SPARK Examiner with Run-time Checker

Generation of VCs for SPARK Programs

GenVCs
Issue: 8.11
Status: Definitive
2nd February 2009

Originator

SPARK Team

Approver

SPARK Team Line Manager

Copyright

Limited Warranty

Praxis High Integrity Systems Limited save as required by law makes no warranty or representation, either express or implied, with respect to this software, its quality, performance, merchantability or fitness for a purpose. As a result, the licence to use this software is sold ‘as is’ and you, the purchaser, are assuming the entire risk as to its quality and performance.

Praxis High Integrity Systems Limited accepts no liability for direct, indirect, special or consequential damages nor any other legal liability whatsoever and howsoever arising resulting from any defect in the software or its documentation, even if advised of the possibility of such damages. In particular Praxis High Integrity Systems Limited accepts no liability for any programs or data stored or processed using Praxis High Integrity Systems Limited products, including the costs of recovering such programs or data.

Note: The SPARK programming language is not sponsored by or affiliated with SPARC International Inc. and is not based on SPARC™ architecture.

Contents

1 Introduction 5

2 Formal Verification 6
2.1 Overview 6
2.2 Proof contexts 7
2.3 The origin of proof annotations 7
2.4 Generating verification conditions 8
2.5 Loop invariants 11
2.6 The form of verification conditions 11
2.7 An introduction to refinement proofs 13
2.8 Behavioural Subtyping and Extended Tagged Types 17
2.9 Concurrency Issues 18

3 SPARK Proof Contexts 19
3.1 Introduction 19
3.2 The language of proof contexts 19
3.3 Proof declarations 22
3.4 Subprogram constraints 28
3.5 Proof statements 31
3.6 Examples of proof contexts 33
3.7 Numerical quantification 35

4 The FDL Language 38
4.1 Introduction 38
4.2 Reserved identifiers 38
4.3 Declarations 39
4.4 Expressions 45

5 Generation of Verification Conditions 50
5.1 A note on the logic used 53
5.2 Verification conditions for integer division example 53
5.3 Further example: “sum” revisited 54
5.4 The relationship between verification conditions and code 54

6 The Representation of SPARK in FDL 56
6.1 Literals 56
6.2 Identifiers 56
6.3 Types 56
6.4 Expressions 58
6.5 Procedure calls 68
6.6 Refinement integrity VC generation 70
6.7 Simplification of trivially true VCs by the Examiner 76
6.8 Traceability between VCs and their source 76

7 The Proof Process with SPARK 79
7.1 Process 79
7.2 Annotation and proof of spark code 79
7.3 Use of the Examiner and other proof tools 81
7.4 Definition of proof functions 81

7.5 Independent assessment 82

8 Examples 83
8.1 Integer increment 83
8.2 Swap routine 84
8.3 Procedure call 87
8.4 Structured variables 90
8.5 Loop invariants 91
8.6 Quantifiers 93
8.7 Abstract own variable refinement for a Stack 95
8.8 Behavioural subtyping of extended types 103
8.9 Unconstrained formal parameters 107
8.10 Dealing with ports 108
8.11 An embedded controller 114

Document Control and References 123
File under 123
Changes history 123
Changes forecast 124
Document references 124

1 Introduction

This manual explains the process of generating *verification conditions* for SPARK programs together with their format, interpretation and proof. Details of the use of the SPARK Examiner, such as command line switches and error messages, are contained in the manual *"SPARK Examiner User Manual"*. The use of the SPADE Automatic Simplifier and Proof Checker is described separately, in other Praxis High Integrity Systems documents.

The structure of the remainder of this document is as follows.

Chapter 2 introduces the concepts of:

- formal verification,
- proof contexts,
- the form and generation of verification conditions (VCs), and
- refinement proofs.

Chapter 3 explains what the various SPARK proof contexts are used for, as well as defining their syntax and placement in SPARK code.

Chapter 4 describes the extended form of the part of the FDL modelling language used for expressing VCs.

Chapter 5 provides detailed examples of the process employed by the Examiner to generate VCs showing the (partial) correctness, with respect to its formal specification, of a piece of SPARK code.

Chapter 6 defines the mapping of SPARK language constructs into the FDL employed by VC generation. This includes the generation of refinement integrity VCs for reasoning about abstract own variables.

Chapter 7 includes guidance on the process of annotating SPARK code and proving it correct (wrt the annotations).

Chapter 8 gives examples of SPARK code, their specifications and correctness VCs.

Readers new to formal verification may like to start with Chapters 2, 5, 7 and 8, referring to Chapters 3, 4 and 6 for technical details, as required.

Readers using this document for reference should consult Chapters 3, 4 and 6, depending on whether they are looking for material on SPARK proof contexts, FDL, or the conversion of SPARK into FDL.

2 Formal Verification

2.1 Overview

Given a formal specification of the functional requirements of a subprogram, expressed as a *precondition* and a *postcondition*, we can attempt to prove that the subprogram meets its specification; this is called formal verification.

- The *precondition* is a predicate which expresses a constraint on the imported variables of the subprogram (eg that a scalar import is in a certain range, that an array is ordered, or perhaps some more complex properties).

- The *postcondition* is a predicate which expresses a relation between the initial values of imported variables and the final values of exported variables of the subprogram. (Where a variable is both imported and exported, we require some notation to distinguish its initial and final state in the postcondition.)

In software for an embedded system, the imports used in the precondition are typically values read from sensors together with historic values forming part of the state, while the exports appearing in the postcondition are values to be transmitted to the outside world, for example via actuators.

A common, though risky, laxity is to say that a piece of software has been “formally verified” or “proved correct.” This laxity makes such claims meaningless: one must always be careful to state what one has proved about a piece of software, or against which specification it has been verified.

The specification used may itself be expressed either informally, as a natural language document, for instance, or formally, in a suitable notation. In either case, though particularly in the former, care must be taken to ensure that interpretation of the specification does not compromise the proof, by introducing unjustified assumptions for instance.

What is generally understood by proof of correctness (with respect to a specification) is a proof of *partial correctness*: that if the precondition is met on entry to the program and it terminates (without a run-time error occurring during execution), then the postcondition will hold on exit.

Proof of termination in general involves providing suitable metrics for each loop; these are typically integer expressions, which are strictly monotonically increasing (or decreasing) and bounded above (below). The SPARK Examiner does not, at present, provide automatic support for the generation of proof obligations to prove the validity of such metrics.

If we establish both partial correctness (wrt a specification) and termination, we are generally said to have achieved a proof of *total correctness*. Even a proof of total correctness may omit to establish the absence of certain errors which may arise in execution, and which may therefore compromise the validity of a proof if care is not taken. For example, if division is used in the code, we must ensure that the implicit precondition (which is present in most programming languages) for such an operation, namely that the divisor is non-zero (or even, for "real" arithmetic, that the divisor is not too small), is established for all possible executions of the program.

We refer to these potential errors as *run-time errors*; for SPARK code which does not make use of real arithmetic, it is possible to construct proofs of the absence of run-time errors either independently from, or as part of, the proof of partial correctness with respect to a specification. The verification conditions for the proof of absence of run-time errors can be generated automatically by the Examiner — see the separate manual *"Generation of Run-time Checks for SPARK Programs"*.

2.2 Proof contexts

In SPARK, pre- and postconditions are expressed as annotations called *proof contexts*. Unlike the core annotations of SPARK (such as `derives`), proof contexts are optional annotations, and are written in an expression language which is an extension of Ada's. Within subprogram bodies, additional proof contexts are used for:

- code cutpoints (eg loop invariants) via `assert` statements;
- well-formation checks, via `check` statements.

We can also declare proof functions (for use in annotations as mathematical functions which we define by means of a collection of proof rules), proof types, and type assertions. These are described in section 3.3.

Each proof context is associated with a particular location in the code:

- the *precondition* is associated with the `begin` of the procedure (or function) body;
- the *postcondition* of a procedure (or the `return` annotation of a function) is associated with the `end P;` of the subprogram body; and
- each `assert` or `check` statement in the code is located at a point in between two executable statements, in general, ie it is associated with a point on the arc of the flowchart of the subprogram which passes between the two statements it appears between. Each such assertion specifies a relation between program variables which must hold at that precise point, whenever execution reaches it. Assert statements generate a cut point on the flowchart of the subprogram, check statements do not.

2.3 The origin of proof annotations

Ideally, the contents of proof contexts should be derived from a formal specification (e.g. in VDM or Z). Even in such a case, however, the annotations present in an implementation will differ from those in a specification in that they will be expressed in terms of concrete data types (e.g. arrays) rather than abstract ones (e.g. sequences). There will also be notational differences: a Z convention in specifying an operation, for instance, is to use *x'* for the value of *x after* the operation, while in SPARK a tilde (`x~`) is used to decorate the *initial* value of an imported, exported variable (apart from in the precondition, where all values are initial). The specification may be used to suggest proof functions and rule definitions to use.

Where specifications are less rigorous great care will be needed to obtain meaningful proof contexts. If a specification is too weak or ambiguous we may prove the code against it but fail in the process to prove the intended properties of the code. For example, consider the following formally-expressed but ambiguous specification:

Suppose array *B(1..n)* is to be *A(1..n)* sorted into order; let

$$same\_elements(A, B) \equiv \forall i \in 1..n, \exists j \in 1..n \bullet A(i) = B(j)$$

This specification ignores duplication, eg

```
if      A(1)=1,     A(2)=2,     A(3)=1
then    B(1)=1,     B(2)=2,     B(3)=3
```

would satisfy the above *same_elements* relation: the specification is *too weak*. We need the idea of an *exact* permutation to allow for arrays which contain repeated elements.

2.4 Generating verification conditions

To establish the correctness of a fragment of code, we must prove that, if the procedure (or function) is invoked in a state which satisfies the precondition, then the postcondition is satisfied on reaching the exit from the code. (Note that we regard the postcondition of a function subprogram to be that the value returned by the code is equal to that specified in its `return` proof context.) A verification condition (VC) is a formula for the correctness of a particular path between two cutpoints — the precondition, the postcondition and any other `assert` annotations present in the body of the code. To generate the VC associated with a particular path between two assertions, we hoist the assertion step by step from the end of the path up to the top.

There are only two sorts of statement which are relevant to the hoisting process:

- an assignment statement (`... := ...`); and
- a conditional statement (`if/case/exit when`).

All other statements can be modelled via these two forms; for example:

- a procedure call can be regarded as performing a collection of simultaneous assignments to the exported variables of the procedure; and
- a loop can be defined implicitly via recursion; for the loop `L`:

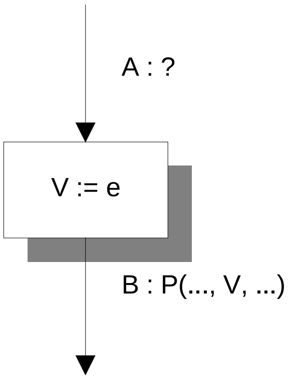
```
          while e loop S; end loop;
```

  we can see, for example, that:

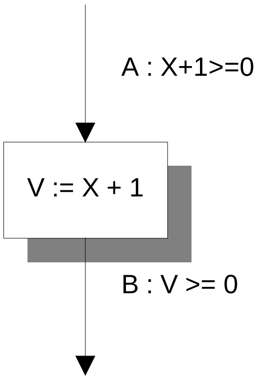
```
L ≡ if e then S; L; end if;
```

2.4.1 Assignment statements

For the annotated code fragment:

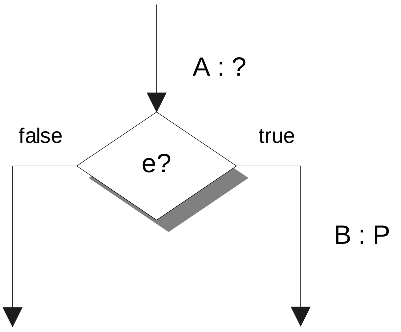


the assertion at point *A* which is sufficient to guarantee that the assertion *P(...,V,...)* (containing instances of variable *V* and other variables) is always true at point *B* is *P(...,e,...)*, ie the assertion with all instances of variable *V* replaced by expression *e*. For example,



2.4.2 Conditional statements

For the annotated code fragment:



the assertion at point *A* which is sufficient to guarantee that assertion *P* is always true at point *B* is $e \Rightarrow P$. Suppose we know that $e \Rightarrow P$ is true at point *A*, for instance, and execution of the code reaches point *B*, so we also know that *e* is true. Then these two formulae *e*, $e \Rightarrow P$ together imply *P*, as we require to establish *P* at point *B*.

2.4.3 The weakest precondition of an assertion

This method allows us to deduce the *weakest precondition* — WP — at a given point for an assertion at another point in the code. We can hoist the assertion successively up the code to reach another assertion in the code; if the assertion at the top of the path implies the weakest precondition at that point of the assertion at the bottom of the path, we have proved the correctness of the path wrt the specification.

We call the formula for the correctness of a path between two assertions a *verification condition* (VC), as noted earlier.

To prove the correctness of the code of an entire procedure, we must prove the correctness of each path between two assertions (and that any subprograms called by the procedure are themselves also correct wrt their specifications). Given the necessary code annotations, the generation of VCs is itself an entirely mechanical process; in general, proving these formulae is not as automatic, however.

It is worth noting that the proof of partial correctness of a subprogram establishes that the exported values returned by it will satisfy the postcondition relation if the subprogram is called with imported values which satisfy its precondition: it says nothing about how the subprogram might behave if it is called with imported values which do *not* satisfy the precondition. Proof that the subprogram is always invoked with imported values which do indeed satisfy its precondition must be performed in the calling environment. This is as it should be: if we write a routine to calculate a square root of a non-negative number, for instance, we must show that wherever it is invoked it is used appropriately, and this can only be done by considering each such invocation in the appropriate context.

2.5 Loop invariants

Assertions planted within the body of a loop need to provide some relation between the variables being modified in the body of the loop. Frequently, they may also need to carry forward other invariant information; for example, if variable X is both imported and exported, but it is not modified within the body of loop *L*, then the loop-invariant for *L* will need to carry forward the assertion `X = X~` (or some other relation) to make the relationship between the initial and final (or so-far) value of X available to subsequent assertions, including the postcondition. Often, one also needs to place bounds on scalars which are modified within loop bodies, and to record how the loop is working incrementally towards satisfying other assertions (such as the postcondition) which follow it. Ideally, loop-invariants should be written at the same time as the loop is coded, since the author of the detailed code should be the person with the best understanding of the behaviour of the loop.

For example, consider a very simple natural number division algorithm, for a processor which doesn't provide a divide instruction; we suppose it is to return a quotient Q and remainder R for M divided by N.

The postcondition which a loop to calculate these will need to establish is:

```
M = Q * N + R and 0 <= R < N
```

(The range constraint on R is necessary to ensure that it is the lowest remainder satisfying the relation.) One way of satisfying the first clause of the postcondition is to note that if we choose `Q = 0` and `R = M`, then the relation is satisfied. This does not in general satisfy the range constraint on R, however. To achieve this, one approach is to look for a succession of values for R, getting smaller with each iteration and ending with the right one. If we decrease R, we will need to increase Q to offset this; yet if Q gets bigger by 1, R needs to decrease by N to preserve the desired relation. This suggests a loop body:

```
Q := Q + 1;   R := R - N;
```

which satisfies the relation `M = Q * N + R` on each successive iteration of the loop, provided it is true on entry to the loop body. Finally, we note that we must exit from this loop body as soon as `R < N` is achieved, and that if we do so, we can ensure `R >= 0` is preserved throughout execution of the loop body, provided it is true on entry to the loop.

2.6 The form of verification conditions

Each verification condition generated by the SPARK Examiner has the following form:

```
H1:  ....
H2:  ....
     ....
   ->
C1:  ....
     ....
```

Each `Hi` is a *hypothesis*.

Each `Ci` is a *conclusion*.

