => (X (J) = 0);
      end loop;
   end Init;

end P;
```

Specification in SPARK 2014:

```
package P
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 Depends => (X => null),
           Pre     => X'First <= 2 and X'Last >= 20,
           Post    => (for all I in X'Range => (X (I) = 0));
end P;
```

Todo

Depending on the outcome of M423-014, either pragma Annotate or pragma Warning will be utilized to accept warnings/errors in SPARK 2014. To be completed in the Milestone 4 version of this document.

Body in SPARK 2014:

```
package body P is

   procedure Init (X : out A) is
   begin
      --  SPARK 2005 example uses accept annotation here:
      --  corresponding syntax is TBD.
      for I in Positive range X'Range loop
         X (I) := 0;
         pragma Loop_Invariant (for all J in X'First .. I => (X (J) = 0));
      end loop;
   end Init;

end P;
```

A.1.4 Nesting of subprograms, including more refinement

This example demonstrates how procedures and functions can be nested within other procedures and functions. Furthermore, it illustrates how global variables 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;
   Y := 20;
end Nesting_Refinement_05;
```

Specification in SPARK 2014:

```
package Nesting_Refinement_14
   with 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 Refined_State => (State => (X, Y))
is
   X, Y: Integer;

   procedure Operate_On_State
```

```
8          with Refined_Global => (In_Out => X,
9                                  Output => Y)
10      is
11         Z: Integer;
12
13         procedure Add_Z_To_X
14            with Global => (In_Out => X,
15                            Input  => Z)
16         is
17         begin
18            X := X + Z;
19         end Add_Z_To_X;
20
21         procedure Overwrite_Y_With_Z
22            with Global => (Output => Y,
23                            Input  => Z)
24         is
25         begin
26            Y := Z;
27         end Overwrite_Y_With_Z;
28
29      begin
30         Z := 5;
31         Add_Z_To_X;
32         Overwrite_Y_With_Z;
33      end Operate_On_State;
34
35   begin -- Promised to initialize State
36         -- (which consists of X and Y)
37      X := 10;
38      Y := 20;
39   end Nesting_Refinement_14;
```

A.2 Package patterns

A.2.1 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:

```
1  package Stacks_05 is
2     Stack_Size : constant := 100;
3     type Pointer_Range is range 0 .. Stack_Size;
4     subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
5     type Vector is array(Index_Range) of Integer;
6
7     type Stack is
8        record
9           Stack_Vector : Vector;
```

```
10          Stack_Pointer : Pointer_Range;
11       end record;
12
13    function Is_Empty(S : Stack) return Boolean;
14    function Is_Full(S : Stack) return Boolean;
15
16    procedure Clear(S : out Stack);
17    procedure Push(S : in out Stack; X : in Integer);
18    procedure Pop(S : in out Stack; X : out Integer);
19 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:

```
1  package Stacks_05 is
2
3     type Stack is private;
4
5     function Is_Empty(S : Stack) return Boolean;
6     function Is_Full(S : Stack) return Boolean;
7
8     procedure Clear(S : out Stack);
9     procedure Push(S : in out Stack; X : in Integer);
10    procedure Pop(S : in out Stack; X : out Integer);
11
12 private
13    Stack_Size : constant := 100;
14    type Pointer_Range is range 0 .. Stack_Size;
15    subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
16    type Vector is array(Index_Range) of Integer;
17
18    type Stack is
19       record
20          Stack_Vector : Vector;
21          Stack_Pointer : Pointer_Range;
22       end record;
23 end Stacks_05;
```

Private type with refined pre/post contracts in the body

This example demonstrates how *pre* and *post* conditions, that lie in the specification of a package, can be refined in the package's body. Contracts that need not be refined, do not have to be repeated in the body of a package. In this particular example, the body of the SPARK 2005 might seem to be needlessly repeating contracts. However, this is not true since the contracts that are being repeated are indirectly being refined through the refinement of the *Is_Empty* and *Is_Full* functions.

Specification in SPARK 2005:

```
1  package Stacks_05
2  is
3
4     type Stack is private;
```

```
5
6       function Is_Empty(S : Stack) return Boolean;
7       function Is_Full(S : Stack) return Boolean;
8
9       procedure Clear(S : in out Stack);
10      --# post Is_Empty(S);
11      procedure Push(S : in out Stack; X : in Integer);
12      --# pre  not Is_Full(S);
13      --# post not Is_Empty(S);
14      procedure Pop(S : in out Stack; X : out Integer);
15      --# pre  not Is_Empty(S);
16      --# post not Is_Full(S);
17
18   private
19      Stack_Size : constant := 100;
20      type    Pointer_Range is range 0 .. Stack_Size;
21      subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
22      type    Vector        is array(Index_Range) of Integer;
23
24      type Stack is
25         record
26            Stack_Vector  : Vector;
27            Stack_Pointer : Pointer_Range;
28         end record;
29   end Stacks_05;
```

Body in SPARK 2005:

```
1    package body Stacks_05 is
2
3       function Is_Empty (S : Stack) return Boolean
4       --# return S.Stack_Pointer = 0;
5       is
6       begin
7          return S.Stack_Pointer = 0;
8       end Is_Empty;
9
10      function Is_Full (S : Stack) return Boolean
11      --# return S.Stack_Pointer = Stack_Size;
12      is
13      begin
14         return S.Stack_Pointer = Stack_Size;
15      end Is_Full;
16
17      procedure Clear (S : in out Stack)
18      --# post Is_Empty(S);
19      is
20      begin
21         S.Stack_Pointer := 0;
22      end Clear;
23
24      procedure Push (S : in out Stack; X : in Integer)
25      --# pre  not Is_Full(S);
26      --# post not Is_Empty(S) and
27      --#      S.Stack_Pointer = S~.Stack_Pointer + 1 and
28      --#      S.Stack_Vector = S~.Stack_Vector[S.Stack_Pointer => X];
29      is
30      begin
31         S.Stack_Pointer := S.Stack_Pointer + 1;
```

```
32          S.Stack_Vector (S.Stack_Pointer) := X;
33       end Push;
34
35       procedure Pop (S : in out Stack; X : out Integer)
36       --# pre  not Is_Empty(S);
37       --# post not Is_Full(S) and
38       --#      X = S.Stack_Vector(S~.Stack_Pointer) and
39       --#      S.Stack_Pointer = S~.Stack_Pointer - 1 and
40       --#      S.Stack_Vector = S~.Stack_Vector;
41       is
42       begin
43          X := S.Stack_Vector (S.Stack_Pointer);
44          S.Stack_Pointer := S.Stack_Pointer - 1;
45       end Pop;
46    end Stacks_05;
```

Specification in SPARK 2014:

```
1  package Stacks_14
2  is
3
4     type Stack is private;
5
6     function Is_Empty(S : Stack) return Boolean;
7     function Is_Full(S : Stack) return Boolean;
8
9     procedure Clear(S : in out Stack)
10       with Post => Is_Empty(S);
11    procedure Push(S : in out Stack; X : in Integer)
12       with Pre  => not Is_Full(S),
13            Post => not Is_Empty(S);
14    procedure Pop(S : in out Stack; X : out Integer)
15       with Pre  => not Is_Empty(S),
16            Post => not Is_Full(S);
17
18 private
19    Stack_Size : constant := 100;
20    type    Pointer_Range is range 0 .. Stack_Size;
21    subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
22    type    Vector        is array(Index_Range) of Integer;
23
24    type Stack is
25       record
26          Stack_Vector  : Vector;
27          Stack_Pointer : Pointer_Range;
28       end record;
29 end Stacks_14;
```

Body in SPARK 2014:

```
1  package body Stacks_14 is
2     function Is_Empty(S : Stack) return Boolean
3        with Refined_Post => Is_Empty'Result = (S.Stack_Pointer = 0)
4     is
5     begin
6        return S.Stack_Pointer = 0;
7     end Is_Empty;
8
9     function Is_Full(S : Stack) return Boolean
```

```
10          with Refined_Post => Is_Full'Result = (S.Stack_Pointer = Stack_Size)
11       is
12       begin
13          return S.Stack_Pointer = Stack_Size;
14       end Is_Full;
15
16       procedure Clear(S : in out Stack)
17          with Refined_Post => Is_Empty(S)
18       is
19       begin
20          S.Stack_Pointer := 0;
21       end Clear;
22
23       procedure Push(S : in out Stack; X : in Integer)
24          with Refined_Pre  => S.Stack_Pointer /= Stack_Size,
25               Refined_Post => (S.Stack_Pointer = S'Old.Stack_Pointer + 1 and
26                                S.Stack_Vector = S'Old.Stack_Vector'Update(S.Stack_Pointer => X))
27       is
28       begin
29          S.Stack_Pointer := S.Stack_Pointer + 1;
30          S.Stack_Vector(S.Stack_Pointer) := X;
31       end Push;
32
33       procedure Pop(S : in out Stack; X : out Integer)
34          with Refined_Pre  => S.Stack_Pointer /= 0,
35               Refined_Post => (X = S.Stack_Vector(S'Old.Stack_Pointer) and
36                                S.Stack_Pointer = S'Old.Stack_Pointer - 1 and
37                                S.Stack_Vector = S'Old.Stack_Vector)
38       is
39       begin
40          X := S.Stack_Vector(S.Stack_Pointer);
41          S.Stack_Pointer := S.Stack_Pointer - 1;
42       end Pop;
43    end Stacks_14;
```

Public child extends non-tagged parent ADT

The following example covers the main differences between a child package and an arbitrary package, namely:

- The private part of a child package can access the private part of its parent.
- The body of a child package can access the private part of its parent.
- The child does not need a with clause for its parent.

A private type and private constant are declared in the parent. The former is accessed in the body of the child, while the latter is accessed in the private part of the child.

Specifications of both parent and child in SPARK 2005:

```
1  package Pairs_05 is
2
3     type Pair is private;
4
5     -- Sums the component values of a Pair.
6     function Sum (Value : in Pair) return Integer;
7
8  private
9
```

```
10       type Pair is
11          record
12             Value_One : Integer;
13             Value_Two : Integer;
14          end record;
15
16        Inc_Value : constant Integer := 1;
17
18    end Pairs_05;
```

