SPARK 2014 LRM

Release 0.2

Altran and AdaCore

February 25, 2013

CONTENTS

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.

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 compiled by and executed by any Ada 2012 implementation.

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 *Requirements Given in this Document* provides an overview of the requirements presented in this document over and above the language definition rules of the sort that appear in the Ada 2012 Reference Manual (RM).
- 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 *Generic Language-Independent Requirements* presents language-independent requirements that are common to a number of the language features described in this document.
- 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 2 – 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).

Readers interested in how SPARK 2005 constructs and idioms map into SPARK 2014 should consult the appendix *SPARK 2005 to SPARK 2014 Mapping Specification*. Note that this section does not cover all language features presented in this document – although it covers the main features – and will be updated for the Milestone 3 version of this document.

Todo

Update mapping specification section to cover all necessary language features. To be completed in the milestone 3 version of this document.

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 the SPARK 2014 aspects and pragmas.
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 the proof and flow analysis checks to be performed on the language feature.

All sections are always listed and if no content is required then the corresponding section will be marked *Not applicable*. 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 3 version of this document.

1.5 Formal Analysis

SPARK 2014 will be amenable to a range of formal analyses, including but not limited to:

- Data-flow analysis, which considers the initialization of variables and the direction of data flow into and out of subprograms.
- Information-flow analysis, which also considers the coupling between the inputs and outputs of a subprogram. 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 may not be possible if the legality rules of Ada 2012 and those presented in this document are not met. Similarly, a formal verification may 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.

1.5.1 Further Detail 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)).

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 Statements*).

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

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 dynamic semantics of the two languages are defined to be identical, so that a valid SPARK 2014 program may be compiled and executed by means of an Ada compiler.

SPARK 2014 programs that have failed their static analysis checks can still be valid Ada 2012 programs. An incorrect SPARK 2014 program with, say, inconsistent dataflow annotations 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 checking of dataflow dependencies.

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 2012 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.

1.7 Requirements Given in this Document

1.7.1 Detailed SPARK 2014 Language Definition

The detailed SPARK 2014 language definition is given in Ada terminology and has two main components. The first defines extensions to Ada 2012 in terms of new aspects, pragmas and attributes to provide SPARK 2014 features such as global specifications for subprograms. The second defines a subset of Ada 2012 by excluding certain language features. The exclusions, the new aspects, pragmas, attributes and rules specify the largest SPARK 2014 language for which formal analyses are supported.

Code Policies may be applied which effectively reduce further the language subset which may be analyzed but may make analysis and proof easier, more precise and be suitable for some application areas (see *Code Policies*).

1.7.2 Higher-Level Requirements

Higher-level requirements are given in non Ada specific terminology and have the following structure:

1. Strategic requirements to be met by the SPARK 2014 language and its associated toolset (given in this chapter).
2. Requirements to provide particular language features.
3. For each such language feature, requirements are given to define how that feature should work in a way that is - as much as possible - language independent. [This means that language features may be understood independently of the low-level details needed to make them work.]

Where relevant, a rationale will be given to explain why the requirement is levied. Further narrative detail is given on each of the strategic requirements.

Since this detail does not strictly belong in this document in future it will be extracted and included in a new requirements document.

1.7.3 Presentation of Language Feature Requirements

For each language feature, higher-level requirements are given under the following headings:

1. *Goals to be met by language feature*: this defines the broad need behind a given language feature, along with requirements on the capabilities that the feature needs to support.
2. *Constraints*: this defines any ways in which we need to restrict the nature of the language feature, typically to serve the needs of analysis or verification.
3. *Consistency*: here, we consider the other language features being implemented and consider what the relationship should be between this and those features.
4. *Semantics*: here we define what the language feature means and hence what it means for the program to be correct against any specification given using this feature.

1.7.4 Reading these Requirements

The higher-level requirements are naturally given in language that is less precise than would be expected of rules in a language reference manual. Where greater precision is required, this will be given in the language definition rules themselves.

1.7.5 Generic Requirements

A number of requirements apply to multiple language features and they are given at the end of this chapter in section *Generic Language-Independent Requirements*.

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.

- 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.
- SPARK 2014 shall provide support for constructive, generative and retrospective analysis as follows (see section *Constructive, Generative and Retrospective Analysis* for further details):
  - SPARK 2014 shall support constructive (modular) specification, analysis and verification of (partially) developed programs, to allow static analysis as early as possible in the development lifecycle. [Hence, package and subprogram bodies need not be present for formal verification to proceed.]
  - SPARK 2014 shall complete by generation from the body code, where possible, incomplete contracts. For instance, if a dependency relation is given on a subprogram but a subprogram nested within does not have a dependency relation, it should be generated by the tools. This may shorten development time and should simplify maintenance.
  - SPARK 2014 shall support retrospective analysis where useful forms of verification can be achieved with code that complies with the core SPARK 2014 restrictions, but otherwise does not have any contracts. Implicit contracts can be computed from the bodies of units, and then used in the analysis of other units, and so on. Parts of the program which are not compliant with SPARK 2014 subset cannot be fully verified by the tools but may be represented by a SPARK 2014 compatible contract at the unit level.
- *Code Policies* shall be allowed that reduce the subset of Ada 2012 that may be used in line with specific goals such as domain needs or certification requirements (these are similar to *Profiles* but may be imposed at a finer granularity and the effect of a breach may also be different). This 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.
- SPARK 2014 shall support the development, analysis and verification of programs which are only partly within the SPARK 2014 language, with other parts in another language, for instance, full Ada or C. SPARK 2014 compatible contracts at unit level will form the boundary interface between the SPARK 2014 and other parts of the program. Many systems are not written in a single programming language and when retrospectively analyzing pre-existing code it may well not all conform to the SPARK 2014 subset. *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 Entities*.
- 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.
- Support for specifying and verifying properties of secure systems shall be improved over what is available in SPARK 2005.
- SPARK 2014 shall support the analysis of volatile variables, typically external inputs or outputs. See section *Volatile State* for further details.
- 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 property is sound. Sound analysis means that the method never asserts a property to be true when it is not true." Language features that defy sound analysis

will be eliminated or their 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.*

- 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 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.*

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.

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.
- The goto statement is not permitted.
- The use of controlled types is not 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 permitted.

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 is 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.

Since it should be possible to apply these policies at multiple levels of granularity - for example at a package level rather than at a library level - and since it need not be the case that violation of one of these policies leads to a compilation error, then the existing Ada mechanisms of pragma Restriction and pragma Profile are not suitable. Hence, pragma Code_Policy will be introduced as a counterpart to pragma Profile and pragma Guideline will be introduced as a counterpart to pragma Restriction, meaning that a Code_Policy is a grouping of Guidelines.

It is intended that code policies can be customised or new policies specified from a collection of guidelines.

No further detail is given in the current draft of this document on Code Policies.

Todo

Complete detail on Code Policies. To be completed in the Milestone 3 version of this document.

1.9.4 Ghost Entities

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. In principle such entities could be completely removed from the program without any functional impact.

SPARK 2014 supports ghost functions which 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 have to be defined within an external proof tool to facilitate formal verification.

Any function, ghost or otherwise, may have its specification defined within an external proof tool for formal verification purposes. The specification is outside of the SPARK 2014 language and toolset and therefore cannot be checked by these. 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 a function can be specified in SPARK 2014, then its specification can be recast as the expression of an expression function without further implementation. This may not give the most efficient implementation but if the function is a ghost function it may be sufficient.

Further Ghost entities are to be added in future drafts of this document.

Todo

Complete detail on Ghost Entities. To be completed in the Milestone 3 version of this document.

1.9.5 Constructive, Generative and Retrospective Analysis

SPARK 2005 strongly favored the *constructive* analysis style where all program units required contracts to be provided on their specifications. The contracts are needed to perform in-depth static analysis and formal verification. These contracts had to be designed and added at an early stage to assist modular analysis and verification, and then maintained by the user as a program evolved. When the body is implemented (or modified) it is checked that it conforms to its contract.

However, some of these contracts – if they are not explicitly provided – can be implicitly synthesized for a subprogram from its body (provided the contracts of any subprograms it calls are specified or have already been generated). The contracts can then be used in the analysis of calling subprograms and so on. In SPARK 2014 the contracts which may be synthesized from an implemented subprogram body are the global specification and the dependency relation. It may be possible to generate some of the package contracts also once the package body and its private dependents have been implemented.

Unlike the Global and Depends aspects used in flow analysis, the SPARK 2014 tools will not attempt to automatically synthesize for a given subprogram body the other aspects (i.e. Pre and Post), which define the subprogram's contract for the purpose of formal verification.

There are three main use cases where generation of contracts are required:

- Code has been developed as SPARK 2014 but in order to reduce costs not all the contracts are included on all subprograms by the developer.
- Code is in maintenance phase, it may or may not have complete contracts. If the contracts are complete, the generated contracts may be compared with the given contracts and auto correction used to update the contracts if the changes are acceptable. If the contracts are incomplete they are automatically generated for analysis purposes.
- Legacy code is analyzed which has no or incomplete contracts.

Hence, as well as still fully supporting the constructive development mode, SPARK 2014 is designed to facilitate the generation of contracts, which supports retrospective analysis.

Note that in the case where legacy code is being analyzed there may be 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 may be necessary:

- An automatic identification of what code is in SPARK 2014 and what is not.

- An annotation of the boundary between the SPARK 2014 and non-SPARK 2014 code with suitable SPARK 2014 compatible contracts. If this is not done then the analysis would have to assume some suitably conservative contract.

Note that when language features are presented and defined in the remainder of this document, it is assumed that analysis and verification is being performed constructively and no explicit detail is given on generative or retrospective analysis.

No further detail is given in the current draft of this document on Constructive, Generative and Retrospective analysis and Verification.

Todo

Add detail on how retrospective analysis will work when we have a mix of SPARK 2014 and non-SPARK 2014. To be completed in the Milestone 3 version of this document.

Todo

Complete detail on constructive, generative and retrospective analysis and verification. To be completed in the Milestone 3 version of this document.

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 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 construct appearing in a unit may not be in, or may depend on features not in, the SPARK 2014 subset. The construct may generate a warning or an error which may be justifiable. This does not necessarily render the whole of the unit as not in SPARK 2014. If the construct generates a warning, or if the error is justified, then the unit is considered to be in SPARK 2014 except for the errant construct.

- It is the *use* of a construct not in SPARK 2014 (generally within the statements of a body) that potentially moves the code outside of the SPARK 2014 subset. An unsuppressible error will be generated in such a case and the body containing the code 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.

No further detail is given in the current draft of this document on mixing code that is in and out of SPARK 2014. Although there are a number of places where a statement is given on what is in or out of SPARK 2014, that information is not yet complete and nothing further is given on how it should be used.

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 Volatile State

A variable or a state abstraction (see *State Abstraction, Hidden State and Refinement*) may be designated as volatile. A volatile variable or state abstraction need not have the same value between two reads without an intervening update. Similarly an update of a volatile variable (or state abstraction) may 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.

In formal verification a series of reads and updates of a volatile variable or state abstraction may be modeled by a sequence or a trace.

In both 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.

1.10 Generic Language-Independent Requirements

The following detail relates to higher-level requirements but applies to multiple language features. Hence, it is given in a single place to ease readability.