The formula represents:

```
(H1 ∧ H2 ∧ ...) -> (C1 ∧ ...)
```

The hypotheses are derived from the assertion at the head of the path and the path traversal condition and the conclusions are derived from the assertion at the end of the path (suitably transformed).

To prove a VC, we can assume that the hypotheses are true and demonstrate that the conclusions follow from them. We must prove all of the VCs associated with a subprogram to establish the correctness of the code with respect to its specification.

In proving a subprogram P which calls another subprogram Q, in the proof of correctness of P we assume the correctness of the subprogram Q. If we find subsequently that Q does not satisfy its specification, then:

- if we only need to modify the body of Q to correct the error, the proof of the correctness of P is not affected; but

- if we have to modify the specification (ie the precondition and/or the postcondition) of Q, we must re-establish the correctness of P to ensure Q's precondition is met when called and that the assumption of Q's postcondition is sufficient to allow the proof of P's correctness.

2.6.1 Simplification and proof of verification conditions

Verification conditions can be quite large formulae for non-trivial programs; consequently, pencil-and-paper proofs carry a large risk of error; for this reason, automated proof support tools are essential in practice. We can make a number of observations, however:

- The larger and/or more complex the program, the larger the number of VCs and the greater their complexity, in general.

- Procedural decomposition and self-containment helps to reduce the number and size of VCs.

- Many VCs contain a number of trivial components, which can be eliminated by automatic simplification.

- We must strike the right balance in our annotations to achieve VCs that can be proved as simply as possible.

Tool support for simplification and proof are provided by the SPADE Automatic Simplifier, the SPADE Proof Checker and the Proof Obligation Summariser (POGS). The Simplifier, which is distributed with the SPARK Examiner, deals effectively with the simpler verification conditions allowing effort to be concentrated on the more challenging ones. The Proof Checker is an interactive assistant that aids in the construction of proofs, prevents the construction of erroneous proofs and provides an audit trail of the proof work undertaken. Finally, the POGS tool helps with the management of proof work by providing a summary of VCs, indicating their source and current proof status. For further details of the use of these tools, refer to:

- the *“SPADE Automatic Simplifier User Manual”*,
- the SPADE Proof Checker's *“User Manual”* and *“Rules Manual”*, and
- the “*POGS User Manual*”.

2.7 An introduction to refinement proofs

This section provides a basic introduction to refinement proofs involving abstract own variables or private types. It includes only sufficient explanation of refinement to describe the facilities currently supported by the SPARK Examiner. Any reader interested in a more comprehensive treatment of formal refinement should consult an appropriate textbook. As refinement is a more advanced proof concept, new readers may choose to skip this section if they are not interested in proofs involving abstract own variables or the refinement of private types.

2.7.1 SPARK refinement

Excluding refinement proofs involving abstract own variables, SPARK has supported two distinct forms of refinement for many years.

1. Own variable refinement in flow analysis.
2. Using pre- and postconditions, or return annotations, to specify subprogram behaviour.

In both types of refinement[1] the fundamental purpose is to be able to hide or ignore detail that is unnecessary for the analysis being performed.

With own variable refinement in flow analysis an abstract (SPARK) own variable represents a set of concrete (Ada) variables used in the bodies of subprograms.

In the case of pre- and postconditions etc. the refinement arises because a formal specification defines only the end effect of a subprogram and not the detail of any algorithms used to achieve that effect.

The hiding of unnecessary detail in refinement is beneficial because it can lead to:

- simpler specifications that are easier to understand and review;
- reduced maintenance, both through less material to maintain and material being less sensitive to changes in implementation detail;
- decreased coupling between program units, as changes in one unit are less likely to require changes in another.

[1] readers familiar with refinement will note that the first type is data refinement and the second type is algorithm refinement

However, for any refinement to be valid it must be established that the specification, although less detailed, is consistent with the actual implementation behaviour.

The facilities first included in Release 5 of the Examiner provide the means to reason about own variable refinement. That is, to specify the behaviour of subprograms (using pre- and postconditions or return annotations) in terms of abstract SPARK own variables and prove that the specifications are consistent with the Ada implementations of the subprograms.

2.7.2 Abstract and concrete views

Clearly for refinement to exist there must be two separate descriptions or views of behaviour, one of which 'refines' (i.e. in some sense provides more detail than) the other. We use the term **abstract** to refer to the less detailed view and the term **concrete** to refer to the more detailed view.

In the case of information flow analysis using abstract (i.e. non-Ada) own variables:

- the abstract view of information flow is the derives annotation in a package specification;

- whereas the concrete view is the derives annotation in the package body.

With proofs of functional behaviour:

- the abstract view consists of pre- and postconditions (or return annotations);

- the concrete view is the corresponding Ada code.

Note that the concrete view in one refinement may also be the abstract view in another refinement. For example, a package's abstract own variable may be refined by variables that are themselves abstract (from child or embedded packages).

Where private types are used, the abstract view is the external one where the implementation details of the private type are concealed and the refined one, within the body of the package that declares the type, is where the implementation details are known.

2.7.3 Abstraction relations

To show that a refinement is valid it must be established that the abstract and concrete views are consistent. In essence this means that the concrete view can only add detail to the abstract view, everything stated in the abstract view must still be true in the concrete view.

Where private types are processed by a subprogram, the variables manipulated in the abstract and concrete view are the same but the concrete view provides a richer set of operations to describe the subprogram's behaviour.

With own variable refinement[2] the abstract and concrete views actually refer to different variables. So to check that the views are consistent requires a definition of how the abstract and concrete variables relate to each other. We call this relationship the **abstraction relation**.[3]

For example, with flow analysis involving own variable refinement the abstraction relation is provided by the '`--#` **own**' annotation in the package body (which lists the concrete variables that the abstract own variable represents).

2.7.4 Refinement verification conditions

For flow analysis, establishing the validity of a refinement is completely automated by the SPARK Examiner. However, establishing the validity of refinement involving proof annotations requires the generation of verification conditions (VCs). In this section we briefly describe the VCs that are generated when proving the correctness of own variable and private type refinement.

As with flow analysis, involving abstract own variables, proof requires two separate sets of annotations:

- an abstract specification of a subprogram's behaviour in a package specification (that refers to abstract own variables or a type's *private* view);
- a concrete specification in the package body (that does not refer to the abstract own variables given in the package specification).

From these annotations two VCs are generated to establish the validity of the refinement:

- to show the consistency of any abstract and concrete preconditions;
- to show the consistency of any abstract and concrete postconditions or return annotations.

Each of these VCs is briefly described below.

2.7.4.1 Precondition VC

The fundamental condition for an abstract and concrete precondition to be consistent is that whenever the abstract precondition holds then so does the concrete. That is, the concrete precondition is true if the abstract precondition is.[4] The concrete precondition is said to be **weaker** than the abstract precondition.

[2] as it is a type of data refinement

[3] Other terms used for this in the refinement literature include retrieve relation or coupling invariant.

[4] This condition on the relationship between abstract and concrete preconditions is sometimes referred to as the applicability or safety condition.

This condition is necessary as only the abstract precondition of a subprogram is visible to any callers external to the subprogram's package. VCs of such external callers therefore establish the abstract precondition. However, the body of the called subprogram is proved correct with respect to the concrete precondition. So if the abstract precondition does not guarantee that the concrete precondition of the called subprogram is met, then the caller may be proved correct despite the fact that the execution of the called subprogram may fail.

2.7.4.2 Postcondition VC

Fundamentally an abstract and concrete postcondition are consistent if whenever the concrete postcondition is true then so is the abstract.[5] The concrete postcondition is said to be **stronger** than the abstract postcondition.

This condition is necessary as only the abstract postcondition of a subprogram is visible to any callers external to the subprogram's package. VCs of such external callers therefore assume the abstract postcondition. However, the body of the called subprogram is proved correct with respect to the concrete postcondition. So if the concrete postcondition does not guarantee that the abstract precondition of the called subprogram is met, then the caller may be proved correct despite the fact that the execution of the called subprogram does not achieve what the caller assumes.

The VC for the consistency of return annotations is similar. (For details see section 6.6.5.)

2.7.5 Overview of refinement proof facilities

2.7.5.1 Own variable refinement

SPARK and the Examiner support refinement proofs involving an abstract own variable as follows.

1. The abstract own variable is type announced in its package specification.

2. The type used in the announcement must be a unique abstract type (declared using a SPARK annotation for proof types). Such a type is only visible in SPARK proof contexts (ie there is **no** corresponding Ada type).

3. External to the package the Examiner treats the abstract type as if it is a private Ada type (and hence only equality comparisons are possible).

4. Properties of the abstract own variable have to be expressed using proof functions.

5. Public subprograms in the abstract own variable's package, which import and/or export the own variable, have two sets of proof annotations: an abstract set in the package specification and a concrete set in the package body.

[5] This condition on the relationship between abstract and concrete postconditions is sometimes referred to as the correctness or liveness condition.

6 For VC generation *internal* to the package the Examiner has a built-in abstraction relation – namely the abstract type is modelled as a record type with one field for each of the concrete own variables that the abstract variable is refined to.

7 For each subprogram described in item 5 above, two additional VCs are generated by the Examiner to ensure that:

- the abstract precondition is not weaker than the concrete precondition;
- the concrete postcondition is not weaker than the abstract postcondition (or that the return annotations are consistent);

and hence formally prove that the refinement is valid.

2.7.5.2 Private type refinement

SPARK and the Examiner support refinement proofs involving a variable of a private type as follows.

1 Public subprograms in the private type's package, which import and/or export the variables of the private type, have two sets of proof annotations: an abstract set in the package specification and a concrete set in the package body.

2 In the abstract annotation, where the type's implementation is not known, properties of the variable have to be expressed using proof functions.

3 In the concrete annotation, internal to the package, the private type's implementation details are known and the appropriate operators, selectors and functions can be used.

4 For each subprogram described in item 1 above, two additional VCs are generated by the Examiner to ensure that:

- the abstract precondition is not weaker than the concrete precondition;
- the concrete postcondition is not weaker than the abstract postcondition (or that the return annotations are consistent);

and hence formally prove that the refinement is valid.

2.8 Behavioural Subtyping and Extended Tagged Types

SPARK95 permits the use of tagged types and type extensions. Extended types may inherit operations from their ancestor, root types. An important principle in the use of such type extensions is that each extended type should be a *behavioural subtype* of its ancestor type; this is known as the Liskov-Wing substitution principle[6]. Given an extended type *E* which is derived from an ancestor tagged type *T*, then the principle requires that for each primitive operation *P* of *E* and *T*:

pre $(P_T) \Rightarrow pre(P_E)$ *and* $post(P_E) \Rightarrow post(P_T)$

Or, in English, an operation on an extended type must *require less* and *promise more* than the primitive operation it overrides.

The Examiner generates VCs to show that the substitution principle is respected. These are indicated in the VC file with the title "*For checks of subclass inheritance integrity*". An example can be found at Section 8.8. Note that the Examiner does not require the pre and post conditions of the extended operation to be written in a particular or special form (for example, or-ed with its original precondition and and-ed with its original postcondition) provided that the generated VCs can be shown to be true.

2.9 Concurrency Issues

From Release 7.0, the SPARK Examiner provides support for extensions to SPARK which include the "Ravenscar Tasking Profile". These extensions are known as "RavenSPARK" and are described fully in the manual "The SPARK Ravenscar Profile". RavenSPARK allows the construction of programs comprising a fixed set of tasks, protected objects and interrupt handlers. To avoid the possibility of constructing invalid proofs the Examiner prohibits the use of shared variables (i.e. protected variables that may be accessed by more than one program thread) in constraints and proof statements (as described in Sections 3.4 and 3.5).

[6] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994

3 SPARK Proof Contexts

3.1 Introduction

SPARK Proof Contexts are annotations which are not mandatory annotations of the SPARK language but which are extensions to it for formal verification purposes, and are recognised by the Examiner. They are the means by which we introduce the *preconditions*, *postconditions* and *assertions* described earlier into a SPARK program text.

SPARK proof contexts are of three kinds:

Subprogram constraints. These are annotations specifying pre- and postconditions of procedures and preconditions and return values of functions.

Proof statements. These are employed within the body of a subprogram to specify *cut points* (eg loop-invariants) via an `assert` statement, or to introduce other kinds of proof obligations (such as range constraints) via `check` statements.

Proof declarations. The formal specification of a program unit can often conveniently be expressed in terms of *proof functions* (as opposed to SPARK function subprograms) with well-defined properties (which may be specified in the form of re-write rules, for use with the SPADE Proof Checker). The declaration of such functions in a SPARK program text makes it possible to refer to them in subsequent proof contexts. If refinement VCs involving abstract own variables are to be generated, then SPARK *abstract proof types* must be declared to represent the types of the abstract own variables. Finally, a *type assertion* may be used to specify the compiler-dependent base-type of a signed integer type declaration.

The syntax rules of SPARK prescribe the places where proof contexts may appear. The following sections describe the use of the three kinds of proof contexts in more detail.

3.2 The language of proof contexts

The language used to write expressions in subprogram constraints and proof statements is the language of SPARK expressions with the following extensions:

- logical implication;
- structured object update expressions;
- quantified expressions;
- the application of proof functions; and
- decorations to denote the initial values of variables which are both imported and exported.

3.2.1 Logical implication

To assist in writing the Boolean-valued expressions used in proof contexts SPARK is extended, in such contexts, to include the logical implies (`->`) and equivalence (`<->`) operators. For example:

```
X(J) = 3 <-> B.
```

(Note that implication and equivalence bind less tightly than equality.)

Although it is possible to use alternatives to implication and equivalence, their use is recommended since the SPADE Automatic Simplifier and Proof Checker are better able to simplify Boolean expressions of this form.

3.2.2 Structured object update expressions

When one or more components of a structured variable are updated this cannot be represented with a single, simple SPARK expression. For this reason, SPARK in proof contexts is extended to support the description of such updating. An update takes the form of the name of a record or array followed by one or more override expressions in square brackets. The override expressions show which components of the record or array are being overwritten and with which values. Components which do not appear in the list of override expressions retain their original values.

update ::= name **[** override { **;** override } **]**
override ::= expression { , expression } **=>** expression

3.2.2.1 Record updates

Record updates take the form `R[N => C; M => D]` where R is the name of a record, `N` and `M` are the names of fields in R and `C` and `D` are expressions whose types are the same as fields `N` and `M`, respectively of R. The update expressions can be read as *record R with the value of field N replaced by C and the value of field M replaced by D (with other fields remaining unchanged).*

3.2.2.2 Array updates

Array update expressions are similar to those for records except that the field names are replaced by expressions of the index type of the array indicating which elements of the array are being updated. For example: `A[3 => 5; J+1 => 0; J-1 => X + Y]`, which represents array `A` but with elements 3, `J+1` and `J-1` (in turn) overwritten by the values of the expressions shown. Update expressions can be written for multi-dimensional arrays by including the index expressions for each dimension of the array to the left of the arrow symbol. For example `A[1,1 => 0; 2,2 => 1]` to update elements `(1,1)` and `(2,2)`.

3.2.2.3 Updates as names

The sequence of override expressions in the square brackets of an update expression are read from left to right. Update expressions are considered to be names in the grammar so they may be used as a prefix to a further update expression. Thus `V[O;P]` and `V[O][P]` are equivalent in meaning. Similarly an update expression can be indexed into or selected from, in order to read the value of a component of the expression, eg `A[1 => 2](1)` (which has the value 2) or `R[F1 => 2].F1` (which also reduces to 2). When VCs are generated from expressions involving update expressions they are modelled in FDL using `element` and `update` in the case of arrays and `upf_` and `fld_` functions in the case of records.