```
1     --#inherit Pairs_05;
2
3     package Pairs_05.Additional_05
4     is
5
6        --  Additional operation to add to the ADT, which
7        --  increments each value in the Pair.
8        procedure Increment (Value: in out Pairs_05.Pair);
9        --# derives Value from Value;
10
11    private
12
13       --  Variable declared to illustrate access to private part of
14       --  parent from private part of child.
15       Own_Inc_Value : constant Integer := Pairs_05.Inc_Value;
16
17    end Pairs_05.Additional_05;
```

Bodies of both parent and child in SPARK 2005:

```
1     package body Pairs_05
2     is
3
4        function Sum (Value : in Pair) return Integer
5        is
6        begin
7             return Value.Value_One + Value.Value_Two;
8        end Sum;
9
10    end Pairs_05;
```

```
1     package body Pairs_05.Additional_05
2     is
3
4        procedure Increment (Value: in out Pairs_05.Pair) is
5        begin
6           -- Access to private part of parent from body of public child.
7           Value.Value_One := Value.Value_One + Own_Inc_Value;
8           Value.Value_Two := Value.Value_Two + Own_Inc_Value;
9        end Increment;
10
11    end Pairs_05.Additional_05;
```

Specifications of both parent and child in SPARK 2014:

```
1     package Pairs_14 is
2
3     --  No change to parent.
4
```

```
   type Pair is private;

   -- Sums the component values of a Pair.
   function Sum (Value : in Pair) return Integer;

private

   type Pair is
      record
         Value_One : Integer;
         Value_Two : Integer;
      end record;

    Inc_Value : constant Integer := 1;

end Pairs_14;
```

```
package Pairs_14.Additional_14
is

   --  Additional operation to add to the ADT, which
   --  increments each value in the Pair.
   procedure Increment (Value: in out Pairs_14.Pair);
      with Depends => (Value => Value);

private

   --  Variable declared to illustrate access to private part of
   --  parent from private part of child.
   Own_Inc_Value : constant Integer := Pairs_14.Inc_Value;

end Pairs_14.Additional_14;
```

Bodies of both parent and child in SPARK 2014:

As per SPARK 2005.

Tagged type in root ADT package

The following example illustrates the use of a tagged type in an ADT package.

Specification in SPARK 2005:

```
package Stacks_05 is

   type Stack is tagged private;

   function Is_Empty(S : Stack) return Boolean;
   function Is_Full(S : Stack) return Boolean;

   procedure Clear(S : out Stack);
   --# derives S from ;

   procedure Push(S : in out Stack; X : in Integer);
   --# derives S from *, X;

   procedure Pop(S : in out Stack; X : out Integer);
   --# derives S, X from S;
```

```
16
17 private
18    Stack_Size : constant := 100;
19    type Pointer_Range is range 0 .. Stack_Size;
20    subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
21    type Vector is array(Index_Range) of Integer;
22    type Stack is tagged
23       record
24          Stack_Vector : Vector;
25          Stack_Pointer : Pointer_Range;
26       end record;
27 end Stacks_05;
```

Body in SPARK 2005:

N/A

Specification in SPARK 2014:

```
1  package Stacks_14 is
2
3     type Stack is tagged private;
4
5     function Is_Empty(S : Stack) return Boolean;
6     function Is_Full(S : Stack) return Boolean;
7
8     procedure Clear(S : out Stack)
9        with Depends => (S => null);
10
11    procedure Push(S : in out Stack; X : in Integer)
12       with Depends => (S =>+ X);
13    --  The =>+ symbolizes that any variable on the left side of =>+,
14    --  depends on all variables that are on the right side of =>+
15    --  plus itself. For example (X, Y) =>+ Z would mean that
16    --  X depends on X, Z and Y depends on Y, Z.
17
18    procedure Pop(S : in out Stack; X : out Integer)
19       with Depends => ((S,X) => S);
20
21 private
22    Stack_Size : constant := 100;
23    type Pointer_Range is range 0 .. Stack_Size;
24    subtype Index_Range is Pointer_Range range 1 .. Stack_Size;
25    type Vector is array(Index_Range) of Integer;
26
27    type Stack is tagged
28       record
29          Stack_Vector : Vector;
30          Stack_Pointer : Pointer_Range;
31       end record;
32 end Stacks_14;
```

Body in SPARK 2014:

N/A

Extension of tagged type in child package ADT

The following example illustrates the extension of a tagged type in a child package.

Specification in SPARK 2005:

```
--# inherit Stacks_05;
package Stacks_05.Monitored_05 is

   type Monitored_Stack is new Stacks_05.Stack with private;

   overriding
   procedure Clear(S : out Monitored_Stack);
   --# derives S from ;

   overriding
   procedure Push(S : in out Monitored_Stack; X : in Integer);
   --# derives S from S, X;

   function Top_Identity(S : Monitored_Stack) return Integer;
   function Next_Identity(S : Monitored_Stack) return Integer;

private

   type Monitored_Stack is new Stacks_05.Stack with
      record
         Monitor_Vector : Stacks_05.Vector;
         Next_Identity_Value : Integer;
      end record;

end Stacks_05.Monitored_05;
```

Body in SPARK 2005:

```
package body Stacks_05.Monitored_05 is
   subtype Index_Range is Stacks_05.Index_Range;

   overriding
   procedure Clear(S : out Monitored_Stack) is
   begin
      S.Stack_Pointer := 0;
      S.Stack_Vector := Stacks_05.Vector'(Index_Range => 0);
      S.Next_Identity_Value := 1;
      S.Monitor_Vector := Stacks_05.Vector'(Index_Range => 0);
   end Clear;

   overriding
   procedure Push(S : in out Monitored_Stack; X : in Integer) is
   begin
      Stacks_05.Push(Stacks_05.Stack(S), X);
      S.Monitor_Vector(S.Stack_Pointer) := S.Next_Identity_Value;
      S.Next_Identity_Value := S.Next_Identity_Value + 1;
   end Push;

   function Top_Identity(S : Monitored_Stack) return Integer is
      Result : Integer;
   begin
      if Is_Empty(S) then
         Result := 0;
      else
         Result := S.Monitor_Vector(S.Stack_Pointer);
      end if;
      return Result;
```

```
30      end Top_Identity;
31
32      function Next_Identity(S : Monitored_Stack) return Integer is
33      begin
34         return S.Next_Identity_Value;
35      end Next_Identity;
36
37   end Stacks_05.Monitored_05;
```

Specification in SPARK 2014:

```
1    package Stacks_14.Monitored_14 is
2
3       type Monitored_Stack is new Stacks_14.Stack with private;
4
5       overriding
6       procedure Clear(S : out Monitored_Stack)
7          with Depends => (S => null);
8
9       overriding
10      procedure Push(S : in out Monitored_Stack; X : in Integer)
11         with Depends => (S =>+ X);
12
13      function Top_Identity(S : Monitored_Stack) return Integer;
14      function Next_Identity(S : Monitored_Stack) return Integer;
15
16   private
17
18      type Monitored_Stack is new Stacks_14.Stack with
19         record
20            Monitor_Vector : Stacks_14.Vector;
21            Next_Identity_Value : Integer;
22         end record;
23
24   end Stacks_14.Monitored_14;
```

Body in SPARK 2014:

As per SPARK 2005.

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 Ada2012's visibility rules. No restrictions have been applied in terms of visibility and thus no SPARK 2014 code is provided in this section.

Specification of parent in SPARK 2005:

```
package Parent_05
is
  function F 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
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
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
is
  pragma Elaborate_Body(Public_Child_A_05);
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
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 return Integer is separate;
end Parent_05;
```

Body of public child A in SPARK 2005:

```
package body Parent_05.Public_Child_A_05
is
  procedure Proc(I : in out Integer)
  --#derives I from I;
  is
  begin
   I := I + Parent_05.F;  -- OK
  end Proc;
end Parent_05.Public_Child_A_05;
```

A.2.2 Abstract State Machines (ASMs)

Visible, concrete state

Initialized by declaration

The example that follows presents a way of initializing a concrete state (a state that cannot be refined) at the point of the declaration of the variables that compose it. This can only be done in SPARK 2005. In SPARK 2014 state abstractions cannot share names with variables and concequently cannot be implicitly refined.

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;

end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
   with 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));
```

```
7
8     procedure Pop(X : out Integer)
9        with Global => (Input  => S_State,
10                       In_Out => Pointer_State);
11 end Stack_14;
```

Body in SPARK 2014:

```
1  package body Stack_14
2     with Refined_State => (S_State       => S,
3                            Pointer_State => Pointer)
4  is
5     Stack_Size : constant := 100;
6     type    Pointer_Range is range 0 .. Stack_Size;
7     subtype Index_Range   is Pointer_Range range 1..Stack_Size;
8     type    Vector        is array(Index_Range) of Integer;
9
10    S : Vector := Vector'(Index_Range => 0);  -- Initialization of S
11    Pointer : Pointer_Range := 0;             -- Initialization of Pointer
12
13    procedure Push(X : in Integer)
14       with Refined_Global => (In_Out => (S, Pointer))
15    is
16    begin
17       Pointer := Pointer + 1;
18       S(Pointer) := X;
19    end Push;
20
21    procedure Pop(X : out Integer)
22       with Refined_Global => (Input  => S,
23                               In_Out => Pointer)
24    is
25    begin
26       X := S(Pointer);
27       Pointer := Pointer - 1;
28    end Pop;
29
30 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:

```
1  package body Stack_05
2  is
3     Stack_Size : constant := 100;
4     type    Pointer_Range is range 0 .. Stack_Size;
5     subtype Index_Range   is Pointer_Range range 1..Stack_Size;
6     type    Vector        is array(Index_Range) of Integer;
7
8     S : Vector;
9     Pointer : Pointer_Range;
10
11    procedure Push(X : in Integer)
```