1.10.1 Definition of Terms for Higher-Level Requirements

The following terms are used in the presentation of the higher-level requirements; each is intended to have a definition consistent with that when used in language definition rules.

1. *Hidden state*: state declared within a package but that is not directly accessible by users of that package.
2. *Inputs and outputs of a subprogram*: the set of data items that may be read or written - either directly or indirectly - on a call to that subprogram.
3. *Global data of a subprogram*: the inputs and outputs of a subprogram minus the formal parameters.
4. *Entire variable*: a variable that is not a subcomponent of a larger containing variable.
5. *Entity*: the semantic object that represents a given declaration.

1.10.2 State Abstraction, Hidden State and Refinement

1. **Requirement:** When specifying properties of a subprogram, it shall be possible to refer to (an abstraction of) hidden state without knowing the details of that hidden state.

   Rationale: allows modular verification and also allows the management of complexity.

2. **Requirement:** It shall be possible to manage hierarchies of data abstraction [i.e. it shall be possible to manage a hierarchical organization of hidden state].

   Rationale: to allow modular verification and the management of complexity in the presence of programs that have a hierarchical representation of data.

1.10.3 Naming

1. **Requirement:** Variable names in a global specification of a subprogram are distinct from the formal parameter names of the subprogram .

   Rationale: A variable cannot be both a formal parameter and a global variable simultaneously.

2. **Requirement:** Names used in the new flow analysis specifications shall refer to entire variables. Within a subprogram body flow analysis will operate at an individual subcomponent level for objects of a record type.

   Rationale: For the flow analysis model at the inter-subprogram level, updating part of a variable is regarded as updating all of it. Within a subprogram body the subcomponents of a record type object are tracked individually.

3. **Requirement:** Where distinct names are referenced within a given flow analysis specification, then those names shall refer to distinct entities.

   Rationale: to support flow analysis and to aid clarity of the interface definition.

1.10.4 Properties of Specifications

1. **Requirement:** When specifying program behavior in terms of a relation or a set, it shall be possible to explicitly provide a null relation or an empty set.

   Rationale: to explicitly identify programs that - for example - do not reference global data. This is especially needed in the presence of retrospective analysis, where absence of a specification cannot mean presence of a null specification.

2. **Requirement:** It shall be possible to designate - both visible and hidden - state items that are Volatile and for each to give a mode of either in or out.

   Rationale: to model programs that refer to external state, since that state is modeled differently to internal state.

3. **Requirement:** When specifying subprogram behavior other than via proof statements – such as global data – it shall be necessary to provide a complete specification.

   Rationale: To allow provision of at least the same functionality and error detection as SPARK 2005 and to allow modular analysis. This is also necessary for security analysis.

1.11 Notes on the Current Draft

This document is a draft that covers all language-independent requirements for the main language features, provides syntax where possible and otherwise provides the detailed rules necessary to support implementation of basic flow analysis. Where detail is not relevant to meeting these needs then it has typically been removed, though a “ToDo” will indicate that there is material still to be provided.

Note this means there are certain of the strategic requirements that are currently not decomposed into language definition detail. Where this is the case, it will have been explicitly indicated in this chapter.

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*.

Certain type and subtype declarations can specify a default value to be given to declared objects of the (sub)type. There are several syntatic names and schemes for defining the default value: the `default_expression` of discriminants and record components, Default_Value aspect of scalar types and Default_Component_Value aspect for an array-of-scalar subtype. These are collectively known as *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.

3.2.1 Classification of Operations

No restrictions or extensions.

3.2.2 Subtype Predicates

Todo

It is intended to support subtype predicates. Analysis is required to determine if any subset rules need to be applied and also regarding any extra proof rules that might need to be applied.

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.

3.4 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

A scalar type declaration is in SPARK 2014 unless it has a default initialization that is not in SPARK 2014.

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_part` is in SPARK 2014 if it is not an access type and its default initialization, if any, is in SPARK 2014

3.8 Record Types

An entity declared by a `record_type_definition` is in SPARK 2014 if all of its components are in SPARK 2014 and if a component has a default initialization then all of the components must have a default initialization. A default initialization, if present must also be in SPARK 2014.

SPARK 2014 does not permit partial default initialization of record objects.

3.9 Tagged Types and Type Extensions

No extensions or restrictions currently identified, though see ToDo.

Todo

RCC comment: This will need to describe any global restrictions on tagged types (if any) and any additional Restrictions that we may feel users need. To be completed in the Milestone 4 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 3 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 3 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 type is not allowed in SPARK 2014.

4.3 Aggregates

The box symbol, <>, may only be used in an aggregate if the type of the component(s) to which it pertains has a default initialization. This restriction is covered in more detail in the following subsections.

4.3.1 Record Aggregates

Verification Rules

1. A `record_component_association` may only use the option `component_choice_list => <>` if the type denoted by each *component*`_selector_name` has a default initialization.

4.3.2 Extension Aggregates

No extensions or restrictions.

4.3.3 Array Aggregates

Verification Rules

1. The symbol <> may only be used after an **others** symbol in a `positional_array_aggregate` if the component type of the array has a default initialization.
2. The symbol <> may only be used in a `array_component_association` of a `named_array_aggregate` if the component type of the array has a default initialization.

4.3.4 Update Expressions

Todo

Detail on Update Expressions needs to be put into the standard format. To be completed in the Milestone 3 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 a value.

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

4.5 Operators and Expression Evaluation

No extensions or restrictions.

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.

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

5.5.2 Generalized Loop Iteration

SPARK 2014 does not permit the use of variable iterators.

Todo

Need to consider further the support for iterators and whether the application of constant iterators could be supported. To be completed in Milestone.4 version of this document.

5.5.3 Loop Invariants, Variants and Entry Values

High-Level Requirements

1. Goals to be met by language features:
   - **Requirement:** SPARK 2014 shall include feature/s to support proof of loop termination.

     Rationale: To aid detection of a serious programming error.
   - **Requirement:** SPARK 2014 shall include feature/s to support proof of partial correctness of code containing loops.

     Rationale: To support proof.
   - **Requirement:** Within a loop, it shall be possible to refer to the value of a given variable on entry to that loop.

     Rationale: To support proof.
2. Constraints, Consistency, Semantics, General requirements:
   - Not applicable

Language Definition

Two loop-related pragmas, `Loop_Invariant` and `Loop_Variant`, and a loop-related attribute, `Loop_Entry` are defined. The pragma `Loop_Invariant` is similar to pragma `Assert` except for its proof semantics. 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.

Loop_Invariant is just like pragma Assert with respect to syntax, name resolution, legality rules, dynamic semantics, and assertion policy, except for an legality rule given below.

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

Loop_Variant has different dynamic semantics as detailed below.

Syntax

1. Pragma Loop_Variant expects a list of parameters which are a specialization of an Ada expression as follows:

```
loop_variant_parameters ::= loop_variant_item {, loop_variant_item}
loop_variant_item       ::= change_direction => discrete_expression
change_direction        ::= Increases | Decreases
```

To be completed in the Milestone 3 version of this document.

Todo

Provide detail on pragmas Loop_Invariant and Loop_Variant, and attribute Loop_Entry. To be completed in the Milestone 3 version of this document.

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 Statements

This section discusses the pragmas Assert_And_Cut and Assume.

5.9.1 High-Level Requirements

1. Goals to be met by language feature:
   - **Requirement:** It shall be possible for users to explicitly state assumptions within the text of a subprogram to support the formal verification of that subprogram.

     Rationale: This allows facts about the domain to be used in a proof in a clean and explicit way.
   - **Requirement:** It shall be possible for users to assert at a given point within a subprogram the minimum set of facts required to complete formal verification of that subprogram.

     Rationale: This allows an explicit statement of what is necessary to complete formal verification and also assists the efficiency of that verification.
2. Constraints, Consistency, Semantics, General requirements:
   - Not applicable

5.9.2 Language Definition

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 ofpragma `Assume` can be assumed to be True for the remainder of the subprogram. 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.

Pragma Assert_And_Cut and Loop_Invariant are similar to an Assert statement except they also act 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 from the statement list before the cut point. Only the given Boolean expression is carried forward.

Assert_And_Cut, Assume and Loop_Invariant are the same as pragma Assert with respect to Syntax, Name Resolution, Legality Rules, Dynamic Semantics, and assertion policy. Apart from the legality rule that restricts the use of Loop_Invariant to a loop (see *Loop Invariants, Variants and Entry Values*).

Note: (TJJ 21-Feb-2013) Loop_Invariant is partially covered in two separate sections when we re-instate and complete the loop invariant, variant, loop entry value text we should rationalize the placement of the description of loop invariant to one section.

Verification Rules

1. Pragma Assert_And_Cut and Loop_Invariant have similar rules to pragma Assert and follow from the usual rule that any runtime check [in this case, the check is that the evaluation of the assertion expression yields True] introduces a corresponding proof obligation. The difference is that these two pragmas introduce cut points: which indicate to a prover that it may, after proving the truth of the assertion, dispose of certain other conclusions that may have been inferred at that point.
2. The verification rules for pragma Assume are significantly different. [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.]

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.

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 variable or a state abstraction (see *Abstraction of State*). Global items are presented in Global aspects (see *Global Aspects*).

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

Static Semantics

1. The *final* value of a global item or parameter of a subprogram is its value immediately following the successful call of the subprogram.
2. The *initial* 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 final value of an output of the subprogram.

   As a special case, a global item or parameter is also considered an input if it is deemed to have no observable effect on any output of the subprogram but is only used in determining a **null** value. Such a **null** value can only be specified using a an explicit `null_dependency_clause` in the Depends aspect of the subprogram (see *Depends Aspects*).

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.

Todo

In the future we may be able to permit access and aliased formal parameter specs. Target: Release 2 of SPARK 2014 language and toolset or later.

Todo

What do we do regarding null exclusion parameters? To be completed in the Milestone 3 version of this document.

Todo

What do we do regarding function access results and function null exclusion results? To be completed in the Milestone 3 version of this document.

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) and class-wide contract expressions (introduced by Pre'Class and Post'Class), if any, are in SPARK 2014.

Todo

Think about Pre'Class and Post'Class. To be completed in the Milestone 3 version of this document.

6.1.2 Subprogram Contracts

In order to extend Ada's support for specification of subprogram contracts (e.g., the Pre, Post, Pre'Class and Post'Class aspects) 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 Aspects* for further detail on Global aspects and section *Depends Aspects* for further detail on Depends aspects.

6.1.3 Contract Cases

High-Level Requirements

1. Goals to be met by language feature:
   - **Requirement:** It shall be possible to specify pre- and post-conditions in a concise way in the case that subprogram behaviour is specified in terms of what behaviour should be in each of a series of mutually-independent cases.

     Rationale: To provide a more structured way of specifying subprogram behaviour.
2. Constraints, Consistency, Semantics, General requirements:
   - Not applicable

Language Definition

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,
     Post => General_Postcondition
               and then Exactly_One_Of(A1,A2...An)
               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 initial values of formal parameters and global variables and
>
> B1 .. Bn are Boolean expressions that may also use the final values of formal parameters, global variables 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 or entry 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`.