3.2.3 Quantified Expressions

It is often necessary to express constraints about array data structures and quantified expressions are a convenient way of doing this. Quantified expressions are a kind of predicate and their syntax is:

quantified_expression ::= quantifier_kind defining_identifier **in** *discrete*_subtype_mark
[**range** range] **=> (** predicate **);**

quantifier_kind ::= **for all** | **for some**

This syntax has been chosen to be as similar as possible as that for `for` loops and should thus be familiar to SPARK programmers. The grammar term **identifier** is a user-chosen name for the **quantified variable**; this, as is also the case for `for loop` control variables, must not already be visible. The quantified variable has a scope that extends only to the end of the quantified expression so a quantified variable may be re-used in later expressions. If an explicit **range** is given, then the given subtype_mark must not be Boolean or a subtype of Boolean.

The keywords '**for all**' denote a universal (∀) quantification, ie the whole quantified expression is true if, and only if, the predicate part of the quantifier is true for *every* value of the quantified variable in the defined type and range. The keywords '**for some**' denote an existential (∃) quantification, ie the whole quantified expression is true if, and only if, the predicate part of the quantifier is true for *at least* one value of the quantified variable in the defined type and range. Quantifiers can be nested, ie the predicate part of one quantified expression can include another quantified expression.

As an example consider an array of type `Table` that may, or may not, contain some sought value. We can express the return condition of a function that reports whether or not the sought value is present thus:

```
function ValuePresent(A      : Table;
                      Sought : Integer) return Boolean;
--# return for some M in Index => ( A(M) = Sought );
```

A more complex example is a function that, given that `Sought` *does* appear somewhere in the table, returns the index of first occurrence of it. The precondition of the function ensures that it is only called if `Sought` is definitely present and the postcondition uses the implicit form of function return annotations, together with a quantifier to capture the property that the first occurrence of `Sought` is returned.

```
function FindSought(A      : Table;
                    Sought : Integer) return Index;
--# pre for some M in Index => ( A(M) = Sought );
--# return Z => (( A(Z) = Sought) and
--#    (for all M in Index range 1 .. (Z - 1) =>
--#        (A(M) /= Sought)));
```

3.3 Proof declarations

3.3.1 Implicit declarations

3.3.1.1 Implicitly-declared proof constants

Value of a variable on entry to a subprogram

For each variable V *which is both an import and an export* of a procedure subprogram P, a constant of the same type as V is implicitly declared, to represent the initial value of V within P's proof contexts. The name of this constant is obtained by decorating V with the suffix ~ (tilde). The name of this constant may only occur in the postcondition of P, and in proof contexts placed in the body of P. Notice that unless V is both imported *and* exported it may not be decorated with ~.

In SPARK,a function is not permitted to have explicit side effects on program variables and so it is not regarded as exporting any variables. For this reason, ~ is not generally permitted in function return annotations. An exception to this rule occurs when a function is used to read an *external variable* (see section 8.10) which therefore appears in the function's global annotation. Reading an external variable has an implicit side effect of modifying that variable (see section 3.3.1.3). To allow the initial value of an external variable, prior to this side effect, to be specified, ~ may be used in function annotations only in the case of external variables globally referenced by the function.

The table below shows the meaning of a variable identifier X and its decorated variant X~ in various locations.

Location	X	X~
In a subprogram precondition	Initial value of X	Not permitted
In a procedure postcondition	Final value of X	Initial value of X
In a function return annotation	Final value of X*	Initial value of X*
In a check or assert statement within a subprogram	Value of X at that point in the subprogram's execution	Initial value of X

* where X is an external variable of mode IN globally referenced by the function

Value of a variable on entry to a `for` loop

For each variable V *which is used in the range bounds expressions of a for loop*, a constant of the same type as V is implictly declared to represent the value of V *on entry to the loop.* The name of this constant is obtained by decorating V with the suffix % (percent). The name of this constant may only occur in proof contexts within the loop body.

The need to refer to variable values on entry to for loops arises from the semantic rules of Ada that require the loop exit bound expressions to be evaluated once only, and remain fixed, even if variables appearing in them are modified in the loop body.

Example (see Section 6.4.12 for more information on loop models)**:**

```
type MyInt is range  0 .. 1000;

procedure Double (X : in out MyInt)
--# derives X from X;
--# pre X <= 500;
--# post X = 2 * X~;
is
begin
   for I in MyInt range 1 .. X loop -- X used in exit expression
      X := X + 1;                   -- X changed in loop body
      --# assert X% = X~ and        -- assertion of X initial value
      --#        X = X~ + I;        -- rest of loop invariant
   end loop;
end Double;
```

Note: where variables appear in the loop bound expressions but are not altered in the loop body it will usually only be necessary to add the assertion X% = X to the loop invariant.

3.3.1.2 Implicitly-declared proof functions

For each SPARK function in the source file the Examiner implicitly declares an FDL function. The signature of this implicitly-declared proof function contains all the arguments of the corresponding SPARK function followed by an additional argument for each global variable referenced by the SPARK function. Because the rules of SPARK prevent functions from having side-effects, the SPARK function and its implicitly-declared proof function are in precise correspondence for number and type of data objects read. Note that a SPARK function may not appear in proof contexts: the corresponding implicitly-declared proof function must instead be used in its place. An implicitly-declared proof function is considered to be declared immediately following the corresponding SPARK function's declaration.

3.3.1.3 Implicitly declared proof attributes

External variables are own variables or own variable refinement constituents declared with a *mode* indicating that they import values from or export values to the external environment. Where external variables of mode in are read, there is considered to be an implicit side-effect which transforms the external variable so that the next read operation will return a (possibly) different value. The conceptual model is that the external in variable is a sequence; reading it returns its head and the side-effect returns its tail ready for the next read. Values written to external out variables are considered to be appended to a sequence of values passed out to the environment.

For each external variable of mode in, a proof function attribute `'Tail` is declared. The argument to this function attribute is the external variable itself and its return type is the type of the external variable. If `Sensor` is an external in variable then after the operation `X := Sensor` the following is true:

```
X = Sensor~ and
Sensor = Sensor'Tail (Sensor~);
```

For each external variable of mode out, a proof function attribute `'Append` is declared. The first argument to this function attribute is the external variable itself, the second is the value written and its return type is the type of the external variable. If `Actuator` is an external out variable then after the operation `Actuator := X` the following is true:

```
Actuator = Actuator'Append (Actuator~, X);
```

3.3.2 Explicit declarations

There are four types of proof declaration: proof functions, proof types, type assertions, and object assertions. The following sections describe the use of each.

3.3.2.1 Explicitly-declared proof functions

Proof functions may also be declared explicitly and used after their declaration to simplify the expression of proof contexts within a SPARK program. The behaviour of such proof functions is specified in proof rules which can be used by the Proof Checker - see the *"SPADE Proof Checker Rules Manual"*. A proof function declaration has exactly the same syntax as a SPARK function except that it is introduced with an annotation start symbol `--#`. Proof function declarations can appear anywhere a **later_declarative_item** or **subprogram_declaration** is permitted in SPARK. Proof functions declared either explicitly or implicitly are visible only in proof contexts.

A proof function without arguments is a useful way of defining a constant value which is visible only in a proof context.

It is sometimes useful to create a package especially to hold proof declarations; this package can then be `inherit`ed by packages which wish to refer to items in it but it does not need to be `with`'d since nothing in it is visible in the Ada sense. Note that this approach is not always possible since it may introduce circularity into the inherit chain.

3.3.2.2 Explicitly-declared proof types

SPARK abstract proof type declarations may be included in a package anywhere that basic declarations can be given. The relevant clauses from the SPARK syntax are:

basic_proof_declaration ::= proof_type_declaration | type_assertion | object_assertion **;**
proof_type_declaration ::= **–# type** defining_identifier **is** proof_type_definition**;**
proof_type_definition ::= **abstract**

and **basic_proof_declaration** is included as a clause of the **basic_declarative_item** syntax production.[7] The name of a proof type must be distinct from any other SPARK name in scope in the defining package.

[7] Three productions are included in the SPARK syntax for ease of future enhancements.

An abstract proof type, like a user defined proof function, only exists in SPARK annotations and does not have any corresponding Ada definition. Hence such a type can only be used in:

- an abstract own variable type announcement in the type's package only;
- proof annotations in the package specification in which the type is declared;
- proof annotations in any package that `inherits` the type's defining package.

An abstract proof type cannot be used in the body of the package in which it is declared.

3.3.2.3 Type assertions

A type assertion annotation is also a **basic_proof_declaration**. Its syntax is as follows:

type_assertion ::= base_type_assertion **;**
base_type_assertion ::= **--# assert** identifier'Base **is** type_mark**;**

The **base_type_assertion** is used to specify the predefined base type that is chosen by the compiler to represent a particular signed integer or floating point type. This assists the Examiner when generating verification conditions for Overflow_Check for a type T, so that T'Base'First and T'Base'Last are known [2].

In a base type assertion, the identifier given must denote a signed integer or floating point type whose full declaration precedes the base type assertion in the same declarative region. The type mark given must denote one of the predefined signed integer or floating point types, which are typically specified using the target configuration file mechanism [1].

3.3.2.4 Object assertions

An object assertion annotation is also a **basic_proof_declaration.** Its syntax is as follows:

object_assertion ::= **--# for** simple_name_rep **declare** identifier**;**

The **object_assertion** is used to specify whether replacement rules should be generated for the listed composite constants where the constant appears in VCs generated for subprograms directly or indirectly within the scope of the object_assertion. The identifier may take the values **Rule** or **NoRule** depending on whether replacement rules are required or to be suppressed.

Multiple object_assertions may appear for a given constant, although only one is permitted in a given declarative region. An object_assertion in the declarative region of a subprogram takes precedence followed by the object_assertion in the closest enclosing scope of a subprogram finally any object_assertion in the declarative region of the constant declaration is considered.

Replacement rules will be generated for composite constants based on the command line \rules switch [1] and the object_assertions within the code.

/rules switch value	composite constant rule generation policy
none	No replacement rules are generated for composite constants (default)
lazy	Replacement rules are only generated if there is an object assertion requiring rule generation
keen	Replacement rules are generated unless there is an object assertion suppressing rule generation
all	Replacement rules are generated for all composite constants.

3.3.3 Own variable type announcements

The announcement of both concrete and abstract own variables may contain a type mark. (Remember that concrete own variables are those that have Ada declarations.) We describe the announcement of concrete own variables first.

3.3.3.1 Concrete Own Variables

The type announcement of a concrete own variable allows reference to the type to be made in proof contexts prior to its Ada declaration. If this type mark is a selected component then its name must refer to a type declared in the visible part of an inherited package. If the type mark is a simple name then either it is the name of a type declared in package STANDARD or it is treated as an *announcement* of a type or subtype declaration that must occur within the package. Once a type is announced its simple name may not be used, other than for the declaration of further own variables, until the declaration of a type or subtype with the same simple name. When the own variable is declared in a SPARK variable declaration the type it is declared with must be the same as that announced in the own variable declaration.

The following are all legal examples of typed concrete own variables.

```
package P
--# own State : Integer;   -- simple name of type from package standard
is
end P;

package body P
is
  State : Integer; -- own variable declared with same type as announced
end P;

-------------------------------------

package Q
is
```

```
  type Colour is (Red, Yellow, Blue);
end Q;

-------------------------------------

with Q;
--# inherit Q;
package P
--# own State : Q.Colour;   -- selected component from inherited package
is
  State : Q.Colour;   -- own variable declared in package specification
end P;

-------------------------------------

package P
--# own State : Buffer; -- announcement of type buffer
is
end P;

package body P
is
  type Index is range 1..100;
  type Buffer is array(Index) of Integer; -- declaration of
                                          -- announced type

  State : Buffer; -- declaration of own variable
end P;
```

Note that in the final example, packages external to P do not have visibility of the type definition of the own variable (as the declaration of the announced type `Buffer` is in P's body). This limits the statements about `P.State` that can be made in proof contexts in other packages.

3.3.3.2 Abstract Own Variables

The type announcement of an abstract own variable allows reference to the type to be made in proof contexts (in particular, in the declaration of proof functions). There are a couple of rules for the announcement of abstract own variables:

1. An own variable cannot be announced as being of a proof type declared in another package; this is because the proof type is considered to be a record based on the SPARK own variable refinement which is not visible outside the package in which it occurs. Thus `--# own State : P.T;` is illegal if `P.T` is an abstract proof type declared in package P.

2. Only one own variable may be declared as being of each abstract proof type. Thus '`--# own State1, State2 : StateType;`' is illegal if `StateType` is an abstract proof type. This is because the built-in abstraction relation creates a record from the refinement constituents of the own variable. Since both `State1` and `State2` cannot be refined to the same constituents there would be no valid single definition of the `StateType` record.

Once an abstract proof type is announced its simple name may not be used until the declaration of the type.

Abstract proof types are handled in a similar way to private Ada types. Hence, in SPARK annotations no information about an abstract type is available, other than that two objects of the type can be compared for equality. External to an abstract type's package, its FDL model is simply a 'pending' FDL type (see Section 4.3.1). The FDL model internal to the package is described in Section 6.6.1.

In the previous section we stated that the Examiner regards any own variable lacking a type announcement as being abstract. If an abstract own variable is not type announced then the Examiner creates an FDL declaration for the own variable's type, where the type name is formed by appending '`__type`' to the variable's name. All the features of abstract proof (see Section 6.6) are unaffected by the absence of a user-provided name for the abstract proof type. The absence of such a name, however, does fundamentally limit what can be achieved because it means that it is not possible to define proof functions to describe the behaviour of operations in abstract terms.

3.4 Subprogram constraints

Subprogram constraints are annotations that may be used to introduce specifications (pre-, postconditions and return expressions) of subprograms into SPARK program texts.

3.4.1 Procedure constraints

In the SPARK syntactic categories

- **subprogram_declaration**
- **subprogram_body** and
- **body_stub**,

procedure_annotation may be followed immediately by **procedure_constraints**, where

procedure_constraints ::= [**--# pre** predicate]
[**--# post** predicate]

In the above, **predicate** indicates an extended SPARK expression which is of type `Boolean`. The scope of the **procedure_constraints** of a procedure S is that of the declaration of S.

A procedure can only have constraint predicates on both its declaration and body if it imports or exports (as defined by its flow annotations) an abstract own variable from its *own package* or if it imports or exports a variable or parameter of a private type declared in its *own package*. The constraint predicates on the procedure's declaration provide the abstract (external) view of the procedure and those on its body provide the concrete (internal) view.

If the procedure does not import or export an abstract own variable or a variable of a private type (from its own package) then any constraint predicates *must* be placed with the first appearance of the procedure (ie its declaration). These predicates can only refer to the concrete variables of the package containing the subprogram (and hence the same concrete view is used both externally and internally). However, the predicates may refer to abstract own variables from other (inherited) packages that are imported or exported by the procedure.

There are limitations to what may appear in the predicate expressions associated with pre- and postconditions:

precondition. In a precondition of a subprogram S, every identifier must be either

- the name of an imported variable of S;
- the name of an exported formal parameter of S which is of an unconstrained type (to allow the expression of constraints, such as length, on the actual parameter supplied. See Section 8.9 for an example);
- the name of a predefined function or attribute;
- a type-identifier;
- a constant;
- the name of a quantified bound variable; or
- a proof-function identifier.

Note. For convenience, the Examiner also assumes that a subprogram has the implicit precondition that its imports (formal parameters and concrete global variables) are correctly within their type. For example, if an in parameter X is of type Natural, it is not necessary to provide an explicit precondition that X >= 0. Users should be aware, however, that no corresponding check that the actual parameter is correctly in its type is made at the point of call to the subprogram unless run-time checks are being generated (see the manual "Generation of Run-Time Checks for SPARK Programs").

postcondition. In a postcondition of a subprogram S, every name must be as for the precondition of S as specified above, but any variables which are both imported and exported by S may appear decorated with the tilde (~) suffix as described earlier.