```
12      is
13      begin
14         Pointer := Pointer + 1;
15         S(Pointer) := X;
16      end Push;
17
18      procedure Pop(X : out Integer)
19      is
20      begin
21         X := S(Pointer);
22         Pointer := Pointer - 1;
23      end Pop;
24
25   begin  -- initialization
26      Pointer := 0;
27      S := Vector'(Index_Range => 0);
28   end Stack_05;
```

Body in SPARK 2014:

As per SPARK 2005.

Private, concrete state

The following example demonstrates how variables, that need to be hidden from the users of a package, can be placed on the package's private section. The SPARK 2005 body has not been included since it does not contain any annotations.

Specification in SPARK 2005:

```
1   package Stack_05
2   --# own S, Pointer;
3   is
4      procedure Push(X : in Integer);
5      --# global in out S, Pointer;
6
7      procedure Pop(X : out Integer);
8      --# global in     S;
9      --#        in out Pointer;
10  private
11     Stack_Size : constant := 100;
12     type    Pointer_Range is range 0 .. Stack_Size;
13     subtype Index_Range   is Pointer_Range range 1..Stack_Size;
14     type    Vector        is array(Index_Range) of Integer;
15
16     S : Vector;
17     Pointer : Pointer_Range;
18  end Stack_05;
```

Specification in SPARK 2014:

```
1   package Stack_14
2      with Abstract_State => (S_State, Pointer_State)
3   is
4      procedure Push(X : in Integer)
5         with Global => (In_Out => (S_State, Pointer_State));
6
7      procedure Pop(X : out Integer)
```

```
8        with Global => (Input  => S_State,
9                        In_Out => Pointer_State);
10
11 private
12    Stack_Size : constant := 100;
13    type    Pointer_Range is range 0 .. Stack_Size;
14    subtype Index_Range   is Pointer_Range range 1..Stack_Size;
15    type    Vector        is array(Index_Range) of Integer;
16
17    S : Vector
18       with Part_Of => S_State;
19    Pointer : Pointer_Range
20       with Part_Of => Pointer_State;
21 end Stack_14;
```

Body in SPARK 2014:

```
1  package body Stack_14
2     with Refined_State => (S_State       => S,
3                            Pointer_State => Pointer)
4  is
5     procedure Push(X : in Integer)
6        with Refined_Global => (In_Out => (S, Pointer))
7     is
8     begin
9        Pointer := Pointer + 1;
10       S(Pointer) := X;
11    end Push;
12
13    procedure Pop(X : out Integer)
14       with Refined_Global => (Input  => S,
15                               In_Out => Pointer)
16    is
17    begin
18       X := S(Pointer);
19       Pointer := Pointer - 1;
20    end Pop;
21 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:

```
1  package Stack_05
2  --# own State;
3  is
4     procedure Push(X : in Integer);
5     --# global in out State;
6
7     procedure Pop(X : out Integer);
8     --# global in out State;
9
```

```
10     procedure Init;
11     --# global    out State;
12  private
13     Stack_Size : constant := 100;
14     type    Pointer_Range is range 0 .. Stack_Size;
15     subtype Index_Range   is Pointer_Range range 1..Stack_Size;
16     type    Vector        is array(Index_Range) of Integer;
17
18     type Stack_Type is
19        record
20           S : Vector;
21           Pointer : Pointer_Range;
22        end record;
23  end Stack_05;
```

Body in SPARK 2005:

```
1   package body Stack_05
2   --# own State is Stack;
3   is
4      Stack : Stack_Type;
5
6      procedure Push(X : in Integer)
7      --# global in out Stack;
8      is
9      begin
10        Stack.Pointer := Stack.Pointer + 1;
11        Stack.S(Stack.Pointer) := X;
12     end Push;
13
14     procedure Pop(X : out Integer)
15     --# global in out Stack;
16     is
17     begin
18        X := Stack.S(Stack.Pointer);
19        Stack.Pointer := Stack.Pointer - 1;
20     end Pop;
21
22     procedure Init
23     --# global    out Stack;
24     is
25     begin
26        Stack.Pointer := 0;
27        Stack.S := Vector'(Index_Range => 0);
28     end Init;
29  end Stack_05;
```

Specification in SPARK 2014:

```
1   package Stack_14
2      with Abstract_State => State
3   is
4      procedure Push(X : in Integer)
5         with Global => (In_Out => State);
6
7      procedure Pop(X : out Integer)
8         with Global => (In_Out => State);
9
10     procedure Init
```

```
      with Global => (Output => 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;

   type Stack_Type is
      record
         S : Vector;
         Pointer : Pointer_Range;
      end record;
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
   with Refined_State => (State => Stack)
is
   Stack : Stack_Type;

   procedure Push(X : in Integer)
      with Refined_Global => (In_Out => Stack)
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
      with Refined_Global => (In_Out => Stack)
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;

   procedure Init
      with Refined_Global => (Output => Stack)
   is
   begin
      Stack.Pointer := 0;
      Stack.S := Vector'(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;
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_Type is
      record
         S : Vector;
         Pointer : Pointer_Range;
      end record;
end Stack_05;
```

Body in SPARK 2005:

```
package body Stack_05
--# own State is Stack; -- refinement of state
is
   Stack : Stack_Type := Stack_Type'(Pointer => 0, S => Vector'(Index_Range => 0));
   -- initialization by elaboration of declaration

   procedure Push(X : in Integer)
   --# global in out Stack;
   is
   begin
      Stack.Pointer := Stack.Pointer + 1;
      Stack.S(Stack.Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   --# global in out Stack;
   is
   begin
      X := Stack.S(Stack.Pointer);
      Stack.Pointer := Stack.Pointer - 1;
   end Pop;
end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
   with 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);
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;
```

```
15
16      type Stack_Type is
17         record
18            S : Vector;
19            Pointer : Pointer_Range;
20         end record;
21   end Stack_14;
```

Body in SPARK 2014:

```
1   package body Stack_14
2      with Refined_State => (State => Stack) -- refinement of state
3   is
4      Stack : Stack_Type := Stack_Type'(Pointer => 0, S => Vector'(Index_Range => 0));
5      -- initialization by elaboration of declaration
6
7      procedure Push(X : in Integer)
8         with Refined_Global => (In_Out => Stack)
9      is
10     begin
11        Pointer := Pointer + 1;
12        S(Pointer) := X;
13     end Push;
14
15     procedure Pop(X : out Integer)
16        with Refined_Global => (In_Out => Stack)
17     is
18     begin
19        X := S(Pointer);
20        Pointer := Pointer - 1;
21     end Pop;
22  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:

```
1   package body Stack_05
2   --# own State is Stack;  -- refinement of state
3   is
4      Stack : Stack_Type;
5
6      procedure Push(X : in Integer)
7      --# global in out Stack;
8      is
9      begin
10        Stack.Pointer := Stack.Pointer + 1;
11        Stack.S(Stack.Pointer) := X;
12     end Push;
13
14     procedure Pop(X : out Integer)
15     --# global in out Stack;
16     is
17     begin
```

```
18          X := Stack.S(Stack.Pointer);
19          Stack.Pointer := Stack.Pointer - 1;
20       end Pop;
21    begin  -- initialized by package body statements
22       Stack.Pointer := 0;
23       Stack.S := Vector'(Index_Range => 0);
24    end Stack_05;
```

Body in SPARK 2014:

```
1   package body Stack_14
2      with Refined_State => (State => Stack)  -- refinement of state
3   is
4      Stack: Stack_Type;
5
6      procedure Push(X : in Integer)
7         with Refined_Global => (In_Out => Stack)
8      is
9      begin
10        Pointer := Pointer + 1;
11        S(Pointer) := X;
12     end Push;
13
14     procedure Pop(X : out Integer)
15        with Refined_Global => (In_Out => Stack)
16     is
17     begin
18        X := S(Pointer);
19        Pointer := Pointer - 1;
20     end Pop;
21  begin  -- initialized by package body statements
22     Stack.Pointer := 0;
23     Stack.S := Vector'(Index_Range => 0);
24  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:

```
1   package Stack_05
2   --# own Stack;
3   --# initializes Stack;
4   is
5      procedure Push(X : in Integer);
6      --# global in out Stack;
7
8      procedure Pop(X : out Integer);
9      --# global in out Stack;
10  private
11     Stack_Size : constant := 100;
12     type    Pointer_Range is range 0 .. Stack_Size;
13     subtype Index_Range   is Pointer_Range range 1..Stack_Size;
14     type    Vector        is array(Index_Range) of Integer;
15  end Stack_05;
```

Body in SPARK 2005:

```
package body Stack_05
--# own Stack is S, Pointer; -- state refinement
is
   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 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);
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;
end Stack_14;
```

Body in SPARK 2014:

```
package body Stack_14
   with Refined_State => (Stack => (S, Pointer)) -- state refinement
is
   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
```

```
12          Pointer := Pointer + 1;
13          S(Pointer) := X;
14       end Push;
15
16       procedure Pop(X : out Integer)
17          with Refined_Global => (Input  => S,
18                                  In_Out => Pointer)
19       is
20       begin
21          X := S(Pointer);
22          Pointer := Pointer - 1;
23       end Pop;
24    begin  -- partial initialization by body statements
25       S := Vector'(Index_Range => 0);
26    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. The body is not being provided since it does not add any further insight.

Specification in SPARK 2014:

```
1  package stack_14
2     with Abstract_State     => State,
3          Initializes        => State,
4          Initial_Condition => Is_Empty  -- Stating that Is_Empty holds
5                                         -- after initialization
6  is
7     function Is_Empty return Boolean
8        with Global => State;
9
10    function Is_Full return Boolean
11       with Global => State;
12
13    function Top return Integer
14       with Global => State,
15            Pre    => not Is_Empty;
16
17    procedure Push (X: in Integer)
18       with Global => (In_Out => State),
19            Pre    => not Is_Full,
20            Post   => Top = X;
21
22    procedure Pop (X: out Integer)
23        with Global => (In_Out => State),
24             Pre    => not Is_Empty;
25 end stack_14;
```

Private, abstract state, refining onto concrete state of private child

The following example shows a parent package Power that contains a State own variable. This own variable is refined onto concrete state contained within the 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;
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);
     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;
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;
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
is
   State : Integer;

   procedure Read (Level : out Integer)
   is
   begin
      Level := State;
   end Read;
end Power_05.Source_A_05;
```

```
package body Power_05.Source_B_05
is
   State : Integer;

   procedure Read (Level : out Integer)
   is
   begin
      Level := State;
   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 Abstract_State => 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 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 Abstract_State => (State with Part_Of => Power_14.State)
is
   procedure Read (Level : out Integer)
      with Global => State,
           Depends => (Level => State);
end Power_14.Source_A_14;
```

```
private package Power_14.Source_B_14
   with Abstract_State => (State with Part_Of => Power_14.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 Refined_State => (State => S)
is
   S : Integer;

   procedure Read (Level : out Integer)
      with Refined_Global  => S,
           Refined_Depends => (Level => S)
   is
   begin
      Level := S;
   end Read;
end Power_14.Source_A_14;
```

```
package body Power_14.Source_A_14
   with Refined_State => (State => S);
is
   S : Integer;

   procedure Read (Level : out Integer)
      with Refined_Global  => 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 private child, with the private child packages converted into embedded packages.

Specification in SPARK 2005:

```
-- Use of embedded packages to encapsulate state
package Power_05
--# own State;
is
   procedure Read_Power(Level : out Integer);
```

```
6       --# global State;
7       --# derives Level from State;
8    end Power_05;
```

Body in SPARK 2005:

```
1    package body Power_05
2    --# own State is Source_A.State,
3    --#              Source_B.State;
4    is
5
6      --  Embedded package spec for Source_A
7      package Source_A
8      --# own State;
9      is
10        procedure Read (Level : out Integer);
11         --# global State;
12         --# derives Level from State;
13     end Source_A;
14
15     --  Embedded package spec for Source_B.
16     package Source_B
17     --# own State;
18     is
19       procedure Read (Level : out Integer);
20       --# global State;
21       --# derives Level from State;
22     end Source_B;
23
24     --  Embedded package body for Source_A
25     package body Source_A
26     is
27       State : Integer;
28
29       procedure Read (Level : out Integer)
30       is
31       begin
32         Level := State;
33       end Read;
34     end Source_A;
35
36     --  Embedded package body for Source_B
37     package body Source_B
38     is
39       State : Integer;
40
41       procedure Read (Level : out Integer)
42       is
43       begin
44         Level := State;
45       end Read;
46
47     end Source_B;
48
49     procedure Read_Power(Level : out Integer)
50     --# global Source_A.State, Source_B.State;
51     --# derives
52     --#     Level
53     --#     from
```

```
54    --#         Source_A.State,
55    --#         Source_B.State;
56    is
57       Level_A : Integer;
58       Level_B : Integer;
59    begin
60       Source_A. Read (Level_A);
61       Source_B.Read (Level_B);
62       Level := Level_A + Level_B;
63    end Read_Power;
64
65 end Power_05;
```

Specification in SPARK 2014:

```
1  -- Use of embedded packages to encapsulate state
2  package Power_14
3     with Abstract_State => State
4  is
5     procedure Read_Power(Level : out Integer)
6        with Global  => State,
7             Depends => (Level => State);
8  end Power_14;
```

Body in SPARK 2014:

```
1  package body Power_14
2     with Refined_State => (State => (Source_A.State, Source_B.State))
3  is
4
5     --  Embedded package spec for Source_A
6     package Source_A
7        with Abstract_State => State
8     is
9        procedure Read (Level : out Integer)
10          with Global  => State,
11               Depends => (Level => State);
12    end Source_A;
13
14    --  Embedded package spec for Source_B.
15    package Source_B
16    with
17       Abstract_State => State
18    is
19      procedure Read (Level : out Integer)
20         with Global  => State,
21              Depends => (Level => State);
22    end Source_B;
23
24    --  Embedded package body for Source_A
25    package body Source_A
26       with Refined_State => (State => S)
27    is
28      S : Integer;
29
30      procedure Read (Level : out Integer)
31         with Refined_Global  => S,
32              Refined_Depends => (Level => S)
33      is
```

```
34      begin
35        Level := S;
36      end Read;
37    end Source_A;
38
39    --  Embedded package body for Source_B
40    package body Source_B
41       with Refined_State => (State => S)
42    is
43      S : Integer;
44
45      procedure Read (Level : out Integer)
46         with Refined_Global  => S,
47              Refined_Depends => (Level => S)
48      is
49      begin
50        Level := S;
51      end Read;
52
53    end Source_B;
54
55    procedure Read_Power(Level : out Integer)
56       with Refined_Global  => (Source_A.State, Source_B.State),
57            Refined_Depends => (Level => (Source_A.State, Source_B.State))
58    is
59       Level_A : Integer;
60       Level_B : Integer;
61    begin
62       Source_A. Read (Level_A);
63       Source_B.Read (Level_B);
64       Level := Level_A + Level_B;
65    end Read_Power;
66
67  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 concrete 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.2.3 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 specification of the stack package are not being presented since they are identical to previous examples.

Specification of main program in SPARK 2005:

```
1  with Input_Port_05, Output_Port_05, Stacks_05;
2  --# inherit Input_Port_05, Output_Port_05, Stacks_05;
3  --# main_program;
4  procedure Copy_05
```

```
5  --# global in     Input_Port_05.Input_State;
6  --#        out    Output_Port_05.Output_State;
7  --# derives Output_Port_05.Output_State from Input_Port_05.Input_State;
8  is
9     The_Stack   : Stacks_05.Stack;
10    Value       : Integer;
11    Done        : Boolean;
12    Final_Value : constant Integer := 999;
13 begin
14    Stacks_05.Clear(The_Stack);
15    loop
16       Input_Port_05.Read_From_Port(Value);
17       Stacks_05.Push(The_Stack, Value);
18       Done := Value = Final_Value;
19       exit when Done;
20    end loop;
21    loop
22       Stacks_05.Pop(The_Stack, Value);
23       Output_Port_05.Write_To_Port(Value);
24       exit when Stacks_05.Is_Empty(The_Stack);
25    end loop;
26 end Copy_05;
```

Specification of input port in SPARK 2005:

```
1  package Input_Port_05
2    --# own in Input_State;
3  is
4     procedure Read_From_Port(Input_Value : out Integer);
5     --# global in Input_State;
6     --# derives Input_Value from Input_State;
7
8  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 16#CAFE#;
   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:

```
1  package Output_Port_05
2    --# own out Output_State;
3  is
4     procedure Write_To_Port(Output_Value : in Integer);
5     --# global out Output_State;
6     --# 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 16#CAFE#;
   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;
--  Approach for identifying main program is TBD.
procedure Copy_14
   with 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 Abstract_State => (Input_State with External, Input_Only)
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 Abstract_State => (Output_State with External, Output_Only)
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.

```
package body Input_Port_14
   with Refined_State => (Input_State => Input_S)
is
   Input_S : Integer
      with Address => 16#CAFE#,
           Volatile;

   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.

```
package body Output_Port_14
   with Refined_State => (Output_State => Output_S)
is
   Output_S : Integer
      with Address => 16#CAFE#,
           Volatile;

   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 ‘Append and ‘Tail contracts

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).

Todo

There will not be an equivalent of ‘Append and ‘Tail in SPARK 2014. However, we will be able to achieve the same functionality using generics. To be completed in the Milestone 4 version of this document.

Specification in SPARK 2005:

```
1  package Input_Port
2    --# own in Inputs : Integer;
3  is
4     procedure Read_From_Port(Input_Value : out Integer);
5     --# global in Inputs;
6     --# derives Input_Value from Inputs;
7     --# post Input_Value = Inputs~ and Inputs = Inputs'Tail (Inputs~);
8
9  end Input_Port;
```

Body in SPARK 2005:

```
package body Input_Port
is

   Inputs : Integer;
   for Inputs'Address use 16#CAFE#;
   --# assert Inputs'Always_Valid;
   pragma Volatile (Inputs);

   procedure Read_From_Port(Input_Value : out Integer)
   is
   begin
      Input_Value := Inputs;
   end Read_From_Port;

end Input_Port;
```

Output driver using 'Append and 'Tail contracts

This example uses the Output package from section Basic Input and Output Device Drivers and adds a contract using the 'Append attribute.

Todo

There will not be an equivalent of 'Append and 'Tail in SPARK 2014. However, we will be able to achieve the same functionality using generics. To be completed in the Milestone 4 version of this document.

Specification in SPARK 2005:

```
1  package Output_Port
2    --# own out Outputs : Integer;
3  is
4     procedure Write_To_Port(Output_Value : in Integer);
5     --# global out Outputs;
6     --# derives Outputs from Output_Value;
7     --# post Outputs = Outputs'Append (Outputs~, Output_Value);
8  end Output_Port;
```

Body in SPARK 2005:

```
package body Output_Port
is
```

```
      Outputs : Integer;
      for Outputs'Address use 16#CAFE#;
      pragma Volatile (Outputs);

      procedure Write_To_Port(Output_Value : in Integer)
      is
      begin
         Outputs := Output_Value;
      end Write_To_Port;

   end Output_Port;
```

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:

```
1  package Switch
2  --# own in State;
3  is
4
5     type Reading is (on, off, unknown);
6
7     function ReadValue return Reading;
8     --# global in State;
9
10 end Switch;
```

Component Switch specifications in SPARK 2005:

```
1  --# inherit Switch;
2  private package Switch.Val1
3  --# own in State;
4  is
5     function Read return Switch.Reading;
6     --# global in State;
7
8  end Switch.Val1;
```

```
1  --# inherit Switch;
2  private package Switch.Val2
3  --# own in State;
4  is
5     function Read return Switch.Reading;
6     --# global in State;
7
8  end Switch.Val2;
```

```
1  --# inherit Switch;
2  private package Switch.Val3
3  --# own in State;
4  is
5     function Read return Switch.Reading;
6     --# global in State;
7
8  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) +
          ConvertToValue (B) + ConvertToValue (C));
   end ReadValue;

end Switch;
```

Abstract Switch specification in SPARK 2014:

```
package Switch
   with Abstract_State => (State with External, Input_Only)
is
   type Reading is (on, off, unknown);

   function ReadValue return Reading
      with Global => (Input => State);
end Switch;
```