Note: (TJJ 29/11/12) Do we need this verification rule? Could it be captured as part of the general statement about proof?

6.1.4 Global Aspects

High-level requirements

1. Goals to be met by language feature:

   - **Requirement:** It shall be possible to specify the list of global data read and updated when the subprogram is called. [Note that the data read can include data used in proof contexts, including assertions.]

     Rationale: to allow provision of at least the same functionality as SPARK 2005 and to allow modular analysis.

   - **Requirement:** It shall be possible to specify the mode (input, output or both) for each global data item.

Rationale: This matches the presentation of formal parameters, and the information is used by both flow analysis and proof.

- **Requirement:** It shall be possible to identify globals that are used only in proof contexts.

  Rationale: since the list of global data items constrains the data that can be read and updated when the subprogram is called, then the global data list needs to cover data items that are read in proof contexts.

2. Constraints:
   - No further Global-specific requirements needed.
3. Consistency:
   - **Requirement:** The mode associated with a formal parameter [of an enclosing subprogram] or volatile variable in a global data list shall be consistent with the mode associated with it at the point of its declaration.

     Rationale: this provides an early basic consistency check.
4. Semantics:
   - **Requirement:** A global data item with an input mode is read on at least one executable path.

     Rationale: by definition.
   - **Requirement:** A global data item with an output mode is written on at least one executable path.

     Rationale: by definition.
   - **Requirement:** A global data item with an output mode but no input mode is written on all executable paths.

     Rationale: to ensure that data items with output mode are always initialized on completion of a call to the subprogram.
   - **Requirement:** A global data item that is only read in a proof context shall not have an input or output mode.

     Rationale: the effect of reading data items in a proof context is fundamentally different from the reading of data items outside of a proof context, since the former does not contribute to information flow relations.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Language definition

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 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
```

where `null_global_specification` ::= **null**

Legality Rules

1. A `global_item` shall denote an entire variable or a state abstraction; this rule is a name resolution rule.

Note: (SB) This rule may eventually be relaxed to allow references to non-static constants.

---

2. Each `mode_selector` shall occur at most once in a single Global aspect.
3. A function subprogram may not have a `mode_selector` of `Output` or `In_Out` in its Global aspect.
4. `global_items` in the same Global aspect specification shall denote distinct objects or state abstractions.
5. A `global_item` occurring in a Global aspect of a subprogram aspect specification shall not denote a formal parameter of the subprogram.

Static Semantics

1. A `global_specification` that is a `global_list` is considered to be a `moded_global_list` with the `mode_selector` Input.
2. A `global_item` is *referenced* by a subprogram if:
   - It is an input or an output of the subprogram, or;
   - Its initial value is used to determine the value of an assertion expression within the subprogram, or;
   - Its initial 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.

Dynamic Semantics

There are no dynamic semantics associated with a Global aspect.

Verification Rules

There are no verification rules associated with a Global aspect of a subprogram declaration. The rules given in the Subprogram Bodies section under Global aspects are checked when a subprogram body is analyzed.

Examples

```
with Global => null; -- Indicates that the subprogram does 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 Aspects

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** It shall be possible to specify the dependency relation - that is, which outputs are dependent on which inputs - that is met by a given subprogram.

     Rationale: To allow provision of at least the same functionality as SPARK 2005 and to allow modular analysis.
   - **Requirement:** It shall be possible to refer to both global data and formal parameters in the dependency relation.

     Rationale: The inputs and outputs are given by both the global data and the formal parameters.
   - **Requirement:** It shall be possible to assume an implicit dependency relation on functions and so an explicit statement shall not be required.

     Rationale: this is typical usage and saves effort.
2. Constraints:
   - No further Depends-specific requirements needed.
3. Semantics:
   - **Requirement:** That (X,Y) is in the dependency relation for a given subprogram (i.e. X depends on Y) means that X is an output of the subprogram such that the initial value of the input Y is used to set the final value of X on at least one executable path.

     Rationale: by definition.
4. Consistency:
   - **Requirement:** The dependency relation defines an alternative view of the inputs and outputs of the subprogram and that view must be equivalent to the list of global data items and formal parameters and their modes (ignoring data items used only in proof contexts).

     Rationale: this provides a useful early consistency check.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Language Definition

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. Depends aspects are simple specifications.

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 final value of output, X, on the completion of the subprogram is at least partly determined from the initial 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 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`.

Legality Rules

1. Every `input` and `output` of a `dependency_relation` of a Depends aspect shall denote an entire variable or a state abstraction; this rule is a name resolution rule.

Note: (SB) This rule may eventually be relaxed to allow references to non-static constants as inputs.

---

1. An `input` must have a mode of **in** or **in out** and an `output` must have an mode of **in out** or **out**. [Note: As a consequence an entity which is both an `input` and an `output` shall have a mode of **in out**.]
2. 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.
3. There can be at most one `output_list` which is a **null** symbol and if it exists it must be the `output_list` of the last `dependency_clause` in the `dependency_relation`. An `input` which is in an `input_list` of a **null** `output_list` may not appear in another `input_list` of the same `dependency_relation`.
4. The entity denoted by an `output` in an `output_list` shall not be denoted by any other `output` in that `output_list` or any other `output_list`.
5. The entity denoted by an `input` in an `input_list` shall not be denoted by any other `input` in that `input_list`.
6. Every `output` of the subprogram shall appear in exactly one `output_list`.
7. Every `input` of the subprogram shall appear in at least one `input_list`.
8. A `null_dependency_clause` shall not have an `input_list` of **null**.

Static Semantics

1. The grammar terms `input` and `output` have the meaning given to input and output given in *Subprogram Declarations*.
2. A `dependency_clause` has the meaning that the final value of every `output` in the `output_list` is dependent on the initial value of every `input` in the `input_list`.
3. 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)).]
4. A `dependency_clause` of the form A =>+ A has the same meaning as A => A.
5. A `dependency_clause` with a **null** `input_list` means that the final value of each `output` in the `output_list` does not depend on any `input`, other than itself, if the `output_list =>+` **null** self-dependency syntax is used.
6. A `null_dpendency_clause` represents a *sink* for each `input` in the `input_list`. The `inputs` in the `input_list` have no discernible effect from an information flow analysis viewpoint. [The purpose of a `null_dependency_clause` is to facilitate the abstraction and calling of subprograms whose implementation is not in SPARK 2014.]
7. 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).]
8. A function which does not have an explicit Depends aspect is assumed to have the `dependency_relation` that its result is dependent on all of its inputs. [Generally a Depends aspect is not required for functions.]

Todo

Add rules relating to volatile state. To be completed in the Milestone 3 version of this document.

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

There are no verification rules associated with a Depends aspect of a subprogram declaration. The rules given in the Subprogram Bodies section under Depends aspects are checked when a subprogram body is a analyzed.

Todo

Consider whether to capture the rules from SPARK 2005 about flow=auto mode in this document or whether it is purely a tool issue (in SPARK 2005, in flow=auto mode if a subprogram is missing a dependency relation then the flow analysis assumes all outputs of the subprogram are derived from all of its inputs).

Examples

```
procedure P (X, Y, Z in : Integer; Result : out Boolean)
with Depends => (Result => (X, Y, Z));
-- The final value of Result depends on the initial 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 final values of A and B depend on the initial values of X and Y.
-- The final value of C depends on the initial values of X and Z.
```

```
-- The final value of D depends on the initial value of Y.
-- The final 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 final value of A depends on the initial value of A as well as the
-- initial values of X and Y.
-- Similarly, the final value of B depends on the initial value of B
-- as well as the initial values of A, X and Y.
-- The final value of C depends on the initial value of C and Z.
-- The final value of D depends only on the initial 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

High-level requirements

1. Goals to be met by language feature:

   - **Requirement:** It shall be possible to specify functions which are used for testing and verification only. Their presence should have no effect on the functionality of program execution which terminates normally (without exception).

     Rationale: In principle such functions could be removed from the code (possibly automatically by the compiler) on completion of testing and verification and have no effect on the functionality of the program.

   - **Requirement:** It shall be possible to specify functions which are used for formal verification only which have no implementation.

     Rationale: A function used for formal verification purposes may be difficult (or impossible) to specify or implement in SPARK 2014. A function without an implementation will be defined, for proof purposes, in an external proof tool.

2. Constraints:

   - In order to be removed they can only be applied in places where it can be ascertained that they will not be called during normal execution of the program (that is with test and verification constructs disabled).
   - A function without an implementation cannot be called during execution of a program.

3. Consistency:

   Not applicable.

4. Semantics:

   Not applicable.

5. General requirements:

   - See also section *Ghost Entities*.

Language definition

In SPARK 2014 a function may be denoted as being a Ghost function using the boolean `aspect_mark` Ghost. This shows an intent that this function should only be called directly, or indirectly from within assertion expressions excluding predicate subtypes. In Ada subtype predicates are executed irrespective of the assertion policy.

Legality Rules

1. A function with a Ghost `aspect_mark` in the `aspect_specification` of its declaration may only be called from within an assertion expression, excluding subtype predicates, or from within another ghost function.

Static Semantics

1. There are no static semantics associated with Ghost aspects.

Dynamic Semantics

1. There are no dynamic semantics associated with Ghost aspects.

Verification Rules

1. There are no verification rules associated with Ghost aspects.

Examples

```
function A_Ghost_Function (X, Y : Integer) return Integer
with
   Pre  => X + Y <= Integer'Last,
   Post => X + Y > 0,
   Ghost;
-- The body of the function is declared elsewhere.

function A_Ghost_Expression_Function (X : Y : Integer) return Boolean is (X < Y)
with
   Ghost;
```

6.1.7 Non-Executable Ghost Functions

SPARK 2014 permits the use of non-executable ghost functions that have no body and are used in formal specification and verification only. A non-executable ghost function is introduced by declaring a ghost function with an Import `aspect_mark` in its declaration.

If a call is made, directly or indirectly, to this function other than in an assertion expression which is not a subtype predicate, or if the assertion policy Ignore is not selected, an error will be reported when an attempt is made to build and execute the program.

It is expected that the definition of a non-executable ghost function will be provided within an external proof tool.

There are no additional legality rules, static or dynamic semantics or verification rules associated with non-executable ghost functions.

Examples