The precondition of a subprogram specifies the properties of the computational state which are required to hold whenever the subprogram is called; the post-condition specifies the required relationship between the initial and final values of the state whenever execution of the subprogram terminates.

For example

```
procedure Inc(X : in out Integer)
--# derives X from X;
--# pre X < Integer'last;
--# post X = X~ + 1;
is
begin
  X := X + 1;
end Inc;
```

If a pre- or postcondition is omitted, either abstract or concrete, the Examiner regards the constraint to be the predicate ‘`true`’ in all generated VCs. Typically this will be of use if a subprogram has no precondition, ie it may always safely be called. An empty postcondition may be of use if an attempt is being made to prove some property of a SPARK program where the values returned by a particular procedure are not of interest, for instance the absence of run-time errors.

The constraints assumed by the Examiner also mean that SPARK code is *always* proved against constraint predicates involving only concrete variables from the package. (As if a subprogram has abstract constraints, and no user supplied concrete constraints, then the Examiner uses concrete constraints of ‘`true`’ in generated VCs.).

3.4.2 Function constraints

In the SPARK syntactic categories

- **subprogram_declaration**
- **subprogram_body**
- **body_stub**

the **function_annotation** may be followed by **function_constraints**, where

function_constraints ::= [**--# pre** predicate]
[**--#** return_expression]
return_expression ::= **return** annotation_expression | **return** simple_name **=>** predicate

The rules for the content, visibility and placement of function preconditions are identical to those for procedures. In place of a procedure’s postcondition, functions have a return expression which may specify the value returned by the function either explicitly or implicitly.

Example 1, an explicit return statement:

```
function Inc(X : Integer) return Integer
--# return X + 1;
is
begin
  return X + 1;
end Inc;
```

Example 2, an implicit return statement:

```
function Max(X, Y : Integer) return Integer
--# return M => (X > Y -> M = X and
--#              Y > X -> M = Y);        -- deliberately incomplete
```

```
is
  Result : Integer;
begin
  if X > Y then
    Result := X;
  else
    Result := Y;
  end if;
  return Result;
end Max;
```

The => symbol can be read as “satisfying” or “such that”. The identifier (here, M) in front of the => symbol has the same type as the result of the function and is used as a local name for the result; it must not be an otherwise visible identifier at the point of the function declaration. The entire return expression in the above example can be read as: *return some M such that if X is greater than Y then M is X and if Y is greater than X then M is Y.*

Note in this example that the return expression annotation is indeterminate in the case where X = Y. We can nevertheless prove that the function meets this partial specification, though ideally the return expression annotation should specify the value returned in such a case also.

The rules for the placement of abstract or concrete return annotations are the same as for procedure postconditions. Similarly, if a function’s precondition or return annotation is omitted, either abstract or concrete, the Examiner regards the constraint to be the predicate ‘true’ in all generated VCs (see also section 6.6.5.). Again this means that SPARK code is *always* proved against constraint predicates involving only concrete variables from the package.

Note that the implicitly-declared proof function (see section 0) associated with an Ada function is declared immediately after that function’s signature and so may be used in its return annotation, thus:

```
function Extract return Integer;
--# global State;
--# return Extract (State);  -- use of implicit proof function
```

This technique can be useful in refinement proofs involving functions. See section 6.6.5.

3.5 Proof statements

Proof statements are transparent to the SPARK flow analysers, but they are used by the verification-condition generator to produce verification conditions and other formulae relating to the validity of a SPARK text.

The syntax category **statement** is extended with the category **proof_statement**.

```
proof_statement   ::=   --# assert predicate |
                        --# check  predicate
```

Every sequence of statements must contain at least one statement which is not a proof statement.

In addition the syntax of loop statements is modified to allow an assertion to appear between **iteration_scheme**, if there is one, and the reserved word `loop`.

Both check and assert statements express constraints that the programmer wishes to show are true. VCs will be generated, by the Examiner, whose proof will confirm this. The expression describing the constraint may refer both to imported and exported variables (with decoration if appropriate) and to the subprogram's local variables. Whether a check or assert is used determines how VCs are produced.

3.5.1 Assert statements

An assert statement represents a cut point in the program control flow graph. Each loop must have an assert statement which cuts the loop and serves as a loop invariant. A loop invariant is a Boolean expression which must be true every time execution of the code of the loop body reaches the point at which it is placed. For example

```
while I <= n loop
  ........
  --# assert I <= n and
  --#        X = X~ and .....;
  ........
end loop;
```

Because an assert statement is a cutpoint, hypotheses from previous check and assert statements are not carried through an assert statement; what is known about the subprogram’s state after an assert statement is only that which is included in it. For example:

```
............
--# check X /= 0;
Y := Z / X;
while Y > 0 loop
--# assert f(X, Y) > 0;
-- from here on X /= 0 cannot be assumed
............
```

It is therefore necessary sometimes to include additional clauses in an assert statement to specify relations between variables which are being maintained by the code if they are needed for some later part of a proof.

As well as their use as loop invariants, assert statements can be useful to break a long sequence of statements into several parts, each of which can be proved separately (though the need to do this may suggest that the code could be better structured, eg through procedural decomposition). Such use may simplify the overall task by reducing the number of VCs and/or making each somewhat easier to prove, since we need only reason about a relatively contained change of state.

3.5.2 Check statements

A check statement is not a cutpoint (unlike assert). Like assert, however, it also results in the generation a proof obligation to show that the associated formula is true whenever execution reaches the point at which it is embedded in the code.

We can use check to prove the absence of certain run-time errors, eg:

```
--# check i in IndType and
--#       a(i) in XType;
x := a(i);

--# check j /= 0
k := i / j;
```

(Note, however, that the SPARK Examiner with Run-time Checker generates VCs for such run-time conditions automatically, so that it is no longer necessary to insert check statements manually for this particular purpose.)

3.6 Examples of proof contexts

3.6.1 Naive integer square root

This routine finds the integer square root of a non-negative number by starting the search at zero and trying successive candidates in turn until the root is found. (This is an extremely inefficient algorithm, particularly for larger N, but it is easy to prove its correctness.)

```
procedure Integer_Square_Root (N   : in     Natural;
                               Root:    out Natural)
--# derives Root from N;
--# pre    N >= 0;
--# post   (Root * Root <= N) and
--#        (N < (Root + 1) * (Root + 1));
is
   R, S, T : Natural;
begin
   R := 0; S := 1; T := 1;
   loop
      --# assert  (T = 2 * R + 1) and
      --#         (S = (R + 1) * (R + 1)) and
      --#         (R * R <= N);
      exit when S > N;
      R := R + 1;
      T := T + 2;
      S := S + T;
   end loop;
   Root := R;
end Integer_Square_Root;
```

3.6.2 A more efficient integer square root

This algorithm expresses exactly the same functionality as the above example, but employs a binary search to find the square root, resulting in a much more efficient algorithm in the general case. It has also been written as a function subprogram, rather than a procedure, to illustrate the use of the return expression annotation. In all other respects, its visible specification is identical.

```
function Integer_Square_Root (N: in Natural) return Natural
--# pre    N >= 0;
--# return Root => (Root * Root <= N) and
--#                (N < (Root + 1) * (Root + 1));
is
   lower, upper, middle: Integer;
begin
   lower := 0;
   upper := N + 1;
   loop
      --# assert  (0 <= lower) and (lower < upper) and
      --#         (upper <= N + 1) and
      --#         (lower * lower <= N) and
      --#         (N < upper * upper);
      exit when lower + 1 = upper;
      middle := (lower + upper) / 2;
      if middle * middle > N then
         upper := middle;
      else
         lower := middle;
      end if;
   end loop;
   return lower;
end Integer_Square_Root;
```

3.6.3 Greatest common divisor

This is an implementation of the Euclidean algorithm to find the greatest common divisor of two non-negative numbers (at least one of which must be positive: this must be the first parameter of the procedure below). It also calculates two (non-unique) multipliers, Y and Z satisfying the relation shown in the postcondition below. Note that we use a proof function, gcd, as a shorthand for the mathematical function whose result we wish to calculate: its definition may be provided to the SPARK proof tools in the form of one or more proof rules (which are not given here).

```
--# function gcd(A, B: Natural) return Natural;
```

```
procedure Greatest_Common_Divisor
                  (M, N: in Integer; X, Y, Z: out Integer)
--# derives X, Y, Z from M, N;
--# pre    (m > 0) and (n >= 0);
--# post   (x = gcd(m, n)) and (x = y * m + z * n);
is
   a1, a2, b1, b2, c, d, q, r, t: Integer;
begin
   a1 := 0; a2 := 1; b1 := 1; b2 := 0; c := m;  d :=
   n;
   loop
      --# assert  (c > 0) and (d >= 0) and
      --#         (gcd(c, d) = gcd(m, n)) and
      --#         (a1 * m + b1 * n = d) and
      --#         (a2 * m + b2 * n = c);
      exit when d = 0;
      q := c / d; r := c - q * d;
      a2 := a2 - q * a1; b2 := b2 - q * b1;
      c := d; d := r;
      t := a1; a1 := a2; a2 := t;
      t := b1; b1 := b2; b2 := t;
   end loop;
   x := c;  y := a2;  z := b2;
end Greatest_Common_Divisor;
```

3.7 Numerical quantification

As described in section 3.2.3, the Examiner supports the explicit use of quantification over predicates, eg to allow the statement of loop invariants:

```
i := index'first;
loop
  a(i) := i;
  exit when i = index'last;
  --# assert
  --#   index'first <= i and i < index'last and
  --#   (for all k in Integer range index'first..i =>
  --#      (a(k) = k));
  i := i + 1;
end loop;
```

(where k is an identifier not already in scope at the point that the quantification is introduced, in order to avoid variable capture).

However, the Examiner does not support explicit quantification over (non-predicate) expressions, eg to express the sum of a series of numbers. A numerical quantification example follows of what to do in such situations. (The format of proof rules, included in this example as an illustration, are covered in detail in the *“SPADE Proof Checker ——Rules Manual”*.)

Consider taking the sum of a collection of numbers: we must use proof function notation in this case. Suppose a program is to calculate

$$S = \sum_{i=m}^{n} a_i$$

To specify its postcondition, we can declare

```
--# function sum (A: arrtype; L, U: index) return Integer;
```

intending `sum(x, y, z)` to be the sum of the elements of array `x` in the index range `y..z` inclusive. We may then write:

```
--# post s = sum (a, m, n);
```

We must take care to address the boundary cases, however: what is `sum(a, 3, 2)`, for instance? Either the definition of `sum`, as provided in the proof rules to be used with the Proof Checker, must define this, or we must take care to ensure the question cannot arise. Two alternative means of addressing this are proposed below.

Solution 1: Make `sum` a total function:

```
sum(1):   sum(A, I, J)   may_be_replaced_by 0 if [I>J].
sum(2):   sum(A, I, I)   may_be_replaced_by element(A, [I]).
sum(3):   sum(A, I, J)   may_be_replaced_by
                            sum(A, I, J-1) + element(A,[J])
                               if [I<J].
```

We do not then need a precondition to the subprogram, and it is clear what result should be returned for empty index ranges.

Solution 2: Preclude empty index ranges:

```
--# pre m <= n;
```

In this case, the definition of `sum` can instead be:

```
sum(1):   sum(A, I, I)   may_be_replaced_by element(A, [I]).
sum(2):   sum(A, I, J)   may_be_replaced_by
                            sum(A, I, J-1) + element(A, [J])
                               if [I<J].
```

A possible implementation (with `sum` partial) is:

```
begin  -- SumTotal
  k = m;
  s := a(k);
  loop
    --# ???;
    exit when k = n;
    k := k + 1;
    s := s + a(k);
  end loop;
end SumTotal;
```

What can we propose for a loop invariant for the above, given the proposed placement?

From the precondition, `m <= n`, so on entry to the loop we can see that `m <= k <= n` (since `k = m` initially). Does the loop preserve this? Yes, because if we get back to the invariant, we did not exit (so `k <> n` held, giving `k < n`) and we have incremented `k` by 1.

The other component that we are calculating is `sum`: on reaching the invariant initially on entry to the loop, `s = a(m) = sum(a, m, m) = sum(a, m, k).`

4 The FDL Language

4.1 Introduction

When VCs are produced by the Examiner they are expressed in a form which can be manipulated by the Simplifier and Proof Checker; the notation employed is an extended form of the expression language of FDL. FDL is the modelling language of the SPADE tools. Its use in expressing VCs generated from SPARK programs is limited since we are only concerned with FDL *expressions* together with such declarations as are required to denote the type of these expressions; we are not concerned here with FDL *statements*. Within a SPARK source text, proof contexts are expressed in an extended version of the SPARK language itself.

This chapter describes only those elements of FDL used to denote VCs of SPARK programs. Readers interested a full description of FDL are referred to the SPADE FDL manual. New readers may choose to skim over this chapter since it is possible to gain an intuitive understanding of those elements of FDL needed to understand the Examiner's output from later chapters.

FDL bears some resemblance to a programming language; however, it is important to appreciate that FDL is essentially a modelling language used to describe the semantics of constructs present in imperative programming languages, to support the rigorous analysis of algorithms written in these languages. FDL's descriptions of objects, and of operations upon them, are to be interpreted in a mathematical sense. With regard to declarations, the FDL type-identifier `integer`, for example, refers to the set of mathematical integers. In the same way, the operations such as addition and subtraction on these numbers have their mathematical significance: they do not give rise to overflows. Similarly, an FDL declaration of a function introduces a pure function, ie a mapping from one set to another whose evaluation cannot produce any side-effects. We describe below those constructs present in FDL which are needed to express VCs generated by the SPARK Examiner.

4.2 Reserved identifiers

The following is a list of the reserved identifiers of FDL (excluding those that are also reserved words of SPARK). Identifiers in this list may not be used as identifiers in a SPARK program if it is intended to use the VC generation facilities of the Examiner on it. By default the Examiner will report such use as syntax errors; however, it is possible to suppress this behaviour if only the non-proof facilities of the Examiner are to be used — see the *“Examiner User Manual”* for details.

```
are_interchangeable  as                   assume
const                div                  element
finish               first                for_all
for_some             goal                 last
may_be_deduced       may_be_deduced_from  may_be_replaced_by
nonfirst             nonlast              not_in
odd                  pending              pred
proof                real                 requires
save                 sequence             set
sqr                  start                strict_subset_of
subset_of            succ                 update
var                  where
```

In addition identifiers beginning with the character sequences `fld_` or `upf_` are also regarded a predefined FDL identifiers.

4.3 Declarations

Generally it should not be necessary for the user to create FDL declarations, since the Examiner automatically creates a file of those declarations used in the VCs that it generates. The syntax and explanation that follows is intended as a reference to enable these declarations to be read if so desired. A fuller explanation of all the FDL types available can be found in the SPADE FDL Manual.

4.3.1 Types

A type characterizes a set of values which variables and constants of that type may assume, and the set of basic operations which may be performed on them. Type is an attribute possessed by every value and every variable.

FDL has three standard types: the real-type, the Boolean-type and the integer-type. Other types can be created through type declarations.

An FDL type-declaration associates an identifier with a type; the identifier may then be used to denote this type. A type-declaration may also give a definition of the type (in which case the type is said to be *concrete*); alternatively, the type-definition may be the reserved word `pending` (in which case the type is said to be *abstract*).

Abstract FDL types are used solely for abstract own variables, irrespective of whether or not SPARK abstract proof types are used to type announce them (see Section 3.3).

Concrete FDL types are classified into simple types and structured types. A simple type is a type whose values have no components, and form an ordered set. A structured type is a composite type, characterized by the type(s) of its components and the structuring method.

> type-declaration ::= **type** type-identifier **=** type-definition | **pending**
> type-identifier ::= identifier
> type-definition ::= simple-type-definition | structured-type-definition

4.3.1.1 Simple types

A simple type is a type whose values have no components, and form an ordered set. The simple types comprise the ordinal types and the real type. The ordinal types comprise the enumerated types (which are defined by enumeration of their values, and include the standard Boolean type) and the integer type.