Component Switch specifications in SPARK 2014:

```
private package Switch.Val1
   with Abstract_State => (State with External, Input_Only,
                                      Part_Of => Switch.State)
is
   function Read return Switch.Reading
      with Global => (Input => State);
end Switch.Val1;
```

```
private package Switch.Val2
   with Abstract_State => (State with External, Input_Only,
                                      Part_Of => Switch.State)
is
   function Read return Switch.Reading
      with Global => (Input => State);
end Switch.Val2;
```

```
private package Switch.Val3
   with Abstract_State => (State with External, Input_Only,
                           Part_Of => Switch.State)
is
   function Read return Switch.Reading
      with Global => (Input => State);
end Switch.Val3;
```

Switch body in SPARK 2014:

```
with Switch.Val1;
with Switch.Val2;
with Switch.Val3;
package body Switch
   -- State is refined onto three states, each of which has properties Volatile and Input
   with Refined_State => (State => (Switch.Val1.State,
                                    Switch.Val2.State,
                                    Switch.Val2.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 ad an output external variable.

Specification in SPARK 2005:

```
1  package Device
2  --# own State;
3  --# initializes State;
4  is
5    procedure Write (X : in Integer);
6    --# global in out State;
7    --# derives State from State, X;
8  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
  OldX : Integer := 0; -- only component that needs initialization
  StatusPort : Integer;
  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 : Integer;
  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
      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 Abstract_State => State,
        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 Refined_State => (State => (OldX,
                                    StatusPort,
                                    Register))
   -- refinement on to mix of external and ordinary variables
is
  OldX : Integer := 0; -- only component that needs initialization
  StatusPort : Integer
     with Volatile,
          Input_Only;
  -- address clause would be added here

  Register : Integer
     with Volatile,
          Output_Only;
  -- address clause would be added here

  procedure WriteReg (X : in Integer)
     with Refined_Global  => (Output => Register),
          Refined_Depends => (Register => X)
  is
  begin
    Register := X;
  end WriteReg;

  procedure ReadAck (OK : out Boolean)
     with Refined_Global  => (Input => StatusPort),
          Refined_Depends => (OK => StatusPort)
  is
    RawValue : Integer;
  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.

Todo

There will not be an equivalent of 'Append and 'Tail in SPARK 2014. However, we will be able to achieve the same functionality using generics. To be completed in the Milestone 4 version of this document.

Specification in SPARK 2005:

```
1  package Inc
2  --# own in Sensor;
3  is
4     pragma Elaborate_Body (Inc);
5  end Inc;
```

Body in SPARK 2005:

```
package body Inc
--# own Sensor is in S;
is
   S : Integer;
   for S'Address use 16#DEADBEEF#;
   pragma Volatile (S);

   procedure Read (V     : out Integer;
                   Valid : out Boolean)
   --# global in S;
```

```
      --# post (Valid -> V = S~) and
      --#      (S = S'Tail (S~));
      is
         Tmp : Integer;
      begin
         Tmp := S;
         if Tmp'Valid then
            V := Tmp;
            Valid := True;
            --# check S = S'Tail (S~);
         else
            V := 0;
            Valid := False;
         end if;
      end Read;

      procedure Increases (Result : out Boolean;
                           Valid  : out Boolean)
      --# global in S;
      --# post Valid -> (Result <-> S'Tail (S~) > S~);
      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;
```

A.2.4 Package Inheritance

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:

```
1  package Raw_Data
2  --# own State;
3  is
4
5     --# function Data_Is_Valid (Value : Integer) return Boolean;
6
7     procedure Read (Value : out Integer);
8     --# global in State;
9     --# derives Value from State;
10    --# post Data_Is_Valid (Value);
```

```
11
12 end Raw_Data;
```

```
1  with Raw_Data;
2  --# inherit Raw_Data;
3  package Processing
4  --# own State;
5  is
6
7     procedure Read_Processed_Data (Value : out Integer);
8     --# global in State, Raw_Data.State;
9     --# derives Value from State, Raw_Data.State;
10    --# post Raw_Data.Data_Is_Valid (Value);
11
12 end Processing;
```

```
1  with Processing;
2  --# inherit Processing, Raw_Data;
3  package Calculate
4  is
5
6     procedure Read_Calculated_Value (Value : out Integer);
7     --# global in Processing.State, Raw_Data.State;
8     --# derives Value from Processing.State, Raw_Data.State;
9     --# post Raw_Data.Data_Is_Valid (Value);
10
11 end Calculate;
```

Specifications in SPARK 2014:

```
1  package Raw_Data
2     with Abstract_State => State
3  is
4
5     function Data_Is_Valid (Value : Integer) return Boolean
6        with Convention => Ghost;
7
8     procedure Read (Value : out Integer)
9        with Global  => (Input => State),
10            Depends => (Value => State),
11            Post    => Data_Is_Valid (Value);
12
13 end Raw_Data;
```

```
1  with Raw_Data;
2  package Processing
3     with Abstract_State => State
4  is
5
6     procedure Read_Processed_Data (Value : out Integer)
7        with Global  => (Input => (State, Raw_Data.State)),
8             Depends => (Value => (State, Raw_Data.State)),
9             Post    => Raw_Data.Data_Is_Valid (Value);
10
11 end Processing;
```

```
1  with Processing;
2  package Calculate
3  is
```

```
4
5       procedure Read_Calculated_Value (Value : out Integer)
6          with Global  => (Input => (Processing.State, Raw_Data.State)),
7               Depends => (Value => (Processing.State, Raw_Data.State)),
8               Post    => Raw_Data.Data_Is_Valid (Value);
9
10   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:

```
1   package Switch
2   --# own in State;
3   is
4
5      type Reading is (on, off, unknown);
6
7      function ReadValue return Reading;
8      --# global in State;
9
10  end Switch;
```

Component Switch specifications in SPARK 2005:

As in Refinement of external state - voting input switch

Switch body in SPARK 2005:

```
1   with Switch.Val1;
2   with Switch.Val2;
3   with Switch.Val3;
4   package body Switch
5   --# own State is in Switch.Val1.State,
6   --#              in Switch.Val2.State,
7   --#              in Switch.Val3.State;
8   is
9
10     subtype Value is Integer range -1 .. 1;
11     subtype Score is Integer range -3 .. 3;
12
13
14     function ReadValue return Reading
15     --# global in Val1.State;
16     --#        in Val2.State;
17     --#        in Val3.State;
18     is
19        A, B, C : Reading;
20
21        --  Embedded package to provide the capability to synthesize three inputs
```

```
22          --  into one.
23          --# inherit Switch;
24          package Conversion
25          is
26
27             function Convert_To_Reading
28                (Val_A : Switch.Reading;
29                 Val_B : Switch.Reading;
30                 Val_C : Switch.Reading) return Switch.Reading;
31
32          end Conversion;
33
34          package body Conversion
35          is
36
37             type ConvertToValueArray is array (Switch.Reading) of Switch.Value;
38             type ConvertToReadingArray is array (Switch.Score) of Switch.Reading;
39             ConvertToValue : constant ConvertToValueArray := ConvertToValueArray'(Switch.on => 1,
40                                                                                   Switch.unknown => 0,
41                                                                                   Switch.off => -1);
42
43             ConvertToReading : constant ConvertToReadingArray :=
44                                        ConvertToReadingArray'(-3 .. -2 => Switch.off,
45                                                               -1 .. 1  => Switch.unknown,
46                                                               2 ..3    => Switch.on);
47
48             function Convert_To_Reading
49                (Val_A : Switch.Reading;
50                 Val_B : Switch.Reading;
51                 Val_C : Switch.Reading) return Switch.Reading
52             is
53             begin
54
55                return ConvertToReading (ConvertToValue (Val_A) +
56                       ConvertToValue (Val_B) + ConvertToValue (Val_C));
57             end Convert_To_Reading;
58
59          end Conversion;
60
61       begin
62           A := Val1.Read;
63           B := Val2.Read;
64           C := Val3.Read;
65           return Conversion.Convert_To_Reading
66                    (Val_A => A,
67                     Val_B => B,
68                     Val_C => C);
69       end ReadValue;
70
71    end Switch;
```

Abstract Switch specification in SPARK 2014:

```
1  package Switch
2     with Abstract_State => (State with External, Input_Only)
3  is
4     type Reading is (on, off, unknown);
5
```

```
6     function ReadValue return Reading
7        with Global => (Input => State);
8  end Switch;
```

Component Switch specification in SPARK 2014:

As in Refinement of external state - voting input switch

Switch body in SPARK 2014:

```
1  with Switch.Val1;
2  with Switch.Val2;
3  with Switch.Val3;
4  package body Switch
5     --  State is refined onto three states, each of which has properties
6     --  Volatile and Input.
7     with Refined_State => (State => (Switch.Val1.State,
8                                      Switch.Val2.State,
9                                      Switch.Val3.State))
10 is
11    subtype Value is Integer range -1 .. 1;
12    subtype Score is Integer range -3 .. 3;
13
14    function ReadValue return Reading
15       with Refined_Global => (Input => (Val1.State, Val2.State, Val3.State))
16    is
17       A, B, C : Reading;
18
19       --  Embedded package to provide the capability to synthesize three inputs
20       --  into one.
21       package Conversion
22       is
23          function Convert_To_Reading
24             (Val_A : Switch.Reading;
25              Val_B : Switch.Reading;
26              Val_C : Switch.Reading) return Switch.Reading;
27       end Conversion;
28
29       package body Conversion
30       is
31          type ConvertToValueArray is array (Switch.Reading) of Switch.Value;
32          type ConvertToReadingArray is array (Switch.Score) of Switch.Reading;
33          ConvertToValue : constant ConvertToValueArray := ConvertToValueArray'(Switch.on => 1,
34                                                                                Switch.unknown => 0,
35                                                                                Switch.off => -1);
36
37          ConvertToReading : constant ConvertToReadingArray :=
38                                        ConvertToReadingArray'(-3 .. -2 => Switch.off,
39                                                               -1 .. 1  => Switch.unknown,
40                                                               2  .. 3  => Switch.on);
41
42          function Convert_To_Reading
43             (Val_A : Switch.Reading;
44              Val_B : Switch.Reading;
45              Val_C : Switch.Reading) return Switch.Reading
46          is
47          begin
48             return ConvertToReading (ConvertToValue (Val_A) +
49                       ConvertToValue (Val_B) + ConvertToValue (Val_C));
```