```
function A_Non_Executable_Function (X, Y : T) return Integer
with
   Ghost,
   Import;
```

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 arise to flow errors. Perhaps we need an aspect to describe the strict view of a parameter if it is different to the specified Ada mode of the formal parameter? To be completed in the Milestone 3 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.

6.3.3 Global Aspects

If a subprogram does not have a separate declaration then the Global aspect is applied to the declaration of its body or body stub. The implementation of a subprogram body must be consistent with its 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.

Syntax

No extra syntax is associated with Global aspects on subprogram bodies.

Legality Rules

No extra legality rules are associated with Global aspects on subprogram bodies.

Static Semantics

No extra static semantics are associated with Global aspects on subprogram bodies.

Dynamic Semantics

No extra dynamic semantics are associated with Global aspects on subprogram bodies.

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 aspect 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 is mode **in** and has a `mode_selector` of Input;
   - a `global_item` that denotes an output but not an input is always fully initialized on every call of the subprogram, is mode **out** and has a `mode_selector` of Output;
   - otherwise the `global_item` denotes both an input and an output, is mode **in out** and has a `mode_selector` of In_Out.
3. 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.

Todo

Consider how implicitly generated proof obligations associated with runtime checks should be viewed in relation to Proof_In. To be addressed in the Milestone 4 version of this document.

6.3.4 Depends Aspects

If a subprogram does not have a separate declaration then the Depends aspect is applied to the declaration of its its body or body stub. The implementation of a subprogram body must be consistent with its 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.

Syntax

No extra syntax is associated with Depends aspects on subprogram bodies.

Legality Rules

No extra legality rules are associated with Depends aspects on subprogram bodies.

Static Semantics

No extra static semantics are associated with Depends aspects on subprogram bodies.

Dynamic Semantics

No extra dynamic semantics are associated with Depends aspects on subprogram bodies

Verification Rules

1. Each `output` given in the Depends aspect must be an `output` in the implementation of the subprogram body and the `output` must depend on all, but only, the `inputs` given in the `input_list` associated with the `output`.
2. Each `output` of the implementation of the subprogram body is present as an output in the Depends aspect.
3. Each `input` of the Depends aspect is an `input` of the implementation of the subprogram body.

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 (whether the call is dispatching or not).

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.

High-Level Requirements

1. Goals to be met by language feature:
    - Not applicable.
2. Constraints:
    - **Requirement:** An entity that may be updated on a call to a subprogram may not be referred to by distinct names within that subprogram.

      Rationale: Flow analysis specifications are presented and analyzed in terms of names rather than the entities to which those names refer.
3. Semantics:
    - Not applicable.
4. Consistency:
    - Not applicable.
5. General requirements:
    - Not applicable.

Language Definition

Syntax

No extra syntax is associated with anti-aliasing.

Legality Rules

No extra legality rules are associated with anti-aliasing.

Static Semantics

No extra static semantics are associated with anti-aliasing.

Dynamic Semantics

No extra dynamic semantics are associated with anti-aliasing.

Verification Rules

1. In SPARK 2014, 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. In SPARK 2014, 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. In SPARK 2014, 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.

6.5 Return Statements

No extensions or restrictions.

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 *Common Rationale for Refined Aspects*).

CHAPTER

SEVEN

PACKAGES

A *compile-time* constant is a static expression or an expression involving only static expressions [for example an aggregate of static expressions].

In SPARK 2014 a declaration or statement occurring immediately within the package shall only read – whether directly or indirectly – values derived only from compile-time constants.

Among other things this restriction avoids the need to have dependency relations applied to packages.

7.1 Package Specifications and Declarations

Verification Rules

1. Each `basic_declaration` occurring in the visible or private part of a package shall not read, directly or indirectly, any value which is not entirely derived from compile-time constants.

7.1.1 Abstraction of State

The variables declared immediately within a package Q, its embedded packages and its private descendants constitute the state of Q.

The variable declarations are only visible to clients of Q if they are declared in the visible part of Q. The declarations of all other variables are *hidden* from a client of Q. Though the variables are hidden they still form part (or all) of the state of Q and this *hidden state* cannot be ignored for static analyses and proof. *State abstraction* is the means by which this hidden state is managed for static analyses and proof.

SPARK 2014 extends the concept of state abstraction to provide hierarchical data abstraction whereby the hidden state of a package Q may be refined over a tree of private descendants or embedded packages of Q. This provides data refinement similar to the refinement available to types whereby a record may contain fields which are themselves records.

7.1.2 Volatile State

Volatile state is a volatile variable or a volatile state abstraction.

The abstract state aspect provides a way to designate a named abstract state as being volatile, usually representing an external input or output. A volatile variable is designated as volatile using a Volatile aspect possibly with a further designation of whether it is an input or an output.

The read or update of a volatile variable or state abstraction is considered to be both a read and an update of the entity. In Global and Depends aspects this means that volatile entities will be regarded as being both an input and an

output and this fact may be stated explicitly in those aspects, for example by using the `In_Out` mode in the Global aspect. However if a variable or abstract state is explicitly designated as being a Volatile Input or a Volatile Output, an abbreviated form of the Global and Depends aspect is permitted which gives a more intuitive view of the globals and the dependency relation.

If the variable or state abstraction is designated as Volatile Input, then it may only appear as an Input in the Global aspect. There is an implicit declaration that it is also an Output. In a Depends aspect it need not appear as an output as an implicit self dependency of the entity will be declared.

If the variable or state abstraction is designated as Volatile Output, then it may only appear as an Output in the Global aspect. There is an implicit declaration that it is also an Input. In a Depends aspect it need not appear as an input as an implicit self dependency of the entity will be declared.

A volatile variable or volatile state abstraction cannot be mentioned directly in an assertion expression as the reading of a volatile may affect its value.

Todo

More details on volatile variables and definition of a complete model. At the very least, if V is a Volatile Input variable should not have the following assertion provable: T1 := V; T2 := V; pragma Assert (T1 = T2); To be completed in the Milestone 3 version of this document.

Todo

Need to describe the conditions under which a volatile variable can be a parameter of a subprogram. To be completed in the Milestone 3 version of this document.

Todo

Consider more than just simple Volatile Inputs and Outputs; Latched outputs, In_Out volatiles, etc. To be completed in the Milestone 4 version of this document.

7.1.3 Abstract State Aspect

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** It shall be possible to provide an abstracted view of hidden state that can be referred to in specifications of program behavior.

     Rationale: this allows modular analysis, since modular analysis is performed before all package bodies are available and so before all hidden state is known. Abstraction also allows the management of complexity.
2. Constraints:
   - No further abstract state-specific requirements.
3. Consistency:
   - No further abstract state-specific requirements.
4. Semantics:
   - No further abstract state-specific requirements.
5. General requirements:

- See also section *Generic Language-Independent Requirements*.

Language Definition

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, `private_part`, packages nested within these, or within private descendants of the package. 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. Those variables 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* that state abstraction. If a package body contains, for example, a nested package, 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 P must include in its Global aspect the abstract state names with the correct mode that represent the hidden state referenced by P. If P has a Depends aspect then the abstract state names must appear as inputs and outputs of P, as appropriate, in the `dependency_relation` of the Depends aspect.

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

Syntax

```
abstract_state_list         ::= null
                              | state_name_with_properties
                              | (state_name_with_properties { , state_name_with_properties } )
state_name_with_properties ::= state_name
                              | ( state_name with property_list )
property_list               ::= property { , property }
property                    ::= simple_property
                              | name_value_property
simple_property             ::= identifier
name_value_property         ::= identifier => expression
state_name                  ::= defining_identifier
```

Legality Rules

1. The `identifier` of a `simple_property` shall be Volatile, Input, or Output.
2. There shall be at most one occurrence of the `identifiers` Volatile, Input and Output in a single `property_list`.
3. If a `property_list` includes Volatile, then it shall also include exactly one of Input or Output.
4. If a `property_list` includes either Input or Output, then it shall also include Volatile.
5. The `identifier` of a `name_value_property` shall be Integrity.
6. If a `property_list` includes Integrity then it shall be the final property in the list. [This eliminates the possibility of a positional association following a named association in the property list.]
7. A package_declaration or generic_package_declaration shall have a completion [(a body)] if it contains a non-null Abstract State aspect specification.

Static Semantics

1. The visible state and state abstractions of a package P consist of:
   - any variables declared immediately within the visible part of P; and

- any state abstractions declared by the Abstract State aspect specification (if any) of package P; and
- the visible state and state abstractions of any packages declared immediately within the visible part of P.

2. The hidden state of a package P consists of:
   - any variables declared immediately within the private part or body of P;
   - the state abstractions of any packages declared immediately within the visible part of P; and
   - the visible state and state abstractions of any packages declared immediately within the private part or body of P, and of any private child units of P or of their public descendants.

Note: (SB) These definitions may eventually be expanded to include non-static constants, not just variables.

---

3. 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.

---

4. [A state abstraction shall only be named in contexts where this is explicitly permitted (e.g., as part of a Globals 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.]
5. A **null** `abstract_state_list` specifies that a package contains no hidden state. [The specification is checked when the package is analyzed.]
6. A *volatile* state abstraction is one declared with a property list that includes the Volatile property, and either Input or Output.

Verification Rules

There are no Verification Rules associated with the Abstract State aspect.

Dynamic Semantics

There are no Dynamic Semantics associated with the Abstract State aspect.

Examples

```
package Q
with
   Abstract_State => State           -- Declaration of abstract state named State
is                                   -- representing internal state of Q.
  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 Volatile, Input))
is                           -- Three abstract state names are declared A, B & C.
                             -- A and B are non-volatile abstract states
   ...                       -- C is designated as a volatile input.
end X;

package Sensor -- simple volatile, input device driver
   with Abstract_State => (Port with Volatile, Input);
is
   ...
end Sensor;
```

Todo

Further semantic detail regarding Volatile state and integrity levels needs to be added, in particular in relation to specifying these properties for variables which are declared directly within the visible part of a package specification. To be completed in the Milestone 3 version of this document.

7.1.4 Initializes Aspect

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** Flow analysis requires the knowledge of whether each variable has been initialized. It should be possible to determine this from the specification of a unit.

     Rationale: Variables and state abstractions may be initialized within a package body as well as a package specification. It follows not all initializations are visible from the specification. An Initializes aspect is applied to a package specification to indicate which variables and state abstractions are initialized by the package. This facilitates modular analysis.
2. Constraints:
   - No further Initializes-specific requirements.
3. Consistency:
   - No further Initializes-specific requirements.
4. Semantics:
   - **Requirement:** The set of data items listed in an Initializes aspect shall be fully initialized during elaboration of this package.

     Rationale: To ensure that listed data items are always initialized before use.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Language Definition

The Initializes aspect is introduced by an `aspect_specification` where the `aspect_mark` is Initializes and the `aspect_definition` must 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
```

Todo

Provide language definition for Initializes aspect. To be completed in the Milestone 3 version of this document.

7.1.5 Initial Condition Aspect

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** It shall be possible to formally specify the result of performing package elaboration.

     Rationale: This specification behaves as a postcondition for the result of package elaboration and so establishes the "pre-condition" that holds at the point of beginning execution of the program proper. Giving an explicit postcondition supports modular analysis.
2. Constraints:
   - No further Initial Condition-specific requirements.
3. Consistency:
   - No further Initial Condition-specific requirements.
4. Semantics:
   - **Requirement:** The predicate given by the Initial Condition aspect should evaluate to True at the point at which elaboration of the package, its embedded packages and its private descendants has completed.

     Rationale: By definition.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Language Definition

The Initial Condition aspect is introduced by an `aspect_specification` where the `aspect_mark` is "Initial_Condition" and the `aspect_definition` must be an `expression`.

Todo

Provide language definition for Initial Condition aspect. To be completed in the Milestone 3 version of this document.

7.2 Package Bodies

Verification Rules

1. Each declaration of the `declarative_part` of a `package_body` shall not read, directly or indirectly, any value which is not entirely derived from compile-time constants.
2. Each statement of a `handled_sequence_of_statements` of a `package_body` shall not read, directly or indirectly, a value which is not entirely derived entirely from compile-time constants.