4.3.1.2 Enumerated types

An enumerated-type-definition defines an ordered set of values by enumeration of the identifiers which denote these values.

> enumerated-type-definition ::= **(** identifier-list **)**
> identifier-list ::= identifier { **,** identifier }

The ordering of the values is determined by the sequence in which their identifiers are enumerated, ie if `x` precedes `y` then `x` is less than `y`.

The occurrence of an identifier in the identifier-list of an enumerated type constitutes its declaration as a constant.

An identifier cannot appear in more than one enumerated-type-definition.

Example:

```
type Colour = (Red, Orange, Yellow, Green, Blue, Indigo, Violet)
```

4.3.1.3 The Boolean type

The type-identifier `boolean` denotes the Boolean type. This is a standard enumerated type, whose values are the truth values denoted by the constant-identifiers `false` and `true`.

The Boolean-type is equivalent to the type-declaration

```
type boolean = (false, true)
```

together with additional logical operations such as `and` and `or`.

4.3.1.4 The integer-type

The type identifier `integer` denotes the integer type. This is a standard ordinal type, whose values are the positive and negative integers, in the mathematical sense.

4.3.1.5 The real type

The type-identifier `real` denotes the real type. This is a standard simple type, whose values are the real numbers, in the mathematical sense.

4.3.2 Structured types

A structured type is a composite type, characterized by the type(s) of its components and the structuring method. The structured types comprise the array types and record types.

structured-type-definition ::= array-type-definition | record-type-definition

There are no restrictions on the types of the components of structured types, which can even be abstract.

4.3.2.1 Array types

An array type is a structured type consisting of a fixed number of components (called elements) which are all of the same type. An element of an array is designated by one or more index values belonging to declared enumerated or subrange types.

array-type-definition ::= **array [** index-type-list **] of** element-type-identifier

index-type-list ::= index-type-identifier { **,** index-type-identifier }

index-type-identifier ::= type-identifier

element-type-identifier ::= type-identifier

Examples:

```
type Bit          = (ZeroBit, OneBit)
type BitPosition  = 1 .. 8
type Byte         = array[BitPosition] of Bit
type Address      = 1 .. 48
type Memory       = array[Address] of Byte
type Matrix       = array[Row, Column] of Real
```

4.3.2.2 Record types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a *field*, its type and an identifier which denotes it.

record-type-definition ::= **record** field-list **end**

field-list ::= record-section { **;** record-section }

record-section ::= identifier-list **:** type-identifier

Examples:

```
type DayNumber  = 1 .. 31;
type MonthName  = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
type YearNumber = 0 .. 2000;
type Date       = record
                    Day   : DayNumber;
                    Month : MonthName;
                    Year  : YearNumber
                  end;
```

4.3.3 Constants

A constant-declaration introduces an identifier to denote a value, which can be of any type, concrete or abstract. The declaration consists of an identifier denoting the new constant, followed by the identifier representing its type, followed by a constant-definition. If the constant is declared as being of integer type then the constant-definition can be a signed-integer, a character-string (which could be used to denote a very large integer), or the reserved word `pending`. If a constant is not of integer type then its definition must be either a character-string or the word `pending`.

constant-declaration ::= **const** identifier **:** type-identifier **=** constant-definition
constant-definition ::= signed-integer | character-string | **pending**
signed-integer ::= [sign] unsigned-integer
sign = **+** | **-**

Examples:

```
const Answer: Integer = 42
const Million: Integer = '1000000'
const MatrixOrder: Integer = pending
const pi: Real = '3.141593'
const Avogadro: Real = '6.02295e23'
const EmptyStack: Stack = pending
```

The constants produced by the Examiner are always declared as `pending`; values for such constants are substituted using suitable, automatically-generated proof rules. Proof rule generation for composite constants is governed by the rule generation policy requested on the Examiner command line and object assertions within the source code.

4.3.4 Variables

A variable is an entity to which a value may be attributed.

A variable-declaration consists of a list of identifiers denoting new variables, followed by their type.

variable-declaration ::= **var** identifier-list **:** type-identifier

Examples:

```
var x, y, z: Real
var p, q: Boolean
```

4.3.5 Functions

FDL functions are pure mathematical functions, as noted earlier.

function-declaration ::= **function** identifier [**(** parameter-type-list **)**] **:** type-identifier
parameter-type-list ::= parameter-type { **,** parameter-type }
parameter-type ::= type-identifier

Examples:

```
type Stack = pending;
const EmptyStack : Stack = pending;
function push(Stack, Integer) : Stack;
function pop(Stack) : Stack;
function top(Stack) : Integer;
function is_empty(Stack) : Boolean;
function is_full(Stack) : Boolean;
```

4.3.6 Standard functions

4.3.6.1 Arithmetic functions

`abs(x)` For an expression x of integer type or real type, `abs(x)` denotes the absolute value of x. The type of the value of the function is the same as the type of its parameter, x.

`sqr(x)` For an expression x of integer type or real type, `sqr(x)` denotes `x * x`. The type of the value of the function is the same as the type of its parameter, x.

4.3.6.2 Ordinal functions

`pred(x)` For an expression x of ordinal type T, whose value is greater than the least member of T, `pred(x)` denotes the value immediately preceding that of x (in the enumeration of T). If x is the least member of T then the value of `pred(x)` is undefined.

`succ(x)` For an expression x of ordinal type T, whose value is less than the greatest member of T, `succ(x)` denotes the value immediately succeeding that of x (in the enumeration of T). If x is the greatest member of T then the value of `succ(x)` is undefined.

4.3.6.3 Boolean functions

odd(x) For an expression x of integer type, `odd(x)` is equivalent to the expression `abs(x) mod 2 = 1`.

4.3.6.4 Functions for manipulating arrays

`element` Let A be an expression, of an array type declared in the form `array[ i1, i2, ... , ip ] of T`. Then the function-designator `element(A, [k1, k2, ... , kp])`, in which `k1, k2, ... , kp` are expressions whose values are assignment-compatible with the types `i1, i2, ... , ip` respectively, has the value of the array element `A(k1, k2, ... , kp)`.

`update` Let A be an expression, of an array type declared in the form `array[ i1, i2, ... , ip ] of T`. Then the function-designator `update(A, [k1, k2, ... , kp], x)`, in which `k1, k2, ... , kp` are expressions whose values are assignment-compatible with the types `i1, i2, ... , ip` respectively, and `x` is an expression whose value is assignment-compatible with type T, has the value of the array A after the assignment of the value of `x` to its element `A(k1, k2, ... , kp)`.

Examples:

```
type RowIndex = 1 .. 10;
ColIndex = 1 .. 20;
Matrix   = array[RowIndex, ColIndex] of Real;

var  M: Matrix; r: real;
............................
r := element(M, [3, 4]);
............................
M := update(M, [3, 4], r);
............................
M := update(M, [3, 4], element(M, [4, 3]) + r);
```

4.3.6.5 Functions for manipulating records

The declaration of a record-type R, with fields `f1, f2, ... , fp` of types `t1, t2, ... , tp` respectively, implicitly introduces `p` pairs of functions:

```
fld_f1(r) : t1       upf_f1(r, t1) : r
fld_f2(r) : t2       upf_f2(r, t2) : r
..............       .................
fld_fp(r) : tp       upf_fp(r, tp) : r
```

For an expression `x` whose value is a record of type R, `fld_fi(x)` denotes the value of the field `fi` of the record, and it is of type `ti`. Also, if `y` is an expression whose value is assignment-compatible with type `ti`, then `upf_fi(x, y)` denotes the value of `x`, after the assignment of the value of `y` to its field `fi`.

4.3.6.6 Functions for assembling structured values

For every array or record type T, FDL implicitly declares a function `mk__t` which can be used to assemble objects from a collection of values. The parameters of `mk__t` are a series of assignments to the elements of an object of type `t` with an optional default value appearing first.

For records the function takes the following form:

mk__record ::= **mk__**record_type_identifier **(** argument_list **)**
argument_list ::= association { **,** association }
association ::= field_identifier **:=** expression
record_type_identifier ::= identifier
field_identifier ::= identifier

For arrays the form is slightly more complicated because defaults values are permitted as are assignments to multiple array elements:

mk__array ::= **mk__**array_type_identifier **(** argument_list **)**
argument_list ::= expression | expression associations | associations
associations ::= association { **,** association }
association ::= index { **&** index } := expression
index ::= [index_expression { **,** index_expression }]
index_expression ::= expression [**..** expression]
array_type_identifier ::= identifier

Examples

```
mk__complex(real := 1, imag := 0);
mk_table(0, [1] := 1, [2] := 99);
mk_matrix(0, [1,1] := 1, [2,2] := 2, [3,3] := 1);
mk_lookup([1]  := 0, [2] & [3] := 1, [4] & [5] := 2);
mk_atype([1..10] := 1, [11..20] := 2);
```

4.4 Expressions

Expressions are constructs denoting rules of computation for obtaining values of variables and constants, and generating new values by the application of operators and functions. FDL expressions consist of operands, operators, function-designators, set- and sequence-constructors, and quantifiers. Not all the elements of FDL expressions described in this section are used when VCs are produced from SPARK texts, however; they are included here because they may be introduced when VCs are manipulated with the Proof Checker.

expression ::= disjunction [(**->** | **<->**) disjunction] | quantified-expression
disjunction ::= conjunction { **or** conjunction }
conjunction ::= negation { **and** negation }
negation ::= [**not**] relation
relation ::= sum [relational-operator sum]
sum ::= [sign] term { adding-operator term }
term ::= factor { multiplying-operator factor }
factor ::= primary | primary ****** primary
primary ::= variable-identifier | unsigned-integer | constant-identifier |
function-designator | **(** expression **)**
variable-identifier ::= identifier
constant-identifier ::= identifier

Any factor whose value is of type S, where S is a subrange of a type T, is treated as being of type T.

Examples:

Primaries:	`x`
	`15`
	`abs(x + y)`
	`(x + y + z)`

Terms:	`x * y`
	`i div (1 - i)`

Simple expressions:	`-x`
	`i * j + 1`

Relations:	`Answer = 42`
	`p < q`
	`(i < j) = (j < k)`

Negations:	`not p`

Conjunctions:	`x <= y and y <= z`
	`p and not q`
	`p = q and r`

Disjunctions:	`p or (x > y)`
Expressions:	`(a * a > b * b)  ->  (abs(a) > abs(b))`

4.4.1 Operators

The standard FDL operators are:

multiplying-operator ::= ***** | / | **div** | **mod**
adding-operator ::= **+** | **-**
relational-operator ::= **=** | **<>** | **<** | **>** | **<=** | **>=**

In addition, the SPARK exponentiation operator ** is supported by the SPADE Automatic Simplifier and SPADE Proof Checker as though it were an extension of FDL's expression syntax; it has a precedence which is compatible with that used in Ada, relative to the other FDL operator precedences. Both tools contain rules which express the semantics of exponentiation for a non-negative power of a real or integer argument in Ada.

The types of the operands used with the standard FDL operators must be the same as the types specified in the subsections below, or subranges of those types.

4.4.1.1 Arithmetic operators

The arithmetic operators comprise the unary-minus operator “-” and the binary operators listed in the table below.

operator	operation	types of operands	type of result
`*`	multiplication	integer or real	integer if both operands are of type integer; otherwise real
`/`	division	integer or real	real
`div`	division with truncation	integer	integer
`mod`	modulus	integer	integer
`+`	addition	integer or real	integer if both operands are of type integer; otherwise real
`-`	subtraction	integer or real	integer if both operands are of type integer; otherwise real

The unary-minus operator takes an integer or real operand; it produces a result of the same type as its operand.

The `div` operator truncates towards zero, so that `-(a div b) = -a div b`. Also, `a div -b = -a div b`. The FDL `mod` operator is defined by `a mod b = a - ((a div b) * b)`. The right operands of `div` and `mod` must be non-zero.

4.4.1.2 Boolean operators

In the list of operators given below, the `not` operator is a unary operator and the others are binary operators. They all take Boolean operands and produce a Boolean result.

operator	operation
`not`	logical negation
`and`	logical conjunction
`or`	logical disjunction
`->`	logical implication
`<->`	logical equivalence

4.4.1.3 Relational operators

The relational operators compare their operands and produce a result of type Boolean.

operator	relations	types of operands
= <>	equality and inequality	any type
< > <= >=	ordering	any simple type
-> <->	logical implication and equivalence	Boolean

The operators <= and >= stand for less than or equal, and greater than or equal, respectively. In comparing values of set type they denote set inclusion in FDL, though this overloading is eliminated within the SPADE proof tools.

4.4.1.4 Operator precedence

In the table below, the operators are listed in order of decreasing precedence (but with operators at any level being of equal precedence). Note that this ordering can be inferred from the syntax of expressions.

```
- (unary minus)
*    /    div   mod
+    -
=  <> <  >  <= >=
not
and
or
-> <->
```

In expressions, sequences of operators of the same precedence are applied from left to right.

Since functions cannot have side-effects, the order of evaluation of operands in an expression need not be defined.

4.4.2 Function-designators

A function-designator specifies the evaluation of a function. It consists of the identifier designating the function, and a list of actual parameters. The parameters are expressions, whose values must be assignment-compatible with the corresponding parameter-types in the function.

function-designator ::= function-identifier [**(** actual-parameter-list **)**]
function-identifier ::= identifier
actual-parameter-list ::= actual-parameter { **,** actual-parameter }
actual-parameter ::= expression

Examples: `abs(x)` `gcd(m, 42)` `sin(p + q)`

4.4.3 Quantifiers

A quantified-expression consists of the quantifier, the quantified variable (with its associated range of values) and the Boolean-valued expression that is being quantified over.

quantified-expression ::= quantification-kind **(** quantification-generator **,** boolean-expression **)**
quantification-kind ::= **for_all** | **for_some**
quantification-generator ::= identifier-list **:** type-identifier
boolean-expression ::= expression

Examples:

```
for_all (k : indexrange,
          (1 <= k) and (k < r) ->
            (element(a, [k]) <= element(a, [k + 1])))

for_some (m : mapping,
           for_all (k : indexrange,
               element(a, [element(m, [k])])
                   = element(b, [k])))
```

5 Generation of Verification Conditions

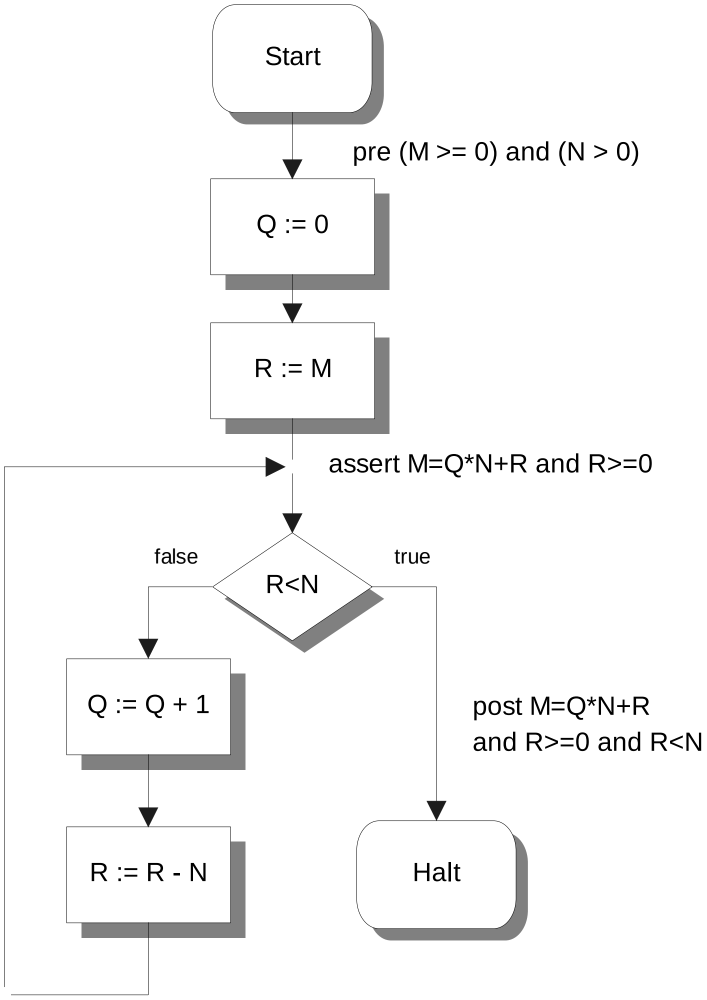
This chapter provides a more detailed example of the process employed by the Examiner to generate verification conditions by hoisting assertions up through the code to generate weakest preconditions, as briefly outlined earlier.