```
50            end Convert_To_Reading;
51
52         end Conversion;
53      begin  --  begin statement of ReadValue function
54         A := Val1.Read;
55         B := Val2.Read;
56         C := Val3.Read;
57         return Conversion.Convert_To_Reading
58                   (Val_A => A,
59                    Val_B => B,
60                    Val_C => C);
61      end ReadValue;
62   end Switch;
```

Circular dependence and elaboration order

This example demonstrates how the Examiner locates and disallows circular dependence and elaboration relations.

Specification of package P_05 in SPARK 2005:

```
1  --# inherit Q_05;
2  package P_05
3  --# own P_State;
4  --# initializes P_State;
5  is
6     procedure Init(S : out Integer);
7  end P_05;
```

Specification of package Q_05 in SPARK 2005:

```
1  --# inherit P_05;
2  package Q_05
3  --# own Q_State;
4  --# initializes Q_State;
5  is
6     procedure Init(S : out Integer);
7  end Q_05;
```

Body of package P_05 in SPARK 2005:

```
1  with Q_05;
2  package body P_05
3  is
4     P_State : Integer;
5
6     procedure Init(S : out Integer)
7     is
8     begin
9        S := 5;
10    end Init;
11 begin
12    Q_05.Init(P_State);
13 end P_05;
```

Body of package Q_05 in SPARK 2005:

```
1  with P_05;
2  package body Q_05
```

```
3  is
4     Q_State : Integer;
5
6     procedure Init(S : out Integer)
7     is
8     begin
9        S := 10;
10    end Init;
11 begin
12    P_05.Init(Q_State);
13 end Q_05;
```

Specification of package P_14 in SPARK 2014:

```
1  package P_14
2     with Abstract_State => P_State,
3          Initializes    => P_State
4  is
5     procedure Init(S : out Integer);
6  end P_14;
```

Specification of package Q_14 in SPARK 2014:

```
1  package Q_14
2     with Abstract_State => Q_State,
3          Initializes    => Q_State
4  is
5     procedure Init(S : out Integer);
6  end Q_14;
```

Body of package P_14 in SPARK 2014:

```
1  with Q_14;
2  package body P_14
3     with Refined_State => (P_State => P_S)
4  is
5     P_S : Integer;
6
7     procedure Init(S : out Integer)
8     is
9     begin
10       S := 5;
11    end Init;
12 begin
13    Q_14.Init(P_S);
14 end P_14;
```

Body of package Q_14 in SPARK 2014:

```
1  with P_14;
2  package body Q_14
3     with Refined_State => (Q_State => Q_S)
4  is
5     Q_S : Integer;
6
7     procedure Init(S : out Integer)
8     is
9     begin
10       S := 10;
```

```
11      end Init;
12   begin
13      P_14.Init(Q_S);
14   end Q_14;
```

A.3 Bodies and Proof

A.3.1 Assert, Assume, Check contracts

Assert (in loop) contract

The following example demonstrates how the *assert* annotation can be used inside a loop. At each run of the loop the list of existing hypotheses is cleared and the statements that are within the *assert* annotation are added as the new hypotheses. The SPARK 2014 equivalent of *assert*, while within a loop, is *pragma Loop_Invariant*.

Specification in SPARK 2005:

```
1  package Assert_Loop_05
2  is
3     subtype Index is Integer range 1 .. 10;
4     type A_Type is Array (Index) of Integer;
5
6     function Value_present (A: A_Type; X : Integer) return Boolean;
7     --# return for some M in Index => (A (M) = X);
8  end Assert_Loop_05;
```

Body in SPARK 2005:

```
1  package body Assert_Loop_05
2  is
3     function Value_Present (A: A_Type; X : Integer) return Boolean
4     is
5        I : Index := Index'First;
6     begin
7        while A (I) /= X and I < Index'Last loop
8           --# assert I < Index'Last and
9           --#        (for all M in Index range Index'First .. I => (A (M) /= X));
10          I := I + 1;
11       end loop;
12       return A (I) = X;
13    end Value_Present;
14 end Assert_Loop_05;
```

Specification in SPARK 2014:

```
1  package Assert_Loop_14
2  is
3     subtype Index is Integer range 1 .. 10;
4     type A_Type is Array (Index) of Integer;
5
6     function Value_present (A: A_Type; X : Integer) return Boolean
7        with Post => (for some M in Index => (A(M) = X));
8  end Assert_Loop_14;
```

Body in SPARK 2014:

```
package body Assert_Loop_14
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_Invariant (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_14;
```

Assert (no loop) contract

While not in a loop, the SPARK 2005 *assert* annotation maps to *pragma Assert_And_Cut* in SPARK 2014. These statements clear the list of hypotheses and add the statements that are within them as the new hypotheses.

Assume contract

The following example illustrates use of an Assume annotation (in this case, the Assume annotation is effectively being used to implement the 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;

end Input_Port;
```

Body for Assume annotation in SPARK 2005:

```
package body Input_Port
is

   Inputs : Integer;
   for Inputs'Address use 16#CAFE#;
   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 Abstract_State => (State_Inputs with External, Input_Only)
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:

```
package body Input_Port
   with Refined_State => (State_Inputs => Inputs)
is
   Inputs : Integer;
   for Inputs'Address use 16#CAFE#;
   pragma Volatile (Inputs);

   procedure Read_From_Port(Input_Value : out Integer)
      with Refined_Global  => (Input => Inputs),
           Refined_Depends => (Input_Value => Inputs)
   is
   begin
      pragma Assume(Inputs in Integer);
      Input_Value := Inputs;
   end Read_From_Port;
end Input_Port;
```

Check contract

The SPARK 2005 *check* annotation is replaced by *pragma assert* in SPARK 2014. This annotation adds a new hypothesis to the list of existing hypotheses. The code is not presented but can be found under "code\check_contract".

A.3.2 Assert used to control path explosion

This example will be added in future, based on the Tutorial 5, Exercise 1 example from the advanced SPARK course.

A.4 Other Contracts and Annotations

A.4.1 Declare annotation

Todo

The declare annotation SPARK is used to control the generation of proof rules for composite objects. It is not clear that this will be required in SPARK 2014, so this section will be updated or removed in future. To be completed in the Milestone 4 version of this document.

A.4.2 Always_Valid assertion

See section Input driver using 'Append and 'Tail contracts for use of an assertion involving the Always_Valid attribute.

A.4.3 Rule declaration annotation

See section Proof types and proof functions.

A.4.4 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.

Todo

Note that the SPARK 2014 *version of the rule declaration annotation has not yet been defined [M520-006] - note that it might not even be needed, though this is to be determined - and so there is no equivalent included in the* SPARK 2014 *code.* To be completed in the Milestone 4 version of this document.

Specification in SPARK 2005:

```
1  package Stack
2  --# own State : Abstract_Stack;
3  is
4
5     --  Proof functions to indicate whether or not the Stack is empty
6     --  and whether or not it is full.
7     --# type Abstract_Stack is abstract;
8     --# function Is_Empty(Input : Abstract_Stack) return Boolean;
9     --# function Is_Full(Input : Abstract_Stack) return Boolean;
10
11    --  Proof function to give the number of elements on the stack.
12    --# function Count(Input : Abstract_Stack) return Natural;
13
14    --  Post-condition indicates that the stack will be
15    --  non-empty after pushing an item on to it, while the pre-condition
16    --  requires it is not full when we push a value onto it.
17    procedure Push(X : in Integer);
18    --# global in out State;
19    --# pre  not Is_Full(State);
20    --# post not Is_Empty(State);
21
22    --  Procedure that swaps the first two elements in a stack.
23    procedure Swap2;
24    --# global in out State;
25    --# pre  Count(State) >= 2;
26    --# post Count(State) =  Count(State~);
27
28    --  Initializes the Stack.
29    procedure Initialize;
30    --# global out State;
31    --# post Is_Empty (State);
32
33    --  Other operations not included as not needed for
34    --  this example.
35
36 private
37    Stack_Size : constant := 100;
```

```
38    type    Pointer_Range is range 0 .. Stack_Size;
39    subtype Index_Range   is Pointer_Range range 1..Stack_Size;
40    type    Vector        is array(Index_Range) of Integer;
41
42    type Stack_Type is
43       record
44          S : Vector;
45          Pointer : Pointer_Range;
46       end record;
47
48    Initial_Stack : constant Stack_Type :=
49       Stack_Type'(S       => Vector'(others => 0),
50                   Pointer => 0);
51
52 end Stack;
```

Body in SPARK 2005:

```
1  package body Stack
2  --# own State is My_Stack;
3  is
4     My_Stack : Stack_Type;
5
6     procedure Push(X : in Integer)
7     --# global in out My_Stack;
8     --# pre My_Stack.Pointer < Pointer_Range'Last;
9     --# post My_Stack.Pointer /= 0;
10    is
11    begin
12       My_Stack.Pointer := My_Stack.Pointer + 1;
13       My_Stack.S(My_Stack.Pointer) := X;
14    end Push;
15
16    procedure Swap2
17    --# global in out My_Stack;
18    --# pre  My_Stack.Pointer >= 2;
19    --# post My_Stack.Pointer = My_Stack~.Pointer;
20    is
21       Temp : Integer;
22    begin
23       Temp := My_Stack.S (1);
24       My_Stack.S (1) := My_Stack.S (2);
25       My_Stack.S (2) := Temp;
26    end Swap2;
27
28    procedure Initialize
29    --# global out My_Stack;
30    --# post My_Stack.Pointer = 0;
31    is
32       --# for Initial_Stack declare Rule;
33    begin
34       My_Stack := Initial_Stack;
35    end Initialize;
36
37 end Stack;
```

Specification in SPARK 2014