7.2.1 State Refinement

A `state_name` declared by an Abstract State aspect in the specification of a package Q is an abstraction of the non-visible variables declared in the private part, body, or private descendants of Q, which together form the hidden state, of Q. In the body of Q each `state_name` has to be refined by showing which variables and subordinate abstract states are represented by the `state_name` (its constituents). A Refined State aspect in the body of Q is used for this purpose.

In the body of a package the constituents of the refined `state_name`, the refined view, has to be used rather than the abstract view of the `state_name`. Refined global, depends, pre and post aspects are provided to express the refined view.

In the refined view the constituents of each `state_name` have to be initialized consistently with their appearance or omission from the Package Depends or Initializes aspect of the package.

7.2.2 Common Rationale for Refined Aspects

Where it is possible to specify subprogram behavior using a language feature that refers to abstract state, it should be possible to define a corresponding *refined* version of the language feature that refers to the decomposition of that abstract state.

The rationale for this is as follows:

1. The semantics of properties defined in terms of abstract state can only be precisely defined in terms of the corresponding concrete state, though nested abstraction is also necessary to manage hierarchies of data.
2. There may be multiple possible refinements for a given abstract specification and so the user should be able to specify what they actually want.
3. This is necessary to support development via stepwise refinement.

7.2.3 Refined State Aspect

High-level requirements

1. Goals to be met by language feature:

   - **Requirement:** For each state abstraction, it shall be possible to define the set of hidden state items that implement or *refine* that abstract state (where the hidden state items can either be concrete state or further state abstractions).

     Rationale: see section *Common Rationale for Refined Aspects*.

2. Constraints:

- **Requirement:** Each item of hidden state must map to exactly one state abstraction.

  Rationale: all hidden state must be covered since otherwise specifications referring to abstract state may be incomplete; each item of that hidden state must map to exactly one abstraction to give a clean and easily understandable abstraction, and for the purposes of simplicity of analysis.

- **Requirement:** Each item of abstract state covered by the package shall be mapped to at least one item of hidden state (either concrete state or a further state abstraction).

  Rationale: the semantics of properties defined in terms of abstract state can only be precisely defined in terms of the corresponding concrete state.

- **Requirement:** Each item of hidden state should appear in at least one global data list within the package body.

  Rationale: If this is not the case, then there is at least one hidden state item that is not used by any subprogram.

3. Consistency:
   - No further Refined state-specific requirements needed.
4. Semantics:
   - No further Refined state-specific requirements needed.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Todo

The consistency rules will be updated as the models for volatile variables and integrity levels are defined. To be completed in the Milestone 3 version of this document.

Todo

Consider whether it should be possible to refine null abstract state onto hidden state. *Rationale: this would allow the modeling of programs that - for example - use caches to improve performance.* To be completed in the Milestone 3 version of this document.

Todo

Consider whether it should be possible to refine abstract onto hidden state without any restrictions, although the refinement would be checked and potential issues flagged up to the user.

Rationale: there are a number of different possible models of mapping abstract to concrete state - especially when volatile state is being used - and it might be useful to provide greater flexibility to the user. In addition, if a facility is provided to allow users to step outside of the language when refining depends, for example, then it may be necessary to relax the abstraction model as well as relaxing the language feature of direct relevance.*

To be completed in the Milestone 3 version of this document.

Language Definition

The Refined State aspect is introduced by an `aspect_specification` where the `aspect_mark` is "Refined_State" and the `aspect_definition` must follow the grammar of `state_and_category_list` given below.

Syntax

```
state_and_category_list          ::= (state_and_category {, state_and_category})
state_and_category               ::= abstract_state_name => constituent_with_property_list
abstract_state_name              ::= state_name
constituent_with_property_list   ::= constituent_with_property
                                   | (constituent_with_property {, constituent_with_property})
constituent_with_property        ::= constituent
                                   | (constituent_list with property_list)
constituent_list                 ::= constituent
                                   | (constituent {, constituent})
```

where

```
constituent ::= variable_name | state_name
```

Todo

Provide language definition for Refined_State aspect. To be completed in the Milestone 3 version of this document.

7.2.4 Abstract State and Package Hierarchy

Todo

We need to consider the interactions between package hierarchy and abstract state. Do we need to have rules restricting access between parent and child packages? Can we ensure abstract state encapsulation? To be completed in the Milestone 3 version of this document.

7.2.5 Initialization Refinement

Todo

Provide Verification Rules for Initializes aspect in the presence of state abstraction. To be completed in the Milestone 3 version of this document.

7.2.6 Refined Global Aspect

Todo

The subject of refined Global, Depends, Pre and Post aspects is still under discussion (and their need questioned) and so the subsections covering these aspects is subject to change. To be resolved and completed by Milestone 3 version of this document.

High-level requirements

1. Goals to be met by language feature:

- **Requirement:** Where a global data list referring to abstract state has been specified for a subprogram, it shall be possible to provide a refined global data list that takes account of the refinement of that abstract state.

  Rationale: see section *Common Rationale for Refined Aspects*.

2. Constraints:
   - No further Refined Global-specific requirements needed.
3. Consistency:
   - Let *Abstract* be the abstraction function defined by state refinement (such that *Abstract* is the identity function when applied to visible state). Let *G* be the global data list and *RG* be the refined global data list. Then:
     - **Requirement:** If *X* appears in *RG* but not all constituents of *Abstract (X)* appear in *RG* then *Abstract (X)* must appear in *G* with at least input mode.

       Rationale: In this case, *Abstract (X)* is not fully initialized by the subprogram and the relevant components must be intialized prior to calling the subprogram.
     - **Requirement:** If *Y* appears in *G*, then at least one *X* such that *Abstract (X) = Y* must appear in *RG*.

       Rationale: By definition of abstraction.
     - **Requirement:** Refinement of modes:
       - If the mode of *X* in *RG* indicates it is **not** used in a proof context, then that mode must be a mode of *Abstract (X)* in *G*.
       - If the mode of *X* in *RG* indicates it **is** used in a proof context and *Abstract(X)* does not have another mode according to the above rules, then the mode of *Abstract(X)* shall indicate it is only used in proof contexts.

       Rationale: In general, modes should be preserved by refinement. However, if one refinement constituent of a state abstraction has an input and/or output mode, then it is no longer of interest whether another constituent is only used in a proof context.
4. Semantics:
   - As per Global aspect.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Todo

The consistency rules will be updated as the model for volatile variables is defined. To be completed in the Milestone 3 version of this document.

Todo

If it ends up being possible to refine null abstract state, then refinements of such state could appear in refined globals statements, though they would need to have mode in out. To be completed in the Milestone 3 version of this document.

Language Definition

A subprogram declared in the visible part of a package may have a Refined Global aspect applied to its body or body stub. The Refined Global aspect defines the global items of the subprogram in terms of the `constituents` of a `state_name` of the package rather than the `state_name`.

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

Todo

Provide language definition for Refined_Global aspect. To be completed in the Milestone 3 version of this document.

7.2.7 Refined Depends Aspect

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** Where a dependency relation referring to abstract state has been given, it shall be possible to specify a refined dependency relation that takes account of the refinement of that abstract state.

     Rationale: see section *Common Rationale for Refined Aspects*.
2. Constraints:
   - No further Refined depends-specific requirements needed.
3. Consistency:
   - **Requirement:** The refined dependency relation defines an alternative view of the inputs and outputs of the subprogram and that view must be equivalent to the refined list of global data items and formal parameters and their modes (ignoring data items used only in proof contexts).

     Rationale: this provides a useful early consistency check.
   - Let *Abstract* be the abstraction function defined by state refinement (such that *Abstract* is the identity function when applied to visible state). Let *D* be a dependency relation and *RD* be the corresponding refined dependency relation. Then:
     - **Requirement:** If *(X,Y)* is in *RD* - i.e. *X* depends on *Y* - then *(Abstract(X), Abstract(Y))* is in *D*.

       Rationale: dependencies must be preserved after abstraction.
     - **Requirement:** If *(X,Y)* is in *RD* and there is *A* such that *Abstract(A)=Abstract(X)* but there is no *B* such that *(A,B)* is in *RD*, then *(Abstract(X),Abstract(X))* is in *D*.

       Rationale: In this case, *Abstract (X)* is not fully initialized by the subprogram and the relevant components must be initialized prior to calling the subprogram.
     - **Requirement:** If *(S,T)* is in *D* then there shall exist *(V,W)* in *RD* such that *Abstract(V)=S* and *Abstract(W)=T*.

       Rationale: By definition of abstraction.
4. Semantics:
   - As per Depends aspect.
5. General requirements:

- See also section *Generic Language-Independent Requirements*.

Todo

The consistency rules will be updated as the model for volatile variables is defined. To be completed in the Milestone 3 version of this document.

Todo

If it is possible to refine null abstract state, then refinements of such state could appear in refined depends statements, but wouldn't map to anything in the depends relation itself and would need to have mode in/out in the refined depends. To be completed in the Milestone 3 version of this document.

Language Definition

A subprogram declared in the visible part of a package may have a Refined Depends aspect applied to its body or body stub. The Refined Depends aspect defines the `dependency_relation` of the subprogram in terms of the `constituents` of a `state_name` of the package rather than the `state_name`.

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

Todo

Provide language definition for Refined_Depends aspect. To be completed in the Milestone 3 version of this document.

7.2.8 Refined Precondition Aspect

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** Where a precondition has been provided for a subprogram declaration, it shall be possible to state a refined precondition that refers to concrete rather than abstract state and/or concrete rather than abstract type detail.

     Rationale: See section *Common Rationale for Refined Aspects*.
2. Constraints:
   - No further Refined precondition-specific requirements needed.
3. Consistency:
   - **Requirement:** The refined precondition of the subprogram must be implied by the precondition.

     Rationale: standard definition of proof refinement.
4. Semantics:
   - As per the semantics of the Precondition aspect.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Language Definition

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` must be a Boolean `expression`.

Todo

Provide language definition for Refined_Pre aspect. To be completed in the Milestone 3 version of this document.

7.2.9 Refined Postcondition Aspect

High-level requirements

1. Goals to be met by language feature:
   - **Requirement:** Where a post-condition has been provided for a subprogram declaration, it shall be possible to state a refined post-condition that refers to concrete rather than abstract state and/or concrete rather than abstract type detail.

     Rationale: See section *Common Rationale for Refined Aspects*.
2. Constraints:
   - No further Refined post-condition-specific requirements needed.
3. Consistency:
   - **Requirement:** The post-condition of the subprogram must be implied by the refined post-condition.

     Rationale: standard definition of proof refinement.
4. Semantics:
   - As per the semantics of the Post-condition aspect.
5. General requirements:
   - See also section *Generic Language-Independent Requirements*.

Language Definition

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 Precondition aspect is introduced by an `aspect_specification` where the `aspect_mark` is "Refined_Post" and the `aspect_definition` must be a Boolean `expression`.

Todo

Provide language definition for Refined_Post aspect. To be completed in the Milestone 3 version of this document.

Todo

refined contract_cases. To be completed in the Milestone 3 version of this document.

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

Syntax

There is no additional syntax associated with type invariants.

Legality Rules

There are no additional legality rules associated with type invariants.