Consider the following SPARK code fragment which represents an integer division algorithm.

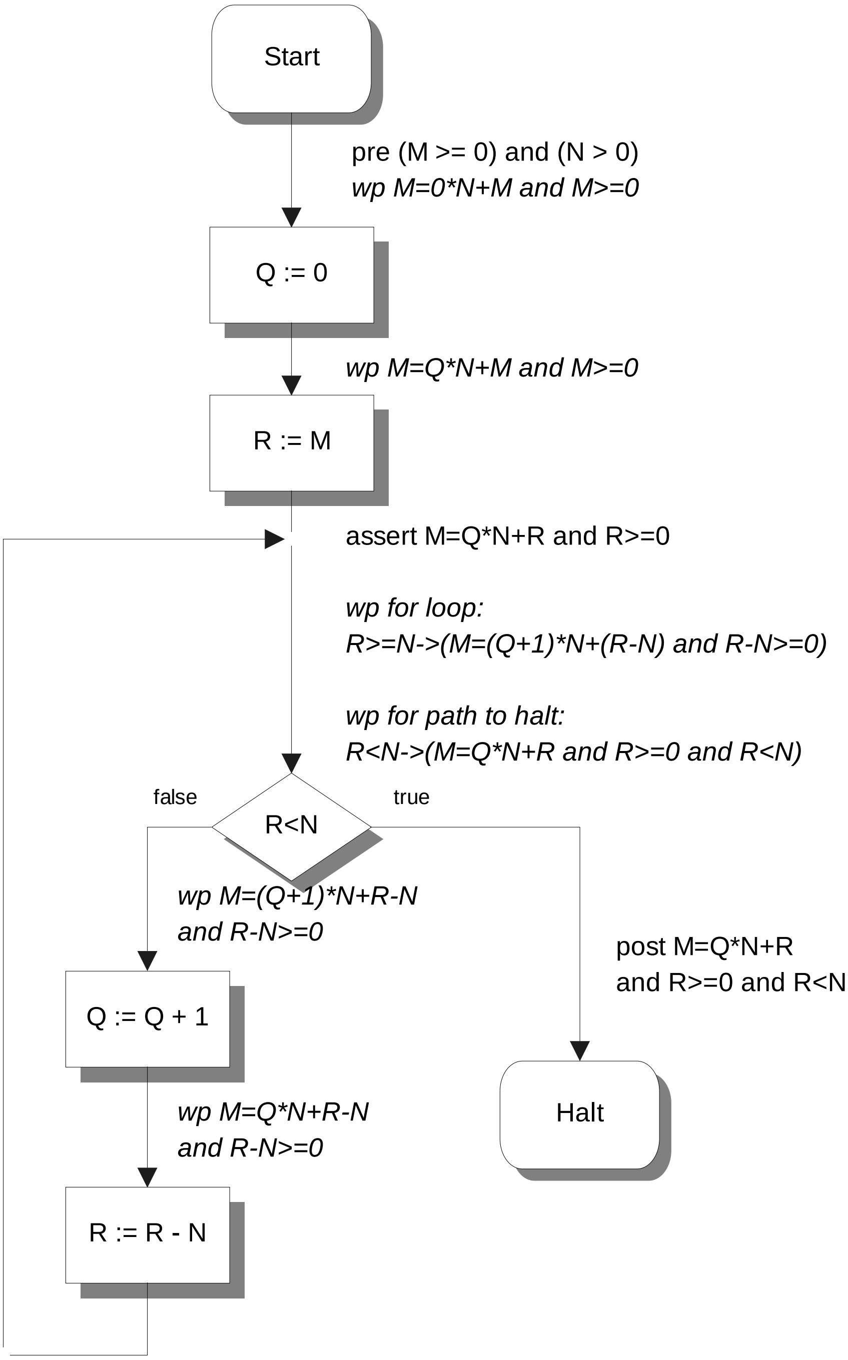
```
--# pre    (M >= 0) and (N > 0);
--# post   (M = Q * N + R) and (R >= 0) and (R < N);

Q := 0;
R := M;
loop
   --# assert (M = Q * N + R) and (R >= 0);
   exit when R < N;
   Q := Q + 1;
   R := R - N;
end loop;
```

This can be represented by the following annotated flow chart.



Weakest preconditions can be generated by hoisting the post condition through the code as described earlier; the weakest precondition of each reachable assertion at each point on the flow chart is shown on the second annotated diagram.


Start
pre (M >= 0) and (N > 0)
wp M=0*N+M and M>=0
Q := 0
wp M=Q*N+M and M>=0
R := M
assert M=Q*N+R and R>=0
wp for loop:
R>=N->(M=(Q+1)*N+(R-N) and R-N>=0)
wp for path to halt:
R<N->(M=Q*N+R and R>=0 and R<N)
false
true
R<N
wp M=(Q+1)*N+R-N
and R-N>=0
post M=Q*N+R
and R>=0 and R<N
Q := Q + 1
Halt
wp M=Q*N+R-N
and R-N>=0
R := R - N

5.1 A note on the logic used

For a path from assertion *A* to another assertion *B*, with traversal condition *T* (possibly rewritten by the assignment statements along the path), the verification condition is:

$$A \Rightarrow (T \Rightarrow B')$$

(where *B'* is *B* rewritten by the assignment statements of that path). $T \Rightarrow B'$ is the weakest precondition of assertion *B* for this path, with *B'* being the assertion *B* after transformation by the assignments and procedure calls which occur along the path.

This formula may be written as:

$$(A \wedge T) \Rightarrow B'.$$

We can rewrite it in this way, because

$$A \Rightarrow (T \Rightarrow B') \equiv (A \wedge T) \Rightarrow B'$$

is a tautology.

5.2 Verification conditions for integer division example

Given the above annotated example, the VCs generated are:

For path from start to loop-invariant:

Weakest precondition: `0 * N + M >= 0.`

VC: `M >= 0 ⇒ M = 0 * N + M and M >= 0.`

For path from loop-invariant back to itself:

Weakest precondition:

```
    R >= N ⇒ (M = (Q + 1) * N + R - N and R - N >= 0).
```

```
VC: (M = Q * N + R and R >= 0 and R >= N) ⇒
       (M = (Q + 1) * N + R - N and R - N >= 0)
```

For path from loop-invariant to end of subprogram:

Weakest precondition:

```
    R < N ⇒ (M = Q * N + R and R >= 0 and R < N).
```

```
VC: (M = Q * N + R and R >= 0 and R < N) ⇒
       (M = Q * N + R and R >= 0 and R < N)
```

5.3 Further example: “sum” revisited

Consider again the annotated code fragment shown below:

```
--# pre   m <= n;
--# post  s = sum(a, m, n);
begin  -- SumTotal;
  k = m;
  s := a(k);
  loop
    --# assert(s = sum(a, m, k)) and
    --#       (m <= k) and (k <= n);
    exit when k = n;
    k := k + 1;
    s := s + a(k);
  end loop;
end SumTotal;
```

The VCs generated from this are as follows:

For path from start to loop-invariant:

```
m <= n ⇒ a(m) = sum(a, m, m).
```

For path from loop-invariant back to itself:

```
(s = sum(a, m, k) ∧ m <= k ∧ k <= n ∧ k ≠ n) ⇒
     (s + a(k+1) = sum(a, m, k+1) ∧ m <= k+1 ∧ k+1 <= n)
```

For path from loop-invariant to end of subprogram:

```
(s = sum(a, m, k) ∧ m <= k ∧ k <= n ∧ k = n) ⇒
     s = sum(a, m, n).
```

5.4 The relationship between verification conditions and code

This section records some common observations about the correspondence between code annotated for proof and the VCs generated from it.

If we prove a VC by establishing a contradiction amongst the hypotheses of the VC, this VC corresponds to a non-executable path. Such a path can be non-executable either implicitly or explicitly:

- If the path traversal condition (without the assertion at the top of the path) contains a contradiction, the path is explicitly non-executable; for example:

  ```
  if x = 1 then y := y + 1; end if;
  if x = 2 then y := y - 1; end if;
  ```

Syntactically, there is a path for $x=1 \wedge x=2$ but *semantically*, it is non-executable.

- If the contradiction is between the traversal condition and the assertion at the top of the path, the path is implicitly non-executable:

```
--# assert x >= 1;
if x < 0 then y := y - 1; end if;
```

If a conclusion *C* in a VC is identical to a hypothesis *H*, this often indicates an expression which is invariant for this path (for example, around a loop).

For a loop with incrementing counter (`i`, say), we must often establish, to prove the subsequent VCs, that `i` attains its maximum value on exit from the body of the loop.

A VC which does not appear to be provable suggests:

- a fault on the corresponding path in the code; or
- that the code annotations are themselves either false or insufficiently strong; or
- possibly both of the above!

Experimenting with counter-examples to invalidate the VC can reveal which case applies.

6 The Representation of SPARK in FDL

When producing VCs the Examiner maps SPARK language constructs (including proof contexts) into their FDL equivalents, which then appear in the VC output files. This Chapter describes the details of this mapping from SPARK to FDL.

6.1 Literals

All literals are converted by removing any underscores within them, converting to base 10 and removing any exponent.

Example:

```
16#FFFF_FFFF#      becomes 4294967295
1e3                becomes 1000
```

Real literals are represented as a rational pair; if the divisor is 1 it is omitted.

Examples:

```
0.5        becomes (1/2)
1.0e3      becomes 1000
```

6.2 Identifiers

SPARK identifiers are represented in FDL by mapping them to lower case and replacing the dot in any package prefix by a double underscore.

Examples:

```
Speed                    becomes       speed
Air_Speed                becomes       air_speed
Doppler.GroundSpeed      becomes       doppler__groundspeed
```

(*Doppler* is the name of the package from which GroundSpeed is being selected)

(Suitable FDL declarations for these newly-created identifiers are generated and placed in the FDL file automatically by the Examiner.)

6.3 Types

The FDL type system is simpler than that of SPARK. The FDL equivalent of SPARK's types are as follows:

Integer types. All integer types from the program being modelled are treated as being of FDL's integer type which behave as integers in the mathematical sense. Checking that the values of these integers remain within the bounds of their original SPARK types can be achieved by proof of suitable check statements inserted either manually or by the SPARK Examiner with Run-time Checker.

Real types. All SPARK fixed and floating point types are modelled as FDL real numbers; again these behave as the mathematical real numbers.

Enumeration types. SPARK's enumeration types are directly replaced by their FDL equivalents. Suitable declarations are placed in FDL file. Note that enumeration literals declared in other packages will each be prefixed by their package name and two underscores.

Array types. SPARK arrays are represented by directly-equivalent FDL arrays. Suitable FDL declarations are automatically produced. Access to and updating of elements of array objects is modelled by predefined FDL functions.

Record types. SPARK records are represented by directly-equivalent FDL records. Suitable FDL declarations are automatically produced. Access to and updating of fields of record objects is modelled by predefined FDL functions.

Tagged records. The modelling of inherited fields in extended record types is modelled explicitly using a field called “inherit” in the FDL records used. For example, given the Ada declarations:

```
type PT is tagged record       -- in package P
   X, Y : Integer;
end record;

type QT is new PT with record -- in package Q
   B : Boolean;
end record;

type RT is new QT with record -- in package R
   F : Float;
   I : Integer;
   N : Natural;
end record;
```

the FDL equivalent is:

```
type p__pt = record
   x, y : integer
end record;

type q__qt = record
   inherit : p__pt;
   b : boolean
end record;
```

```
type rt = record
  inherit : q__qt;
   f : float;
   i : integer;
   n : integer
end record;
```

6.4 Expressions

6.4.1 Arrays access and update

The declaration of array type A introduces element and update functions as described in the chapter on FDL. Thus for instance, the SPARK statement

```
M(I, J) := M(J, K) + 2;
```

is modelled by

```
m := update(m, [i, j], element(m, [j, k]) + 2);
```

It is worth emphasising the different way in which multi-dimensional arrays and arrays of arrays are handled using this notation.

Multi-dimensional arrays: `Arr(I, J)` becomes `element(arr, [i, j])`.

Array of arrays: `Arr(I)(J)` becomes `element(element(arr, [i]),[j])` which can be read as *element j of the array which is element i of array arr*.

(Although SPARK supports the declaration of multi-dimensional arrays and aggregates of them it is recommended that arrays of arrays are used in preference to them at present where proof is to be attempted.)

6.4.2 Records field selection

The declaration of a record type R introduces field access and update functions as described in the chapter on FDL. The following example illustrates their use to model SPARK record operations.

Example: Given the SPARK declaration of the record-type `Date`

```
type DayNumber is 1 .. 31;
type MonthName is
   (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
type YearNumber is 0 .. 2000;
type Date is record
               Day   : DayNumber;
               Month : MonthName;
               Year  : YearNumber
             end record;
```

the following FDL functions are implicitly declared

```
function fld_day(date) : daynumber;
function upf_day(date, daynumber) : date;

function fld_month(date) : monthname;
function upf_month(date, monthname) : date;

function fld_year(date) : yearnumber;
function upf_year(date, yearnumber) : date;
```

The extraction and assignment of field values, for records of type `Date`, is illustrated by the following:

```
mon := fld_month(dat);          {SPARK equivalent: Mon := Dat.Month;}
dat := upf_day(dat, 14);        {SPARK equivalent: Dat.Day := 14; }
dat := upf_month(dat, jul);     {SPARK equivalent: Dat.Month := Jul; }
dat := upf_year(dat, 1789);     {SPARK equivalent: Dat.Year := 1789; }
```

6.4.3 Array aggregates

A SPARK array aggregate, using either named or positional association, is represented in FDL by its `mk__` function which is implicitly declared for each array type. The arguments of this function are read from left to right with any default value arising from a SPARK `others =>` clause appearing as first argument, followed by a sequence of value assignments.

6.4.3.1 Named association

When named association is used, the associations given in the SPARK aggregate are directly mapped on to the arguments of the `mk__` function. For example, given

```
type Index is 1..4;
type A     is array(Index) of Integer;


A'(1=>1,2=>2,3=>3,4=>4)  becomes mk__a([1]:=1, [2]:=2, [3]:=3, [4]:=4)
A'(1 => 1, others => 0)  becomes mk__a(0, [1]:=1)
A'(others => 0)          becomes mk__a(0)
A'(Index => 0)           becomes mk__a([index__first .. (index__last)]
                                   := 0)
A'(Index range 1..4 => 0 becomes mk__a([1 .. 4] := 0)
A'(1..4 => 0)            becomes mk__a([1 .. 4] := 0)
A'(1 | 2 | 3 | 4 => 0)   becomes mk__a([1] & [2] & [3] & [4] := 0)
```

6.4.3.2 Positional association

When positional association is used, the arguments needed by the `mk__` function are constructed as expressions based on the bounds of the index type and the element's position number. Given the same declarations as above

```
A'(1, 2, 3, 4) becomes     mk__a([index__first]:=1,
                                   [index__first+1]:=2,
                                      [index__first+2]:=3,
                                         [index__first+3]:=4)
```

which the SPADE Automatic Simplifier will reduce, given the value of `index__first` which it reads from the rule-file generated by the Examiner, to

```
mk__a([1]:=1, [2]:=2, [3]:=3, [4]:=4)
```

If the `Index` type were instead

```
type Index is (red, yellow, green, blue)
```

then

```
A'(1, 2, 3, 4) would become
     mk__a([index__first]:=1,
              [succ(index__first)]:=2,
                 [succ(succ(index__first))]:=3,
                    [succ(succ(succ(index__first)))]:=4)
```

which the SPADE Automatic Simplifier will reduce, using its knowledge of the enumeration type and the `succ` function and again using the value of `index__first` from the rule-file generated by the Examiner, to

```
mk__a([red]:=1, [yellow]:=2, [green]:=3, [blue]:=4)
```

6.4.4 Record aggregates

Record aggregates behave very much in the same way as the arrays described above but are simpler because they may not contain others clauses, ranges or choices: each field simply appears once either by name or by position. An FDL `mk__` function is implicitly declared for each record type.