```
package Stack
   with Abstract_State => State
is
   -- We have to turn the proof functions into actual functions
   function Is_Empty return Boolean
      with Global     => (Input => State),
           Convention => Ghost;

   function Is_Full return Boolean
      with Global     => (Input => State),
           Convention => Ghost;

   function Count return Natural
      with Global     => (Input => State),
           Convention => Ghost;

   --  Post-condition indicates that the stack will be
   --  non-empty after pushing an item on to it, while the pre-condition
   --  requires it is not full when we push a value onto it.
   procedure Push(X : in Integer)
      with Global => (In_Out => State),
           Pre    => not Is_Full,
           Post   => not Is_Empty;

   --  Procedure that swaps the first two elements in a stack.
   procedure Swap2
      with Global => (In_Out => State),
           Pre    => Count >= 2,
           Post   => Count = Count'Old;

   --  Initializes the Stack.
   procedure Initialize
      with Global => (Output => State),
           Post   => Is_Empty;

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_Type is
      record
         S : Vector;
         Pointer : Pointer_Range;
      end record;

   Initial_Stack : constant Stack_Type :=
      Stack_Type'(S       => Vector'(others => 0),
                  Pointer => 0);
end Stack;
```

Body in SPARK 2014:

```
package body Stack
   with Refined_State => (State => My_Stack)
is
   My_Stack : Stack_Type;

```