Static Semantics

There are no additional static semantics associated with type invariants.

Dynamic Semantics

There are no additional dynamic semantics associated with type invariants.

Verification Rules

1. The Ada 2012 RM lists places at which an invariant check is performed. In SPARK 2014, we add the following places in order to guarantee that an instance of a type always respects its invariant at the point at which it is passed as an input parameter:

   - Before a call on any subprogram or entry that:
     - is explicitly declared within the immediate scope of type T (or by an instance of a generic unit, and the generic is declared within the immediate scope of type T), and
     - is visible outside the immediate scope of type T or overrides an operation that is visible outside the immediate scope of T, and
     - has one or more in out or in parameters with a part of type T.

   the check is performed on each such part of type T. [Note that these checks are only performed statically, and this does not create an obligation to extend the run-time checks performed in relation to type invariants.]

Todo

The support for type invariants needs to be considered further and will be completed for Milestone 3 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.

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

No extensions or restrictions.

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.

Todo

Describe model of renaming for array indexing and slices. To be completed in the Milestone 3 version of this document.

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.

8.6 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 but are targeted to be included in Release 2 of the SPARK 2014 language and toolset.

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 3 version of this document.

Todo

Add Tasking. Target: release 2 of SPARK 2014 language and toolset.

CHAPTER

TEN

PROGRAM STRUCTURE AND COMPILATION ISSUES

High-Level Requirements

1. Goals to be met by language feature:

   - **Requirement:** The ability to analyze incomplete programs.

     Rationale: In order to support incremental development and analysis. To facilitate the use of flow analysis and formal verification as early as possible in the software life-cycle.

2. Constraints, Consistency, Semantics, General requirements:

   - Interface specifications have to be provided for all modules. In analysis the module is represented by its interface specification.

Language Definition

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

High-Level Requirements

1. Goals to be met by language feature:
   - **Requirement:** State abstractions and visible variable declarations shall be visible in the limited view of a package.

     Rationale: This allows the flow analysis specifications of a package P1 to refer to the state of P2 in the case that P1 only has a limited view of P2.
2. Constraints, Consistency, Semantics, General requirements:
   - Not applicable.

Language Definition

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 aspect specification; or
   - as an `input` or `output` in a 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` occuring 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 volatile.

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

11.1 High-Level Requirements

1. Goals to be met by language feature:
   - Not applicable.
2. Constraints:
   - **Requirement:** Most explicit uses of exceptions are excluded from SPARK 2014 as described below. Exceptions can be raised implicitly (for example, by the failure of a language-defined check), but only in the case of a program with an undischarged (or incorrectly discharged, perhaps via an incorrect Assume pragma) proof obligation. Explicit raising of exceptions is dealt with similarly.

     Rationale: Raising and handling of exceptions allow forms of control flow that complicate both specification and verification of a program's behavior.
3. Consistency:
   - Not applicable.
4. Semantics:
   - Not applicable.
5. General requirements:
   - Not applicable.

11.2 Language Definition

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

Generic units are not classified as being SPARK 2014 or non-SPARK 2014. Only their instantiations are.

Todo

Constructive modular analysis of generics (including prove once, use many times). Will require significant restrictions and extra aspects to implement. To be completed in the Milestone 4 version of this document.

CHAPTER

THIRTEEN

REPRESENTATION ISSUES

The SPARK 2014 toolset must be able to analyse programs containing representation clauses and unchecked conversions. Further detail on this will be provided in a subsequent draft of this document.

Todo

Provide detail on Representation Issues. To be completed in the Milestone 4 version of this document.

CHAPTER

FOURTEEN

THE STANDARD LIBRARIES

This chapter will describe 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.

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.

Todo

Provide detail on Standard Libraries. To be completed in the Milestone 4 version of this document.

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

Note that the details of false alarm management are still TBD and so there is currently no equivalent of the accept annotation in the SPARK 2005 body. To be completed in the Milestone 3 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;
```

```
--  TBD: confirm that no inherits clause and check
--  whether anything needed in its place.

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;
```

```
13
14    procedure Pop(S : in out Stack; X : out Integer);
15    --# 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;
```

```
26          else
27             Result := S.Monitor_Vector(S.Stack_Pointer);
28          end if;
29          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   -- Confirm that no inherit clause in SPARK 2014.
2
3   package Stacks_14.Monitored_14 is
4
5      type Monitored_Stack is new Stacks_14.Stack with private;
6
7      overriding
8      procedure Clear(S : out Monitored_Stack)
9         with Depends => (S => null);
10
11     overriding
12     procedure Push(S : in out Monitored_Stack; X : in Integer)
13        with Depends => (S =>+ X);
14
15     function Top_Identity(S : Monitored_Stack) return Integer;
16     function Next_Identity(S : Monitored_Stack) return Integer;
17
18  private
19
20     type Monitored_Stack is new Stacks_14.Stack with
21        record
22           Monitor_Vector : Stacks_14.Vector;
23           Next_Identity_Value : Integer;
24        end record;
25
26  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.

Todo

The SPARK 2014 version of the code is not provided since the restrictions that are to be applied in terms of package visibility are yet to be determined. To be completed in the Milestone 3 version of this document.

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.

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, Pointer),
        Initializes    => (S, Pointer)
is
   procedure Push(X : in Integer)
      with Global => (In_Out => (S, Pointer));

   procedure Pop(X : out Integer)
      with Global => (Input  => S,
                      In_Out => Pointer);
end Stack_14;
```

Body in SPARK 2014:

As per SPARK 2005.

Initialized by elaboration

The following example presents how a package's concrete state can be initialized at the statements section of the body. The specifications of both SPARK 2005 and SPARK 2014 are not presented since they are identical to the specifications of the previous example.

Body in SPARK 2005:

```
package body Stack_05
is
   Stack_Size : constant := 100;
   type    Pointer_Range is range 0 .. Stack_Size;
   subtype Index_Range   is Pointer_Range range 1..Stack_Size;
   type    Vector        is array(Index_Range) of Integer;

   S : Vector;
   Pointer : Pointer_Range;

   procedure Push(X : in Integer)
   is
   begin
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;

begin  -- initialization
   Pointer := 0;
   S := Vector'(Index_Range => 0);
end Stack_05;
```

Body in SPARK 2014:

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 bodies of the packages have not been included since they contain no annotation.

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, Pointer)
3  is
4     procedure Push(X : in Integer)
5        with Global => (In_Out => (S, Pointer));
6
7     procedure Pop(X : out Integer)
8        with Global => (Input  => S,
9                        In_Out => Pointer);
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    Pointer : Pointer_Range;
19 end Stack_14;
```

Private, abstract state, refining onto concrete states in body

Initialized by procedure call

In this example, the abstract state declared at the specification is refined at the body. Procedure *Init* can be invoked by users of the package, in order to initialize the state.

Specification in SPARK 2005:

```
package Stack_05
--# own State;
is
   procedure Push(X : in Integer);
   --# global in out State;

   procedure Pop(X : out Integer);
   --# global in out State;

   procedure Init;
   --# global    out State;
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;
is
   Stack : Stack_Type;

   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;

   procedure Init
   --# global    out Stack;
   is
   begin
      Stack.Pointer := 0;
      Stack.S := Vector'(Index_Range => 0);
   end Init;
end Stack_05;
```

Specification in SPARK 2014:

```
package Stack_14
   with Abstract_State => State
is
   procedure Push(X : in Integer)
      with Global => (In_Out => State);

   procedure Pop(X : out Integer)
      with Global => (In_Out => State);

   procedure Init
      with Global => (Output => State);
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;

   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) -- 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)
      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;
end Stack_14;
```

Initialized by package body statements

This example introduces an abstract state at the specification and refines it at the body. The constituents of the abstract state are initialized at the statements part of the body. The specifications of the SPARK 2005 and SPARK 2014 versions of the code are as in the previous example and have thus not been included.

Body in SPARK 2005:

```
package body Stack_05
--# own State is Stack;  -- refinement of state
is
   Stack : Stack_Type;

```

```
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;
```

```
is
   procedure Push(X : in Integer);
   --# global in out Stack;

   procedure Pop(X : out Integer);
   --# 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_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
      Pointer := Pointer + 1;
      S(Pointer) := X;
   end Push;

   procedure Pop(X : out Integer)
      with Refined_Global => (Input  => S,
                              In_Out => Pointer)
   is
   begin
      X := S(Pointer);
      Pointer := Pointer - 1;
   end Pop;
begin  -- partial initialization by body statements
   S := Vector'(Index_Range => 0);
end Stack_14;
```

Initial condition

This example introduces a new SPARK 2014 feature that did not exist in SPARK 2005. On top of declaring an abstract state and promising to initialize it, we also illustrate certain conditions that will be valid after initialization. The body is not being provided since it does not add any further insight.

Specification in SPARK 2014:

```
package stack_14
   with Abstract_State     => State,
        Initializes        => State,
        Initial_Condition => Is_Empty  -- Stating that Is_Empty holds
                                       -- after initialization
is
   function Is_Empty return Boolean
      with Global => State;

   function Is_Full return Boolean
      with Global => State;

   function Top return Integer
      with Global => State,
           Pre    => not Is_Empty;

   procedure Push (X: in Integer)
      with Global => (In_Out => State),
           Pre    => not Is_Full,
           Post   => Top = X;

```

```
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:

```
1  -- Use of child packages to encapsulate state
2  package Power_05
3  --# own State;
4  is
5     procedure Read_Power(Level : out Integer);
6     --# global State;
7     --# derives Level from State;
8  end Power_05;
```

Body of Parent in SPARK 2005:

```
1  with Power_05.Source_A_05, Power_05.Source_B_05;
2
3  package body Power_05
4  --# own State is Power_05.Source_A_05.State,
5  --#              Power_05.Source_B_05.State;
6  is
7
8    procedure Read_Power(Level : out Integer)
9    --# global Source_A_05.State, Source_B_05.State;
10   --# derives
11   --#     Level
12   --#     from
13   --#         Source_A_05.State,
14   --#         Source_B_05.State;
15   is
16      Level_A : Integer;
17      Level_B : Integer;
18   begin
19      Source_A_05.Read (Level_A);
20      Source_B_05.Read (Level_B);
21      Level := Level_A + Level_B;
22   end Read_Power;
23
24 end Power_05;
```

Specifications of Private Children in SPARK 2005:

```
1  --# inherit Power_05;
2  private package Power_05.Source_A_05
3  --# own State;
4  is
5     procedure Read (Level : out Integer);
6     --# global State;
7     --# derives Level from State;
8  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
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
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:

As per SPARK 2005

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);
   --# global State;
   --# derives Level from State;
end Power_05;
```

Body in SPARK 2005:

```
package body Power_05
--# own State is Source_A.State,
--#               Source_B.State;
is

  --  Embedded package spec for Source_A
  package Source_A
  --# own State;
  is
     procedure Read (Level : out Integer);
      --# global State;
      --# derives Level from State;
```

```
   end Source_A;

   --  Embedded package spec for Source_B.
   package Source_B
   --# own State;
   is
     procedure Read (Level : out Integer);
     --# global State;
     --# derives Level from State;
   end Source_B;