For example, given

```
type R is record
   B1,
   B2  : Boolean;
   I   : Integer;
end record;
```

the record aggregates

```
R'(B1 => True, B2 => False, I  => 0);
R'(True, False,  0);
```

both become

```
mk__r(b1 := true, b2 := false, i := 0)
```

Where aggregates and extension aggregates involving extended records are used, the FDL model will make use of the "inherit" fields described in Section 6.3

6.4.5 Attributes

SPARK attributes are handled in one of three ways, considered below. The Examiner also generates proof rules to express properties of attributes and these are also described below.

6.4.5.1 Attributes 'Succ and 'Pred

`'succ` and `'pred` of numeric types are modelled using addition and subtraction. For enumerated types, the attribute is mapped directly on to its FDL equivalent functions `succ` and `pred`. Given the declarations

```
type Int is range 1..10;
type Colour is (Red, Yellow, Green);
```

the attribute expressions

```
Int'Succ(X);
Colour'Succ(Y);
```

are respectively represented by

```
x + 1
succ(y)
```

6.4.5.2 Constant attributes

Attributes which return a constant value such as `'first` are replaced with an FDL constant for which a suitable declaration is placed in the *.FDL* file. The constant name is created by joining the attribute prefix and attribute identifier together with a double underbar character. Thus `Colour'First` becomes `colour__first`. Where the value of this constant can be determined by the Examiner a suitable replacement rule is written to the *.RLS* file.

6.4.5.3 Function attributes

Function attributes such as `'pos` are replaced with an FDL function for which a suitable declaration is placed in the *.FDL* file. The function name is created by joining the attribute prefix and attribute identifier together with a double underbar character. Thus `Colour'Pos(Red)` becomes `colour__pos(red)`. The same representation is used for both Ada function attributes and the SPARK proof attributes `'Append` and `'Tail` (see Section 3.3.1.3).

6.4.5.4 Enumeration proof rules

For enumeration types the Examiner automatically generates a number of proof rules that are written to the *.RLS* file. These rules embody properties of various attributes, such as the result of combining `'succ` or `'pred` with `'pos` or `'val`.

6.4.6 Operators

Most SPARK operators have an FDL equivalent; however, there are some differences including differences in operator precedence. These are illustrated below. Given `X` and `Y` of a numeric type, `I` and `J` of an integer type and `A` and `B` of Boolean type,

abs X + Y	*becomes*	abs(x) + y	(abs is a function in FDL not an operator)
I / J	*becomes*	i div j	(remains x / y for real types)
I **rem** J	*becomes*	i - i div j * j	(FDL does not have the rem operator)
X /= Y	*becomes*	x <> y	
not A = B	*becomes*	(not a) = b	(not binds more loosely in FDL than in SPARK)
A **xor** B	*becomes*	(a or b) and (not (a and b))	(FDL does not have the xor operator)

SPARK 95 modular types become normal integers in FDL. To capture the modular behaviour of their operators, a suitable mod operator is appended to all expressions involving such types. For example, if X and Y are variables of a modular type T then the SPARK expression `X + Y` becomes `(x + y) mod t__modulus` in FDL.

Modular types also allow use of bitwise binary operators (`and`, `or` and `xor`); these are modelled using FDL functions `bit__and, bit__or and bit__xor`. These general functions are used for all modular types. Unary bitwise “`not X`” for a modular type T is modelled as `T’Last - X`.

Logical operators between one-dimensional arrays of Booleans also have an FDL equivalent. For a given SPARK Boolean array type, each of the logical operators (`not`, `and`, `or` and `xor`) is modelled as an FDL function that takes argument(s) of the SPARK type and returns a Boolean array of the same type. The names of the FDL functions are formed by appending ‘`__ op`’ (where `op` is one of: `not`, `and`, `or`, `xor`) to the array type name. The Examiner also automatically generates a proof rule for each function, which defines the value of elements of the array returned by the function. These proof rules are written to the *.RLS* file.

Given two one-dimensional Boolean arrays A and B of SPARK type `BoolArrayT` (whose index type's first and last elements are `l` and `u`), the following table lists the FDL translation of the various array operators, together with the associated proof rules.

SPARK	FDL Translation	Proof Rule for FDL Function

not A	`boolarrayt__not(a)`	`element(boolarrayt__not(X), [I])`
		`may_be_replaced_by`
		`not element(X, [I])`
		`if [l <= I, I <= u]`

A **and** B	`boolarrayt__and(a,b)`	`element(boolarrayt__and(X,Y), [I])`
		`may_be_replaced_by`
		`element(X, [I]) and element(Y, [I])`
		`if [l <= I, I <= u]`

A **or** B	`boolarrayt__or(a,b)`	`element(boolarrayt__or(X,Y), [I])`
		`may_be_replaced_by`
		`element(X, [I]) or element(Y, [I])`
		`if [l <= I, I <= u]`

A **xor** B	`boolarrayt__xor(a,b)`	`element(boolarrayt__xor(X,Y), [I])`
		`may_be_replaced_by`
		`(element(X, [I]) or element(Y, [I]))`
		`and (not (element(X, [I]) and`
		`element(Y, [I])))`
		`if [l <= I, I <= u]`

Care must be taken when using the binary Boolean-array and modular bitwise operators (`and`, `or` and `xor`) in SPARK proof contexts. This is because the binding powers of the operators, and the overloading of '`and`' and '`or`' with logical operators on predicates, can result in the Examiner wrongly treating operators as operators on predicates, depending on the use of parentheses. For example, if A, B and C are all Boolean arrays of the same type, then a postcondition of:

```
--# post A = B and C;
```

will be parsed by the Examiner as:

```
--# post (A = B) and C;
```

resulting in a semantic error (as there is no operator '`and`' between predicates and Boolean arrays that returns a predicate!). To specify the intended postcondition, that A is the pairwise-conjunction of the two arrays B and C, the user must write:

```
    --# post A = (B and C);
```

Note that the above problem does not arise with the unary operator not, as it binds very tightly and so there is no problem of ambiguity.

6.4.7 Short-circuit forms

The SPARK short-circuit forms of conjunction and disjunction and then and or else are replaced in FDL with their logical equivalents and and or. (For proof of absence of run-time errors, the short-circuit forms are not directly equivalent because the second half of the expression may not be evaluated; however, the Examiner with Run-time Checker inserts the correct check statements in these cases) .

6.4.8 Membership tests

As in Ada, a membership test, in and not in, is either a *range* membership test, expressed in terms of the lower and upper bounds of a range, or it is a *subtype* membership test, with a typemark. In the verification conditions produced by the Examiner, a membership test is always expressed in terms of two inequalities, relating the value of the simple expression of the membership test to the least and greatest values of the range (or type). For example

X **in** Itype;	*becomes*	(x >= itype__first) and (x <= itype__last)
X **in** 3..7;	*becomes*	(x >= 3) and (x <= 7)
Y **in** Etype;	*becomes*	(y >= etype__first) and (y <= etype__last)
Y **in** red..yellow;	*becomes*	(y >= red) and (y <= yellow)
X **not in** Itype;	*becomes*	(x < itype__first) or (x > itype__last)
X **not in** 3..7;	*becomes*	(x < 3) or (x > 7)
Y **not in** Etype;	*becomes*	(y < etype__first) or (y > etype__last)
Y **not in** red..yellow;	*becomes*	(y < red) or (y > yellow)

6.4.9 Function calls

SPARK functions may reference (but not update) global data whereas FDL functions are pure functions in the mathematical sense. To enable the modelling of SPARK functions with globals in FDL the Examiner implicitly declares an FDL proof function for each SPARK function in the source text. The signature of this implicitly declared proof function contains all the arguments of its matching SPARK function followed by an additional argument for each global referenced by the SPARK function. Because the rules of SPARK prevent functions having side-effects the SPARK function and the implicitly declared proof function are in precise correspondence for number and type of data objects read and returned.

For example, given the SPARK function declaration

```
function Add(X : Integer) return Integer;
--# global Increment;
```

the call `Add(X);` becomes `Add(x, increment);`.

6.4.10 Type conversions

The facility for declaring different integer types and subtypes in Ada (and SPARK) provides protection through two quite different forms of type-checking. Firstly, we have rules of type consistency such as the rule that, in an assignment statement, the *type* of the expression must be the same as the (base) type of the assigned variable. Compliance with such *static-semantic* rules is checked by the Examiner, as it parses a text. (Such checks are also performed, prior to code generation, by compilers).

The other important aspect of the integer types and subtypes is that they impose *range constraints* on the values of variables — which values could in other respects be considered as all belonging to the same predefined integer type.

A type conversion — between integer types or subtypes — whilst it may be important to meeting static-semantic rules, is of no relevance to a partial proof of correctness of an algorithm where we assume that the program terminates normally and therefore that all values have remained within their permitted bounds. Where we wish to show that this is indeed the case, we must take into account the constraint introduced by the type conversion; the proof obligations associated with it are automatically inserted by the SPARK Examiner with Run-time Checker.

Type conversions between different real types, or between real types and integer types and subtypes, are more complex. At present, the Examiner ignores the semantic effects of type conversions between different real types. For conversion from a real type to an integer type the Examiner inserts an application of the proof function `round__(real) : integer;` this records the fact that such a conversion takes place, but it should be noted that it does not capture the full semantics of the conversion for the particular real types involved.

6.4.11 Quantified expressions

SPARK quantified expressions (see Section 3.2.3) are simply translated to the corresponding FDL quantified expressions (see Section 4.4.3). However, FDL quantified expressions do not include range constraints on the type of the quantified variable, so any SPARK range constraints are converted into predicates as follows.

A SPARK quantifier of the form	is translated into FDL as

`for some x in atype`
`  range l..u => (E)`
	`for_some (x_ : atype,`
	` l <= x_ and x_ <= u and E_)`†

A SPARK quantifier of the form	is translated into FDL as

`for all x in atype`
`  range l..u => (E)`
	`for_all (x_ : atype,`
	` (l <= x_ and x_ <= u) -> E_)`†

† *Where E_ is E with all occurences of x replaced by x_*

Note that the names of the FDL quantified variables end with an underscore: '_'. This ensures that the quantified variable names are distinct from all Ada names and so avoids the possibility of any name clashes occurring (when SPARK proof annotations are hoisted through code to generate VCs).

6.4.12 Loop structures

For semantic analysis, the for-loop and while-loop iteration schemes are treated as follows.

A while-loop of the form	is treated as

`while E`
`loop`
`  S;`
`end loop;`
	`loop`
	`  exit when not E;`
	`  S;`
	`end loop;`

A for-loop of the form	is treated as

`for I in atype range L..U`
`loop`
`  S;`
`end loop;`
	`if L <= U then`
	`  I := L;`
	`  loop`
	`    S;`
	`    exit when I = U`[8]`;`
	`    I := I + 1;` *-- or 'succ for enumerations*
	`  end loop;`
	`end if;`

[8] The expression U here is the exit expression but with each variable replaced by the value of that variable on entry to the loop. Alterations to variables in the loop body do not affect the value of the loop exit expression. The value of each such variable on entry to the loop can be referred using the % suffix, see Section 3.3.1.1

A for-loop of the form
`for I in atype`
`loop`
`  S;`
`end loop;`
	is treated as
	`I := L;`
	`loop`
	`  S;`
	`  exit when I = U;`
	`  I := I + 1;` *-- or ‘succ for enumerations*
	`end loop;`
	because SPARK does not allow a type to be empty.

A for-loop of the form
`for I in reverse atype`
`   range L..U`
`loop`
`  S;`
`end loop;`
	is treated as
	`if L <= U then`
	`  I := U;`
	`  loop`
	`    S;`
	`    exit when I = L;`
	`    I := I - 1;` *-- or ‘pred for enumerations*
	`  end loop;`
	`end if;`

A for-loop of the form
`for I in Boolean`
`loop`
`  S;`
`end loop;`
	is treated as
	`I := False;`
	`loop`
	`  S;`
	`  exit when I;` *-- that is, I = True*
	`  I := not I;`
	`end loop;`
	because SPARK does not allow ordering of Booleans

A plain loop of the form
`loop`
`  S;`
`end loop;`
	is treated as
	`loop`
	`  S;`
	`  exit when False;`
	`end loop; ;`
	to provide a *syntactic* exit point for flow analysis purposes

Thus, viewing the above models it is apparent that two of the four for-loop constructs have a syntactically-possible path through them (when `L > U`) in which the loop-body is not executed (and similarly, though more obviously, for the `while`-loop). Such paths are reflected in verification conditions for code containing such loop constructs.

6.5 Procedure calls

When generating VCs, a call to a procedure is modelled as

1 a check statement to show that the pre-condition of the called procedure is satisfied by the actual parameters used and the current values of imported global variables;

2 a series of assignments to each of the exports from the procedure; and

3 an assumption (which appears in the hypotheses of VCs for paths which traverse the procedure call) that the procedure's post condition holds for the actual parameters returned and the values of exported global variables.

The Examiner automatically creates new unique identifiers to represent the return value of each export and it is these identifiers which are substituted into the post condition expression to generate the assumption described in point 3 above. The names are generated by appending a double underscore character and a number to the identifier concerned.

For example the following code shows a simple procedure which doubles its integer parameter. This procedure is used to define another procedure, which multiplies its parameter by four.

```
procedure Double(X : in out Integer)
--# derives X from X;
--# post X = X~ * 2;
is
begin
  X := X * 2;
end Double;

procedure Quad(X : in out Integer)
--# derives X from X;
--# post X = X~ * 4;
is
begin
  Double(X);
  Double(X);
end Quad;
```

The VCs generated for procedure Quad are as follows:

```
For path(s) from start to finish:

procedure_quad_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    x__1 = x * 2 .
H5:    x__2 = x__1 * 2 .
        ->
C1:    x__2 = x * 4 .
```

The VC shows the procedure export variables generated by the Examiner to model the effect of calls to procedure `Double: x__1` is the value of X returned by the first call to `Double` and `x__2` that by the second call. The hypotheses of the third VC are generated by the substitution of these variables into the post-condition of `Double` and the conclusion to be proved is the post-condition of `Quad`. The hypotheses also include information about the imports to `Quad` deduced from the Ada type system; in this case that X is in the permitted range of `integer`. Note that additional VCs would be generated if procedure `Double` had any precondition that needed to be checked prior to the call.

6.6 Refinement integrity VC generation

This section gives a comprehensive description of refinement VCs. It defines the:

1. built-in refinement abstraction relation;
2. generation of implicit proof functions for Ada functions;
3. precondition refinement integrity VC;
4. postcondition refinement integrity VC;
5. return annotation refinement integrity VC;
6. nested refinement resulting from embedded and/or child packages.

Unless otherwise explicitly stated, the contents of this section apply equally to SPARK 83 and SPARK 95.