```
6      function Is_Empty return Boolean
7         with Refined_Global => (Input => My_Stack),
8              Refined_Post   => Is_Empty'Result = (My_Stack.Pointer = 0)
9      is
10     begin
11        return My_Stack.Pointer = 0;
12     end Is_Empty;
13
14     function Is_Full return Boolean
15        with Refined_Global => (Input => My_Stack),
16             Refined_Post   => Is_Full'Result = (My_Stack.Pointer = Pointer_Range'Last)
17     is
18     begin
19        return My_Stack.Pointer = Pointer_Range'Last;
20     end Is_Full;
21
22     function Count return Natural
23        with Refined_Global => (Input => My_Stack),
24             Refined_Post   => Count'Result = My_Stack.Pointer
25     is
26     begin
27        return My_Stack.Pointer;
28     end Count;
29
30     procedure Push(X : in Integer)
31        with Refined_Global => (In_Out => My_Stack),
32             Refined_Pre    => My_Stack.Pointer /= Pointer_Range'Last,
33             Refined_Post   => My_Stack.Pointer /= 0
34     is
35     begin
36        My_Stack.Pointer := My_Stack.Pointer + 1;
37        My_Stack.S(My_Stack.Pointer) := X;
38     end Push;
39
40     procedure Swap2
41        with Refined_Global => (In_Out => My_Stack),
42             Refined_Pre    => My_Stack.Pointer >= 2,
43             Refined_Post   => My_Stack.Pointer = My_Stack'Old.Pointer
44     is
45        Temp : Integer;
46     begin
47        Temp := My_Stack.S (1);
48        My_Stack.S (1) := My_Stack.S (2);
49        My_Stack.S (2) := Temp;
50     end Swap2;
51
52     procedure Initialize
53        with Refined_Global => (Output => My_Stack),
54             Refined_Post   => My_Stack.Pointer = 0
55     is
56     begin
57        My_Stack := Initial_Stack;
58     end Initialize;
59  end Stack;
```

A.4.5 Main_Program annotation

See the main program annotation used in section Basic Input and Output Device Drivers.

A.4.6 RavenSPARK patterns

The Ravenscar profile for tasking is not yet supported in SPARK 2014. Mapping examples will be added here in future.

APPENDIX

B

RESTRICTIONS AND PROFILES

A list of restrictions by section and their effect:

2.1 Character Set

1. `No_Wide_Characters`

   This GNAT-defined restriction may be applied to restrict the use of Wide and Wide_Wide character and string types in SPARK 2014.

6.1 Subprogram Declarations

1. `No_Default_Subprogram_Parameters`

   Prohibits the use of default subprogram parameters, that is, a `parameter_specification` cannot have a `default_expression`.

Todo

access and aliased parameter specs, null exclusion parameters. Function access results function null exclusion results. To be completed in the Milestone 4 version of this document.

Note: RCC. Should we forbid these thing outrights, or just ignore them and/or mark the corresponding declarations as "not SPARK"? Assign: ???

6.1.4 Mode Refinement

1. `Moded_Variables_Are_Entire`

   Asserts that a `moded_item` cannot be a subcomponent name.

2. `No_Conditional_Modes`

   Prohibits the use of a `conditional_mode` in a `mode_specification`.

3. `No_Default_Modes_On_Procedures`

   A style restriction that disallows a `default_mode_specification` within a procedure `mode_refinement`. An explicit `Input =>` must be given. A function `mode_refinement` may have a `default_mode_specification`.

6.1.5 Global Aspects

Todo

In the following restriction, is this the assumption of no Global aspect implies Global => null sensible or should we always insist on Global => null?? I hope not!! RCC comment: see discussion under LA11-017 started by RCC on 26/10. To be completed in the Milestone 4 version of this document.

---

1. `Global_Aspects_Required`

   Enforces the use of a `global_aspect` on every subprogram which accesses a *global* variable. When this restriction is in force a subprogram which does not have an explicit `global_aspect` is considered to have a have have one of `Global =>` **null**.

2. `Global_Aspects_On_Procedure_Declarations`

   A less stringent restriction which requires a `global_aspect` on all procedure declarations that access a *global* variable. A `global_aspect` is optional on a subprogram body that does not have a separate declaration. An implicit global aspect is calculated from the body of each subprogram body which does not have an explicit `global_aspect`.

Note: RCC. I have changed "virtual" to "implicit" here since the latter is used to mean the same thing later on and seems more consistent.

6.1.7 Dependency Aspects

1. `Procedures_Require_Dependency_Aspects`

   Mandates that all procedures must have a `dependency_aspect`. Functions may have a `dependency_aspect` but they are not required.

2. `Procedure_Declarations_Require_Dependency_Aspects`

   A less stringent restriction which only requires a `dependency_aspect` to be applied to a procedure declaration. A `dependency_aspect` is optional on a subprogram body that does not have a separate declaration. An implicit dependency aspect is calculated from the body of each subprogram body which does not have an explicit `dependency_aspect`.

3. `No_Conditional_Dependencies`

   Prohibits the use of a `conditional_dependency` in any `dependency_relation`.

4. `Dependencies_Are_Entire`

   Prohibits the use of subcomponents in `dependency_relations`.

6.2 Formal Parameter Modes

1. `Strict_Modes`

   - A *formal parameter* (see Ada RM 6.1) of a subprogram of mode **in** or **in out** (an `import`) must be read on at least one execution path through the body of the subprogram and its initial value used in determining the value of at least one of `export` or the special **null** export symbol.
   - A *formal parameter* of a subprogram of mode **in out** must be updated directly or indirectly on at least one executable path within the subprogram body.
   - A *formal parameter* of a subprogram of mode **out** must be updated directly or indirectly on every executable path through the subprogram body.

   This restriction has to be checked by flow analysis.

6.3 Subprogram Bodies

1. `End_Designators_Required`

Mandates that the final end of every subprogram body, package declaration and package body has a designator which repeats the defining designator of the unit.

Note: RCC. Is End_Designators_Required really ever going to be used? It was only required in S95 to facilitate the implementation of the hide anno really. This feels more like a rule for GNATCheck that users might choose to employ, but I don't think it makes any difference to verifiability, so no business of SPARK 2014?

6.3.2 Global Aspects

1. `No_Scope_Holes`

   A subprogram, P, shall not declare an entity of the same name as a `moded_item` or the name of the object of which the `moded_item` is a subcomponent in its `global_aspect` within a `loop_statement` or `block_statement` whose nearest enclosing program unit is P.

Note: RCC. Is No_Scope_Holes really necessary for proof or any other form of verification?

6.4.2 Anti-Aliasing

1. `Array_Elements_Assumed_To_Overlap`

   Enforces the assumption that array elements are always considered to be overlapping and so, for example, V.A(I).P and V.A(J).Q are considered as overlapping. This restriction can be enforced simply whereas the more general rule that array subcomponents are only considered to be overlapping when they have common indices requires formal proof in general.

Note: RCC. Strongly agree that we need this for rel1, since it gets us back to the simple aliasing rules of S95, without having to resort to proof.

7.1 Packages

1. `End_Designators_Required`

   See the same restriction in section 6.3.

2. `Package_Aspects_Required`

   Enforces the restrictions `Abstract_State_Aspects_Required`, `Initializes_Aspects_Required` and `Refined_State_Aspects_Required`.

7.1.2 Abstract State Aspect

1. `Abstract_State_Aspects_Required`

   Applies to an entire package including any embedded packages and its private child packages and enforces the restriction that a package which has hidden state must have an `abstract_state_aspect`. If this restriction is in force the absence of an `abstract_state_name` implies Abstract_State => **null**.

7.1.3 Initializes Aspect

1. `Initializes_Aspects_Required`

   If any of the state components of a package, including *variables* declared in its visible part are initialized during the elaboration of the package, then the initializes state components must appear in an `initializes_aspect`. If this restriction is in force the absence of an `initializes_aspect` implies Initializes => **null**.

2. `Package_Elaboration_Initializes_Local_State_Only`

Applies to an entire package including any embedded packages and its private child packages and enforces the restriction that the package may only initialize state declared locally to the package during its elaboration. That is, only the *variables* declared immediately within the package.

3. `Package_Elaboration_Initializes_Local_And_Parent_State_Only`

   A package may only initialize a *variable* declared *locally* to the package, a visible *variable* of its parent or indirectly a `state_name` of its parent.

4. `Package_Elaboration_Order_Independence`

   Enforces the rule that elaboration of a package Q may only initialize a *variable* using a *static expression* or using subprograms and *variables local* to Q. Ultimately all the initialization values must be derived from *static expressions*. If this restriction is in force then the predicate of an `initial_condition_aspect` of a package may only refer to state initialized by Q.

7.1.4 Initial Condition Aspect

1. `Initialize_Package_Local_State_Only`

   See the same restriction in section 7.1.3.

2. `Package_Elaboration_Order_Independence`

   See the same restriction in section 7.1.3.

7.2.2 Refined State Aspect

1. `Refined_State_Aspects_Required`

   If a package has an `abstract_state_aspect` then a corresponding ‘‘ refined_state_aspect‘‘ is required.

2. `Null_State_Refinement_Prohibited`

   The `abstract_state_name` **null** cannot be used in a `state_refinement_aspect`.

3. `Strict_Volatile_State_Refinement`

   A `constituent` of a Volatile `abstract_state_name` must be Volatile and be of the same mode.

END OF FILE

APPENDIX

C

TO-DO SUMMARY

Todo

Lift restriction that non-preelaborable subtypes are not subject to flow analysis. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\declarations-and-types.rst, line 50.)

Todo

Add the Dynamic_Predicate aspect to SPARK 2014. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\declarations-and-types.rst, line 74.)

Todo

Lift restriction that non-preelaborable constants are not subject to flow analysis. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\declarations-and-types.rst, line 99.)

Todo

Add ‘Class attribute to SPARK 2014. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\declarations-and-types.rst, line 156.)

Todo

Include interface types in SPARK 2014. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\declarations-and-types.rst, line 184.)

Todo

Update SPARK 2014 to allow prove once/use many approach to generics. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\generic-units.rst, line 53.)

Todo

We need to increase the number of examples given. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 156.)

Todo

Consider adding a glossary, defining terms such as flow analysis and formal verification. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 193.)

Todo

The pragmas equivalent to the new aspects need to be added to this document. To be added in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 268.)

Todo

Complete detail on mixing SPARK 2014 with non-Ada code. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 355.)

Todo

Ensure that all strategic requirements have been implemented. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 403.)

Todo

Where Ada 2012 language features are designated as not in SPARK 2014 in subsequent chapters of this document, add tracing back to the strategic requirement that motivates that designation. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 407.)

Todo

Add detail on restrictions to be applied to tested code, making clear that the burden is on the user to get this right, and not getting it right can invalidate the assumptions on which proof is based. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 531.)

Todo

Complete detail on combining formal verification and testing. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 536.)

Todo

Complete detail on Code Policies. To be completed in the Milestone 4 version of this document. Consider referencing the User's Guide for details of the various profiles.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 576.)

Todo

We need to consider what might need to be levied on the non-SPARK 2014 code in order for flow analysis on the SPARK 2014 code to be carried out. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 807.)

Todo

Complete detail on mixing code that is in and out of SPARK 2014. In particular, where subheadings such as Legality Rules or Static Semantics are used to classify the language rules given for new language features, any rules given to restrict the Ada subset being used need to be classified in some way (for example, as Subset Rules) and so given under a corresponding heading. In addition, the inconsistency between the headings used for statements and exceptions needs to be addressed. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\introduction.rst, line 812.)

Todo

Depending on the outcome of M423-014, either pragma Annotate or pragma Warning will be utilized to accept warnings/errors in SPARK 2014. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\mapping-spec.rst, line 94.)

Todo

There will not be an equivalent of 'Append and 'Tail in SPARK 2014. However, we will be able to achieve the same functionality using generics. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\mapping-spec.rst, line 806.)

Todo

There will not be an equivalent of 'Append and 'Tail in SPARK 2014. However, we will be able to achieve the same functionality using generics. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\mapping-spec.rst, line 831.)

Todo

There will not be an equivalent of 'Append and 'Tail in SPARK 2014. However, we will be able to achieve the same functionality using generics. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\mapping-spec.rst, line 954.)

Todo

The declare annotation SPARK is used to control the generation of proof rules for composite objects. It is not clear that this will be required in SPARK 2014, so this section will be updated or removed in future. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\mapping-spec.rst, line 1223.)

Todo

Note that the SPARK 2014 *version of the rule declaration annotation has not yet been defined [M520-006] - note that it might not even be needed, though this is to be determined - and so there is no equivalent included in the* SPARK 2014 *code.* To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\mapping-spec.rst, line 1250.)

Todo

Are there any other language defined attributes which will not be supported? To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\names-and-expressions.rst, line 45.)

Todo

What do we do about Gnat defined attributes, a useful one is: For a prefix X that denotes an object, the GNAT-defined attribute `X'Valid_Scalars` is defined in SPARK 2014. This Boolean-valued attribute is equal to the conjunction of the `Valid` attributes of all of the scalar parts of X.

[If X has no volatile parts, `X'Valid_Scalars` implies that each scalar subcomponent of X has a value belonging to its subtype. Unlike the Ada-defined `Valid` attribute, the `Valid_Scalars` attribute is defined for all objects, not just scalar objects.]

Perhaps we should list which ones are supported in an appendix? Or should they be part of the main language definition?

It would be possible to use such attributes in assertion expressions but not generally in Ada code in a non-Gnat compiler.

To be completed in the Milestone 4 version of this document. Note that as language-defined attributes form Appendix K of the Ada RM, any GNAT-defined attributes supported in SPARK 2014 will be presented in an appendix.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\names-and-expressions.rst, line 48.)

Todo

Detail on Update Expressions needs to be put into the standard format. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\names-and-expressions.rst, line 103.)

Todo

Add support for more complex models of external state. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\packages.rst, line 256.)

Todo

refined contract_cases. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\packages.rst, line 1767.)

Todo

Add support for type invariants in SPARK 2014. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\packages.rst, line 1973.)

Todo

Provide full detail on Representation Issues. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\representation-issues.rst, line 6.)

Todo

This statement was originally in this chapter "Pragma or aspect `Unchecked_Union` is not in SPARK 2014" this needs to be recorded in the list of unsupported aspects and pragmas. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\representation-issues.rst, line 9.)

Todo

Provide a detailed semantics for Refined_Pre and Refined_Post aspects on Unchecked_Conversion. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\representation-issues.rst, line 60.)

Todo

Need to put some words in here to describe the precautions that may be taken to avoid invalid data. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\representation-issues.rst, line 72.)

Todo

Introduce checks for data validity into the proof model as necessary. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\representation-issues.rst, line 76.)

Todo

access and aliased parameter specs, null exclusion parameters. Function access results function null exclusion results. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\restrictions-and-profiles.rst, line 25.)

Todo

In the following restriction, is this the assumption of no Global aspect implies Global => null sensible or should we always insist on Global => null?? I hope not!! RCC comment: see discussion under LA11-017 started by RCC on 26/10. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\restrictions-and-profiles.rst, line 54.)

Todo

Need to consider further the support for iterators and whether the application of constant iterators could be supported.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\statements.rst, line 54.)

Todo

(TJJ 29/11/12) Do we need this verification rule? Could it be captured as part of the general statement about proof? To be completed in milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\subprograms.rst, line 225.)

Todo

Add ghost types and ghost variables to SPARK 2014. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\subprograms.rst, line 832.)

Todo

Make worst-case assumptions about private types for this rule, or blast through privacy? To be completed in milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\subprograms.rst, line 879.)

Todo

Can a ghost variable be a constituent of a non-ghost state abstraction, or would this somehow allow unwanted dependencies? If not, then we presumably need to allow ghost state abstractions or else it would be illegal for a library level package body to declare a ghost variable. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\subprograms.rst, line 908.)

Todo

Do we want an implicit Ghost convention for an entity declared within a statement whose execution depends on a ghost value? To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\subprograms.rst, line 916.)

Todo

The modes of a subprogram in Ada are not as strict as S2005 and there is a difference in interpretation of the modes as viewed by flow analysis. For instance in Ada a formal parameter of mode out of a composite type need only be partially updated, but in flow analysis this would have mode in out. Similarly an Ada formal parameter may have mode in out but not be an input. In flow analysis it would be regarded as an input and give rise to flow errors.

In deciding whether a parameter is only partially updated, discriminants (including discriminants of subcomponents) are ignored. For example, given an *out* mode parameter of a type with defaulted discriminants, a subprogram might or might not modify those discriminants (if it does, there will of course be an associated proof obligation to show that the parameter's 'Constrained attribute is False in that path).

Perhaps we need an aspect to describe the strict view of a parameter if it is different from the specified Ada mode of the formal parameter? To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\subprograms.rst, line 984.)

Todo

RCC: The above text implies that SPARK 2014 does not support Ada.Calendar, which is specified in RM 9.6. SPARK 2005 supports and prefers Ada.Real_Time and models the passage of time as an external "in" mode protected own variable. Should we use the same approach in SPARK 2014? Discussion under TN [LB07-024]. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\tasks-and-synchronization.rst, line 10.)

Todo

Add Tasking. To be completed in a post-Release 1 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\tasks-and-synchronization.rst, line 16.)

Todo

Provide detail on Standard Libraries. To be completed in a post-Release 1 version of this document. This targeting applies to all ToDos in this chapter.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 11.)

Todo

In particular, it is intended that predefined container generics suitable for use in SPARK 2014 will be provided. These will have specifications as similar as possible to those of Ada's bounded containers (i.e., Ada.Containers.Bounded_*), but with constructs removed or modified as needed in order to maintain the language invariants that SPARK 2014 relies upon in providing formal program verification.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 15.)

Todo

Should we say here which packages are supported in SPARK 2014 or which ones aren't supported? How much of the standard library will be available, and in which run-time profiles?

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 36.)

Todo

How much to say here? S95 supports a subset of Interfaces, and a very small subset of Interfaces.C but that's about it.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 47.)

Todo

What is status of supported for pragma `Unchecked_Union` in GNATProve at present?

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 50.)

Todo

RCC: Need to think about Ada.Real_Time. It's important for all S95 customers, to get at monotonic clock, even if not using RavenSPARK. It does depend on support for external variables, though, since Ada.Real_Time.Clock is most definitely Volatile. TN [LB07-024] raised to discuss this.

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 63.)

Todo

How much here can be supported? Most S95 customers want Ada.Numerics.Generic_Elementary_Functions plus its predefined instantiation for Float, Long_Float and so on. How far should we go?

(The *original entry* is located in C:\sparkdev\spark2014_shared\docs\lrm\source\the-standard-library.rst, line 84.)

APPENDIX

D

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

D.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.

D.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.

D.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.

D.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.

D.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.

D.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'.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.

D.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.