   --  Embedded package body for Source_A
   package body Source_A
   is
     State : Integer;

     procedure Read (Level : out Integer)
     is
     begin
       Level := State;
     end Read;
   end Source_A;

   --  Embedded package body for Source_B
   package body Source_B
   is
     State : Integer;

     procedure Read (Level : out Integer)
     is
     begin
       Level := State;
     end Read;

   end Source_B;

   procedure Read_Power(Level : out Integer)
   --# global Source_A.State, Source_B.State;
   --# derives
   --#     Level
   --#     from
   --#         Source_A.State,
   --#         Source_B.State;
   is
      Level_A : Integer;
      Level_B : Integer;
   begin
      Source_A. Read (Level_A);
      Source_B.Read (Level_B);
      Level := Level_A + Level_B;
   end Read_Power;

end Power_05;
```

Specification in SPARK 2014:

```
-- Use of embedded packages to encapsulate state
package Power_14
   with Abstract_State => State
```

```
is
   procedure Read_Power(Level : out Integer)
      with Global  => State,
           Depends => (Level => State);
end Power_14;
```

Body in SPARK 2014:

```
package body Power_14
   with Refined_State => (State => (Source_A.State, Source_B.State))
is

  --  Embedded package spec for Source_A
  package Source_A
     with Abstract_State => State
  is
     procedure Read (Level : out Integer)
        with Global  => State,
             Depends => (Level => State);
  end Source_A;

  --  Embedded package spec for Source_B.
  package Source_B
  with
     Abstract_State => State
  is
    procedure Read (Level : out Integer)
       with Global  => State,
            Depends => (Level => State);
  end Source_B;

  --  Embedded package body for Source_A
  package body Source_A
     with Refined_State => (State => State)
  is
    State : Integer;

    procedure Read (Level : out Integer)
       with Refined_Global  => State,
            Refined_Depends => (Level => State)
    is
    begin
      Level := State;
    end Read;
  end Source_A;

  --  Embedded package body for Source_B
  package body Source_B
     with Refined_State => (State => State)
  is
    State : Integer;

    procedure Read (Level : out Integer)
       with Refined_Global  => State,
            Refined_Depends => (Level => State)
    is
    begin
      Level := State;
```

```
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. However, it is available under the "codeasm_abstract_state_refined_in_embedded_and_private_child".

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;
7  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;
```

Todo

Note that the syntax for identifying the main program in SPARK 2014 *is still TBD.* To be completed in the Milestone 3 version of this document.

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);
```

```
23          exit when Stacks_14.Is_Empty(The_Stack);
24       end loop;
25    end Copy_14;
```

Specification of input port in SPARK 2014:

```
1  package Input_Port_14
2     with Abstract_State => (Input_State with Volatile, Input)
3  is
4     procedure Read_From_Port(Input_Value : out Integer)
5        with Global  => (Input => Input_State),
6             Depends => (Input_Value => Input_State);
7  end Input_Port_14;
```

Specification of output port in SPARK 2014:

```
1  package Output_Port_14
2     with Abstract_State => (Output_State with Volatile, Output)
3  is
4     procedure Write_To_Port(Output_Value : in Integer)
5        with Global  => (Output => Output_State),
6             Depends => (Output_State => Output_Value);
7  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.

```
1  package body Input_Port_14
2  is
3     Input_State : Integer
4        with Address => 16#CAFE#,
5             Volatile;
6
7     procedure Read_From_Port(Input_Value : out Integer)
8     is
9     begin
10       Input_Value := Input_State;
11    end Read_From_Port;
12 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.

```
1  package body Output_Port_14
2  is
3     Output_State : Integer
4        with Address => 16#CAFE#,
5             Volatile;
6
7     procedure Write_To_Port(Output_Value : in Integer)
8     is
9     begin
10       Output_State := Output_Value;
11    end Write_To_Port;
12 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

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined. Note that this also applies to the use of the Always_Valid annotation.* To be completed in the Milestone 3 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

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 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:

```
package Switch
--# own in State;
is

   type Reading is (on, off, unknown);

   function ReadValue return Reading;
   --# global in State;

end Switch;
```

Component Switch specifications in SPARK 2005:

```
--# inherit Switch;
private package Switch.Val1
--# own in State;
is
   function Read return Switch.Reading;
   --# global in State;

end Switch.Val1;
```

```
--# inherit Switch;
private package Switch.Val2
--# own in State;
is
   function Read return Switch.Reading;
   --# global in State;

end Switch.Val2;
```

```
--# inherit Switch;
private package Switch.Val3
--# own in State;
is
   function Read return Switch.Reading;
   --# global in State;

end Switch.Val3;
```

Switch body in SPARK 2005:

```
with Switch.Val1;
with Switch.Val2;
with Switch.Val3;
package body Switch
--# own State is in Switch.Val1.State,
--#              in Switch.Val2.State,
--#              in Switch.Val3.State;
is

   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;
   type ConvertToValueArray is array (Reading) of Value;
   type ConvertToReadingArray is array (Score) of Reading;

   ConvertToValue : constant ConvertToValueArray := ConvertToValueArray'(on => 1,
                                                                         unknown => 0,
                                                                         off => -1);
```

```
18      ConvertToReading : constant ConvertToReadingArray :=
19                                        ConvertToReadingArray'(-3 .. -2 => off,
20                                                               -1 .. 1 => unknown,
21                                                               2 ..3 => on);
22
23      function ReadValue return Reading
24      --# global in Val1.State;
25      --#        in Val2.State;
26      --#        in Val3.State;
27      is
28         A, B, C : Reading;
29      begin
30          A := Val1.Read;
31          B := Val2.Read;
32          C := Val3.Read;
33          return ConvertToReading (ConvertToValue (A) +
34             ConvertToValue (B) + ConvertToValue (C));
35      end ReadValue;
36
37   end Switch;
```

Abstract Switch specification in SPARK 2014:

```
1   package Switch
2      with Abstract_State => ((State with Volatile, Input))
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 specifications in SPARK 2014:

```
1   private package Switch.Val1
2      with Abstract_State => ((State with Volatile, Input))
3   is
4      function Read return Switch.Reading
5         with Global => (Input => State);
6   end Switch.Val1;
```

```
1   private package Switch.Val2
2      with Abstract_State => ((State with Volatile, Input))
3   is
4      function Read return Switch.Reading
5         with Global => (Input => State);
6   end Switch.Val2;
```

```
1   private package Switch.Val3
2      with Abstract_State => ((State with Volatile, Input))
3   is
4      function Read return Switch.Reading
5         with Global => (Input => State);
6   end Switch.Val3;
```

Switch body in SPARK 2014:

```
1   with Switch.Val1;
2   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) with Volatile, Input))
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.

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

Specification in SPARK 2005:

```
package Device
--# own State;
--# initializes State;
is
  procedure Write (X : in Integer);
  --# global in out State;
  --# derives State from State, X;
end Device;
```

Body in SPARK 2005:

```
package body Device

--# own State is        OldX,

--#              in     StatusPort,

--#                 out Register;

-- refinement on to mix of external and ordinary variables

is

  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;
```

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

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 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
7     procedure Read (Value : out Integer)
8        with Global  => (Input => State),
9             Depends => (Value => State),
10            Post    => (Data_Is_Valid (Value));
11
12 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;
```

```
with Processing;
package Calculate
is

   procedure Read_Calculated_Value (Value : out Integer)
      with Global  => (Input => (Processing.State, Raw_Data.State)),
           Depends => (Value => (Processing.State, Raw_Data.State)),
           Post    => (Raw_Data.Data_Is_Valid (Value));

end Calculate;
```

Package nested inside package

See section Private, abstract state, refining onto concrete state of embedded package.

Package nested inside subprogram

This example is a modified version of that given in section Refinement of external state - voting input switch. It illustrates the use of a package nested within a subprogram.

Abstract Switch specification in SPARK 2005:

```
package Switch
--# own in State;
is

   type Reading is (on, off, unknown);

   function ReadValue return Reading;
   --# global in State;

end Switch;
```

Component Switch specifications in SPARK 2005:

As in Refinement of external state - voting input switch

Switch body in SPARK 2005:

```
with Switch.Val1;
with Switch.Val2;
with Switch.Val3;
package body Switch
--# own State is in Switch.Val1.State,
--#              in Switch.Val2.State,
--#              in Switch.Val3.State;
is

   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;


   function ReadValue return Reading
   --# global in Val1.State;
   --#        in Val2.State;
   --#        in Val3.State;
   is
```

```
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 Volatile, Input))
```

```
is
   type Reading is (on, off, unknown);

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

Component Switch specification in SPARK 2014:

As in Refinement of external state - voting input switch

Switch body in SPARK 2014:

```
with Switch.Val1;
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.Val3.State) with Volatile, Input))
is
   subtype Value is Integer range -1 .. 1;
   subtype Score is Integer range -3 .. 3;

   function ReadValue return Reading
      with Refined_Global => (Input => (Val1.State, Val2.State, Val3.State))
   is
      A, B, C : Reading;

      --  Embedded package to provide the capability to synthesize three inputs
      --  into one.
      package Conversion
      is
         function Convert_To_Reading
            (Val_A : Switch.Reading;
             Val_B : Switch.Reading;
             Val_C : Switch.Reading) return Switch.Reading;
      end Conversion;

      package body Conversion
      is
         type ConvertToValueArray is array (Switch.Reading) of Switch.Value;
         type ConvertToReadingArray is array (Switch.Score) of Switch.Reading;
         ConvertToValue : constant ConvertToValueArray := ConvertToValueArray'(Switch.on => 1,
                                                                               Switch.unknown => 0,
                                                                               Switch.off => -1);

         ConvertToReading : constant ConvertToReadingArray :=
                                     ConvertToReadingArray'(-3 .. -2 => Switch.off,
                                                            -1 .. 1  => Switch.unknown,
                                                            2 ..3    => Switch.on);

         function Convert_To_Reading
            (Val_A : Switch.Reading;
             Val_B : Switch.Reading;
             Val_C : Switch.Reading) return Switch.Reading
         is
         begin
            return ConvertToReading (ConvertToValue (Val_A) +
```

```
47                      ConvertToValue (Val_B) + ConvertToValue (Val_C));
48            end Convert_To_Reading;
49
50         end Conversion;
51      begin  -- begin statement of ReadValue function
52          A := Val1.Read;
53          B := Val2.Read;
54          C := Val3.Read;
55          return Conversion.Convert_To_Reading
56                    (Val_A => A,
57                     Val_B => B,
58                     Val_C => C);
59      end ReadValue;
60   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:

```
with P_05;
package body Q_05
is
   Q_State : Integer;

   procedure Init(S : out Integer)
   is
   begin
      S := 10;
   end Init;
begin
   P_05.Init(Q_State);
end Q_05;
```

Specification of package P_14 in SPARK 2014:

```
package P_14
   with Abstract_State => P_State,
        Initializes    => P_State
is
   procedure Init(S : out Integer);
end P_14;
```

Specification of package Q_14 in SPARK 2014:

```
package Q_14
   with Abstract_State => Q_State,
        Initializes    => Q_State
is
   procedure Init(S : out Integer);
end Q_14;
```

Body of package P_14 in SPARK 2014:

```
with Q_14;
package body P_14
is
   P_State : Integer;

   procedure Init(S : out Integer)
   is
   begin
      S := 5;
   end Init;
begin
   Q_14.Init(P_State);
end P_14;
```

Body of package Q_14 in SPARK 2014:

```
with P_14;
package body Q_14
is
   Q_State : Integer;

   procedure Init(S : out Integer)
   is
   begin
      S := 10;
```

```
10      end Init;
11   begin
12      P_14.Init(Q_State);
13   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:

```
1  package Input_Port
2     with Abstract_State => ((Inputs with Volatile, Input))
3  is
4     procedure Read_From_Port(Input_Value : out Integer)
5        with Global  => (Input => Inputs),
6             Depends => (Input_Value => Inputs);
7  end Input_Port;
```

Body for Assume annotation in SPARK 2014:

```
   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

         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 explostion

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.

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

A.4.3 Rule declaration annotation

See section Proof types and proof functions.

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

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 - note that it may 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:

```
package Stack
--# own State : Abstract_Stack;
is

   --  Proof functions to indicate whether or not the Stack is empty
   --  and whether or not it is full.
   --# type Abstract_Stack is abstract;
   --# function Is_Empty(Input : Abstract_Stack) return Boolean;
   --# function Is_Full(Input : Abstract_Stack) return Boolean;

   --  Proof function to give the number of elements on the stack.
   --# function Count(Input : Abstract_Stack) return Natural;

   --  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);
   --# global in out State;
   --# pre  not Is_Full(State);
   --# post not Is_Empty(State);

   --  Procedure that swaps the first two elements in a stack.
   procedure Swap2;
   --# global in out State;
   --# pre  Count(State) >= 2;
   --# post Count(State) =  Count(State~);

   --  Initializes the Stack.
   procedure Initialize;
   --# global out State;
   --# post Is_Empty (State);

   --  Other operations not included as not needed for
   --  this example.

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 2005:

```
package body Stack
--# own State is My_Stack;
is
   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

```
1   package Stack
2      with Abstract_State => State
3   is
4      -- We have to turn the proof functions into actual functions
5      function Is_Empty return Boolean
6         with Global => (Input => State);
7
8      function Is_Full return Boolean
9         with Global => (Input => State);
10
11     function Count return Natural
12        with Global => (Input => State);
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        with Global => (In_Out => State),
19             Pre    => not Is_Full,
20             Post   => not Is_Empty;
21
22     --  Procedure that swaps the first two elements in a stack.
23     procedure Swap2
```

```
24          with Global => (In_Out => State),
25               Pre    => Count >= 2,
26               Post   => Count = Count'Old;
27
28       --  Initializes the Stack.
29       procedure Initialize
30          with Global => (Output => State),
31               Post   => Is_Empty;
32
33    private
34       Stack_Size : constant := 100;
35       type    Pointer_Range is range 0 .. Stack_Size;
36       subtype Index_Range   is Pointer_Range range 1 .. Stack_Size;
37       type    Vector        is array(Index_Range) of Integer;
38
39       type Stack_Type is
40          record
41             S : Vector;
42             Pointer : Pointer_Range;
43          end record;
44
45       Initial_Stack : constant Stack_Type :=
46          Stack_Type'(S       => Vector'(others => 0),
47                      Pointer => 0);
48    end Stack;
```

Body in SPARK 2014:

```
1     package body Stack
2        with Refined_State => (State => My_Stack)
3     is
4        My_Stack : Stack_Type;
5
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        --  Note that a rule declaration annotation is included at this
57        --  point in the SPARK 2005 code. The corresponding SPARK 2014
58        --  syntax is TBD.
59     begin
60        My_Stack := Initial_Stack;
61     end Initialize;
62  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

TO-DO SUMMARY

Todo

It is intended to support subtype predicates. Analysis is required to determine if any subset rules need to be applied and also regarding any extra proof rules that might need to be applied.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\declarations-and-types.rst, line 42.)

Todo

RCC comment: This will need to describe any global restrictions on tagged types (if any) and any additional Restrictions that we may feel users need. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\declarations-and-types.rst, line 95.)

Todo

Constructive modular analysis of generics (including prove once, use many times). Will require significant restrictions and extra aspects to implement. To be completed in the Milestone 4 version of this document.

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

Todo

Update mapping specification section to cover all necessary language features. To be completed in the milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 112.)

Todo

We need to increase the number of examples given. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 149.)

Todo

Consider adding a glossary, defining terms such as flow analysis and formal verification.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 183.)

Todo

The pragmas equivalent to the new aspects need to be added to this document.

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

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\lrm\source\introduction.rst, line 402.)

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\lrm\source\introduction.rst, line 447.)

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.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 451.)

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\lrm\source\introduction.rst, line 569.)

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\lrm\source\introduction.rst, line 574.)

Todo

Complete detail on Code Policies. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 624.)

Todo

Complete detail on Ghost Entities. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 658.)

Todo

Add detail on how retrospective analysis will work when we have a mix of SPARK 2014 and non-SPARK 2014. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 725.)

Todo

Complete detail on constructive, generative and retrospective analysis and verification. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\introduction.rst, line 729.)

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\lrm\source\introduction.rst, line 824.)

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\lrm\source\introduction.rst, line 829.)

Todo

Note that the details of false alarm management are still TBD and so there is currently no equivalent of the accept annotation in the SPARK 2005 body. To be completed in the Milestone 3 version of this document.

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

Todo

The SPARK 2014 version of the code is not provided since the restrictions that are to be applied in terms of package visibility are yet to be determined. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 332.)

Todo

Note that the syntax for identifying the main program in SPARK 2014 *is still TBD.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 751.)

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined. Note that this also applies to the use of the Always_Valid annotation.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 799.)

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 824.)

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 912.)

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 941.)

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\lrm\source\mapping-spec.rst, line 1210.)

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 1221.)

Todo

Note that the SPARK 2014 *versions of this example are currently TBD, as the relevant syntax is not yet defined.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\mapping-spec.rst, line 1231.)

Todo

Note that the SPARK 2014 *version of the rule declaration annotation has not yet been defined - note that it may 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\lrm\source\mapping-spec.rst, line 1247.)

Todo

Are there any other language defined attributes which will not be supported? To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\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 3 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\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 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\names-and-expressions.rst, line 126.)

Todo

More details on volatile variables and definition of a complete model. At the very least, if V is a Volatile Input variable should not have the following assertion provable: T1 := V; T2 := V; pragma Assert (T1 = T2); To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 81.)

Todo

Need to describe the conditions under which a volatile variable can be a parameter of a subprogram. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 89.)

Todo

Consider more than just simple Volatile Inputs and Outputs; Latched outputs, In_Out volatiles, etc. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 93.)

Todo

Further semantic detail regarding Volatile state and integrity levels needs to be added, in particular in relation to specifying these properties for variables which are declared directly within the visible part of a package specification. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 328.)

Todo

Provide language definition for Initializes aspect. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 394.)

Todo

Provide language definition for Initial Condition aspect. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 438.)

Todo

The consistency rules will be updated as the models for volatile variables and integrity levels are defined. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 543.)

Todo

Consider whether it should be possible to refine null abstract state onto hidden state. *Rationale: this would allow the modeling of programs that - for example - use caches to improve performance.* To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 547.)

Todo

Consider whether it should be possible to refine abstract onto hidden state without any restrictions, although the refinement would be checked and potential issues flagged up to the user.

Rationale: there are a number of different possible models of mapping abstract to concrete state - especially when volatile state is being used - and it might be useful to provide greater flexibility to the user. In addition, if a facility is provided to allow users to step outside of the language when refining depends, for example, then it may be necessary to relax the abstraction model as well as relaxing the language feature of direct relevance.*

To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 552.)

Todo

Provide language definition for Refined_State aspect. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 589.)

Todo

We need to consider the interactions between package hierarchy and abstract state. Do we need to have rules restricting access between parent and child packages? Can we ensure abstract state encapsulation? To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 596.)

Todo

Provide Verification Rules for Initializes aspect in the presence of state abstraction. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 605.)

Todo

The subject of refined Global, Depends, Pre and Post aspects is still under discussion (and their need questioned) and so the subsections covering these aspects is subject to change. To be resolved and completed by Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 614.)

Todo

The consistency rules will be updated as the model for volatile variables is defined. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 674.)

Todo

If it ends up being possible to refine null abstract state, then refinements of such state could appear in refined globals statements, though they would need to have mode in out. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 678.)

Todo

Provide language definition for Refined_Global aspect. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 696.)

Todo

The consistency rules will be updated as the model for volatile variables is defined. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 759.)

Todo

If it is possible to refine null abstract state, then refinements of such state could appear in refined depends statements, but wouldn't map to anything in the depends relation itself and would need to have mode in/out in the refined depends. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 763.)

Todo

Provide language definition for Refined_Depends aspect. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 782.)

Todo

Provide language definition for Refined_Pre aspect. To be completed in the Milestone 3 version of this document.

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

Todo

Provide language definition for Refined_Post aspect. To be completed in the Milestone 3 version of this document.

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

Todo

refined contract_cases. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 882.)

Todo

The support for type invariants needs to be considered further and will be completed for Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\packages.rst, line 942.)

Todo

Provide detail on Representation Issues. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\representation-issues.rst, line 10.)

Todo

Need to consider further the support for iterators and whether the application of constant iterators could be supported. To be completed in Milestone.4 version of this document.

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

Todo

Provide detail on pragmas Loop_Invariant and Loop_Variant, and attribute Loop_Entry. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\statements.rst, line 124.)

Todo

In the future we may be able to permit access and aliased formal parameter specs. Target: Release 2 of SPARK 2014 language and toolset or later.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 65.)

Todo

What do we do regarding null exclusion parameters? To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 69.)

Todo

What do we do regarding function access results and function null exclusion results? To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 73.)

Todo

Think about Pre'Class and Post'Class. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 87.)

Todo

Add rules relating to volatile state. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 681.)

Todo

Consider whether to capture the rules from SPARK 2005 about flow=auto mode in this document or whether it is purely a tool issue (in SPARK 2005, in flow=auto mode if a subprogram is missing a dependency relation then the flow analysis assumes all outputs of the subprogram are derived from all of its inputs).

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 705.)

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 arise to flow errors. Perhaps we need an aspect to describe the strict view of a parameter if it is different to the specified Ada mode of the formal parameter? To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 884.)

Todo

Consider how implicitly generated proof obligations associated with runtime checks should be viewed in relation to Proof_In. To be addressed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\subprograms.rst, line 969.)

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 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\tasks-and-synchronization.rst, line 11.)

Todo

Add Tasking. Target: release 2 of SPARK 2014 language and toolset.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\tasks-and-synchronization.rst, line 17.)

Todo

Provide detail on Standard Libraries. To be completed in the Milestone 4 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\the-standard-library.rst, line 19.)

Todo

Describe model of renaming for array indexing and slices. To be completed in the Milestone 3 version of this document.

(The *original entry* is located in C:\sparkdev\spark2014_shared\lrm\source\visibility-rules.rst, line 43.)

APPENDIX

C

GNU FREE DOCUMENTATION LICENSE

Version 1.3, 3 November 2008

C.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.

C.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.

C.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.

C.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.

C.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.

C.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'.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.

C.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.