6.6.1 Built-in refinement abstraction relation

An abstraction relation specifies the formal relationship between abstract and concrete variables. With an own variable of an abstract proof type the Examiner provides a fixed abstraction relation, for refinement VCs internal to the package. The abstract proof type is modelled in FDL as a record, with one field for each of the variables that the abstract own variable is refined by.

So given a package specification that includes:

```
package P
--# own X : XType;
is
--# type XType is abstract;
    ...
```

and a corresponding own variable annotation in the package body of:

```
--# own X is X1, X2, ... Xn;
```

the Examiner generates the following FDL declaration:

```
type xtype = record
    x1 : x1type;
    x2 : x2type;
    ...
    xn : xntype
  end;
```

For each refinement VC the Examiner includes hypotheses to make the formal link between abstract and concrete variables. These hypotheses are equality predicates written in terms of the concrete variables and take the following form:

```
x1~ = fld_x1(x~) .      (<-> is used in place of = if the fields are Boolean)
x1 = fld_x1(x) .
...
```

6.6.2 Implicit proof functions for Ada functions

The refinement of Ada functions raises a complication when generating the corresponding implicit proof functions. Namely that the implicit proof function for an Ada function (say `P.Fn)` can have a different type in the package specification and body. For example, in P's specification `Fn`'s global list may include an abstract variable A, whereas in the body its global list may include a concrete variable `C1`. Hence the type of the own variable argument, to the corresponding implicit proof function, is different in each case. In fact even the number of arguments could change (eg if `Fn` in its implementation reads more than one of the concrete variables that A is refined by).

Furthermore, in the body of a package both the abstract and concrete forms of an Ada function's implicit proof function may be in scope at the same time. For example, a procedure may use respectively the abstract and concrete versions of an Ada function in the procedure's abstract and concrete preconditions. Consequently, the weaken precondition refinement VC for the procedure needs to refer to both abstract and concrete forms of the implicit proof function.

The solution to this is to distinguish the abstract proof function in the package body by always using the package name as a prefix. Hence, the abstract implicit proof function of a package P's Ada function `Fn`, **always** appears in VCs as '`p__fn`' (even in VCs internal to the package P). Conversely, the concrete implicit proof function always appears in VCs as just '`fn`'.

6.6.3 Precondition refinement integrity VC

A precondition refinement VC is generated for every subprogram that imports or exports an abstract own variable. The VC is simply that the abstract precondition implies the concrete precondition, with the additional hypotheses from the abstraction relation for the imported variables. So the precondition refinement VC takes the following form:

```
H1:  abstract precondition .
H?:  imported abstraction relation hypotheses .
 ->
C1:  concrete precondition .
```

(Note that for clarity we have omitted the standard 'variables in type' hypotheses.)

6.6.4 Postcondition refinement integrity VC

A postcondition refinement VC is generated for every procedure that imports or exports an abstract own variable. The VC is basically that the concrete postcondition implies the abstract postcondition. However, there are additional hypotheses for:

- the abstract and concrete preconditions;
- the abstraction relations for the imported and exported variables.

So the postcondition refinement VC takes the following form:

```
H1:  abstract precondition .
H?:  concrete precondition .
H?:  imported abstraction relation hypotheses .
H?:  exported abstraction relation hypotheses .
H?:  concrete postcondition .
 ->
C1:  abstract postcondition .
```

(Again for clarity we have omitted the standard ‘variables in type’ hypotheses.)

6.6.5 Return annotation refinement integrity VC

A return annotation refinement VC is generated for every function that imports an abstract own variable. Such VCs are similar to postcondition refinement VCs, but are different for the following reasons:

- functions have no exports so there are no abstraction relation hypotheses for exported variables;
- both abstract and concrete forms of an implicit proof function return a variable of the same type (unlike abstract and concrete forms of procedures that export variables of different types);
- return annotations can be either explicit or implicit (or indeed omitted) so an appropriate predicate (loosely equivalent to a procedure's postcondition) has to be constructed, depending on the form of annotation.

The basic form of a return annotation refinement VC is that whatever constraints are placed on the function’s return value by the concrete annotation, they must be sufficient to establish any constraints placed on the return value by the abstract annotation. So the return annotation refinement VC takes the following form:

```
H1:  abstract precondition .
H?:  concrete precondition .
H?:  imported abstraction relation hypotheses .
H?:  concrete return annotation predicate .
 ->
C1:  abstract return annotation predicate .
```

(Again for clarity we have omitted the standard ‘variables in type’ hypotheses.)

It remains to define what the return annotation predicates are in the VC. For both the abstract and concrete return annotations there are three cases, depending on the form of the annotation.

1 If an annotation is omitted then the return annotation predicate is '`true`'.

2 If an annotation is an explicit return annotation of the form:

```
--# return Exp;
```

then the return annotation predicate is '`return = Exp`', where '`return`' is an FDL variable (see below).

3 If an annotation is an implicit return annotation of the form:

```
--# return E => Pred (..., E, ...);
```

then the return annotation predicate is '`Pred (..., return, ...)`' (ie the predicate from the annotation, with all instances of E replaced by `return`).

Note that the Examiner automatically generates an FDL declaration of the variable '`return`', whose type matches the return type of the Ada function. This variable is used (as appropriate) in both the concrete and abstract return annotation predicates, to make the formal link in the refinement VC between the constraints specified by the two return annotations (ie both return annotations specify constraints on the *same* return value). '`return`' is used as the variable name as it is a reserved word of SPARK. This guarantees that there is no possibility of the name clashing with any identifiers from the SPARK code.

A useful technique is to use a function's implicitly-declared proof function as its abstract return annotation (see section 3.4.2) together with some more specific concrete annotation in the package body. For example:

```
package R
--# own State;
--# initializes State ;
is
  function Extract return Integer;
  --# global State;
  --# return Extract (State); -- implicit proof function
end R;

package body R
--# own State is X;
is
  X : Integer := 0;

  function Extract return Integer
  --# global X;
  --# return X; -- refined return annotation
  is
  begin
    return X;
  end Extract;
end R;
```

The refinement proof will be of the form:

```
function_extract_3.
H?:    x = fld_x(state) .
H?:    return = x .
        ->
C1:    return = r__extract(state) .
```

Which can be proved by providing suitable proof rules for implicit proof function `r__extract`. This technique does not directly simplify reasoning about external uses of function `Extract` because proof rules written in terms of package R's abstract state will be needed for this purpose; however, it will clarify the relationship between the abstract and refined state and make those rules easier to formulate and understand. Advice on refinement proof rules is at section 8.7.5.

6.6.6 Nested refinement resulting from embedded and/or child packages

An abstract own variable (`P.X`, say) may be refined to variables of embedded, private child or the parent of public child packages (the last two cases apply to SPARK 95 only). If the variables of the embedded/child/parent packages are concrete then the refinement proofs work exactly as described above (see also the stack example in section 8.6). This is because within the body of package P, the types of the concrete variables can be made visible (if necessary by using type announcement in the embedded/child/parent packages) and hence VCG within P works as already described.

If some of the variables of the embedded/child/parent packages are also abstract, then a nested refinement results. Within P, a record type is created for X and the types of the appropriate components are abstract (as is the type of X external to P). Reasoning about these components within P is conceptually no different from reasoning about `P.X` external to P. Hence, within P the user proof rules (which define the abstract proof functions relating to X) simply refer to proof functions of the embedded/child/ parent packages that characterise the abstract components of X.

6.6.7 Private type refinement

The preceding parts of section 6.6 describe own variable refinement proofs. The situation for private type refinement is essentially the same except that there is no refinement abstraction relation (as decribed in 6.6.1) because the same variables appear in both the abstract and refined view.

A simple example of abstract type refinement follows:

```
package P
is
   type T is private;

   --   proof functions on the private type
   --# function IsValid (X : T) return Boolean;
   --# function IntValOf (X : T) return Integer;

   procedure Get (X : in     T;
                  Y :    out Integer);
   --# derives Y from X;
   --# pre IsValid (X);  -- abstract constraint
   --# post Y = IntValOf (X);

private
   type T is record
      Valid : Boolean;
      Value : Integer;
   end record;
end P;

package body P
is
   procedure Get (X : in     T;
                  Y :    out Integer)
   --   a second flow relation is not required nor allowed
   --# pre X.Valid;  -- concrete constraint
   --# post Y = X.Value;
   is
   begin
      Y := X.Value;
   end Get;
end P;
```

Calls to get from outside package P will use the abstract constraint; those from inside will use the concrete (refined) one.

The VCs generated for Get are as follows (standard 'variables in type' hypotheses have been omitted):

```
For path(s) from start to finish:

procedure_get_1.
H1:    fld_valid(x) .
        ->
C1:    fld_value(x) = fld_value(x) .


For checks of refinement integrity:

procedure_get_2.
H1:    isvalid(x) .
        ->
C1:    fld_valid(x) .


procedure_get_3.
H1:    isvalid(x) .
H?:    fld_valid(x) .
H?:    y = fld_value(x) .
        ->
C1:    y = intvalof(x) .
```

6.7 Simplification of trivially true VCs by the Examiner

It is quite common for VCs to have a conclusion of `true`, eg when generating run-time check (RTC) VCs for code that does not have pre- or postconditions. However, the hypotheses of such VCs can still be very large so we often get VCs of the form:

```
a_very_long_list_of_hypotheses
->
true .
```

Although the SPADE Simplifier readily discharges the proof obligation, it has to read and parse the long list of hypotheses before finding they are not needed; this wastes a significant amount of time. The Examiner now identifies such trivially true VCs and replaces the whole VC with:

```
*** true .     /* trivially true VC removed by Examiner */
```

This format is recognised by the Simplifier, which does not process the VC further.

6.8 Traceability between VCs and their source

To enable the comparison of VCs against the corresponding source code, the Examiner inserts traceability comments in VC files that state what the ‘kinds’ of VCs are and which portions of the code they are generated from.

The Examiner generates a VC for each path between two cutpoints (see Section 2.4). The VCs for all the paths between two cutpoints are grouped together, with a single comment before them that has the general form:

For path(s) from XXX to YYY:

where XXX and YYY denote the proof contexts of the two cutpoints. The following table lists the possible values of XXX (with explanations).

start	proof context at the start of the subprogram, ie its precondition
assertion of line *NN*	user-defined assert statement at line *NN*
default assertion of line *NN*	loop invariant assert statement inserted by the Examiner, at line NN, for RTC VCs

The following table lists the possible values of YYY (with explanations).

assertion of line *NN*	user-defined assert statement at line *NN*
default assertion of line *NN*	loop invariant assert statement inserted by the Examiner, at line NN, for RTC VCs
check associated with statement of line *NN*	user-defined check statement at line *NN*
run-time check associated with statement of line *NN*	Examiner generated check statement for a RTC VC for a SPARK statement at line *NN*
precondition check associated with statement of line *NN*	precondition of the subprogram called at line *NN*
finish	proof context at the end of the subprogram, ie its postcondition or return annotation

The sole exceptions to the above traceability commenting of VCs are as follows:

Refinement integrity VCs (whose cutpoints are the abstract and concrete proof contexts of subprograms). The pair of refinement VCs for a single subprogram is prefixed by the comment:

```
For checks of refinement integrity:
```

Extended , tagged type subtype checks as described in Section 2.7.5.1 and illustrated in Section 8.8. The pair of VCs generated is prefixed by the comment:

```
For checks of subclass inheritance integrity
```

7 The Proof Process with SPARK

7.1 Process

The process of developing provable SPARK code involves consideration of the following:

- The annotate-prove cycle and the effect of specification and code changes on proofs performed.
- Use of the tools to facilitate this process.
- The definition of proof functions, and the proof of properties of proof functions.
- Independent assessment activities.
- Configuration management of proofs.

7.2 Annotation and proof of spark code

As we have already noted, proof annotations should ideally be derived from a formal specification. In a practical project, however, this specification will be subject to change: errors or omissions may be found in it (perhaps through attempting code proof), requirements may change or a more efficient approach may suggest changes to the specification and/or the code.

7.2.1 Changes to called subprograms

Each subprogram can be thought of as having two definitions: its formal specification (the “what”) in the form of a precondition and a postcondition, visible to the outside world, and its implementation (the “how”), in the form of executable code.

In what ways, typically, can a subprogram `Called`, called by a higher-level subprogram `Caller`, change?

- A change to `Called`'s body, but not to its specification.
- A change to `Called`'s specification, perhaps as a result of an attempt to prove the code, but not to the code:
- A change to `Called`'s specification and to its code.
- More fundamentally, it may disappear entirely if the changes to `Caller` are sufficiently wide-ranging. (In this case, there is little general advice that we can offer to cope with such changes.)

7.2.2 Changing only the body of a called subprogram

This is the most benign change of all, from the point of view of bottom-up verification. If `Called`'s visible specification is unchanged, the only proof work necessary as a result of the change is to modify internal assertions and check statements as appropriate for the code change carried out, then to generate VCs for `Called` and prove these.

Where the code change is small, it may be possible to replay some or all of an earlier proof of the routine by using the Proof Checker's *execute* facility, provided command scripts have been kept. However, in general, this may involve editing the script; eg

- hypothesis and/or conclusion numbering may change if internal annotations are altered;
- the content of hypotheses may also change, so that the commands need to be modified accordingly;
- additional conclusions resulting from internal annotation changes will also need to be proved.

Further information on the use of the Proof Checker in this way can be found in the *"SPADE Proof Checker User Manual"*.

7.2.3 Changing only the specification of a called subprogram

In this case, `Called` will need to be reproved; depending on the scale of the changes, replay of existing proof command scripts for the subprogram may succeed with appropriate changes, as for the code-change-only case.

Suppose we have the following:

```
procedure OldCalled (...);      procedure NewCalled (...);
--# pre  OldPre;                --# pre  NewPre;
--# post OldPost;               --# post NewPost;
```

If we can prove that *OldPre ⇒ NewPre*, and only the precondition has changed, then we do not need to reprove `Caller`, since whenever `Caller` called `Called` it did so while satisfying the stronger precondition *OldPre* which suffices to show *NewPre* holds, too.

If we can prove that *NewPost ⇒ OldPost*, then, again, it may not be necessary to reprove `Caller`. In such a case, however, where the postcondition of `Called` has effectively been strengthened, it is likely that this we will want this change to ripple up (into `Caller`'s postcondition, etc.)

If the specification of `Called` has changed in some other way, eg an additional parameter, we will almost certainly need to reverify `Caller` as well. Where the change is slight, and a strong argument can be made to contain the change (eg a strengthening of the precondition of `Called` for a single variable), we may be able to carry out a limited reproof (eg in this example, adding a `check` statement in `Caller` to check the stronger precondition constraint before each call of `Called`, and carrying out only proofs of VCs leading to this check statement).

In general, though, reproof of `Caller` will be necessary, perhaps with changes to its specification and consequent ripple-up. Nevertheless, the command scripts and proof logs of earlier proofs can be very helpful in carrying out the reproofs in less time than the original proofs.

7.2.4 Changing both the body and the code of a called subprogram

This will necessitate reproof of `Called`. Again, previous proof logs and command scripts can help to carry out such reproofs in less time than the original proofs, the extent to which this is the case depending on the scale of the changes carried out.

Again, reproof of the calling environment `Caller` is also highly likely to be necessary. As for the previous case, for relatively minor changes it may be possible to construct a rigorous argument that reproof is unnecessary, but these cases are relatively rare.

As for the previous case, too, such changes, if they affect the visible specification of `Caller`, may also ripple up the calling hierarchy, requiring a number of reproofs to accommodate the change.

7.3 Use of the Examiner and other proof tools

As already noted, when used to generate VCs, the Examiner generates them for *all* suitable subprograms in the file(s) submitted to it.

Care must therefore be taken to ensure that previous versions of VC files generated are not overwritten when the Examiner is run, making comparison between old and new VC files to determine the extent of changes impossible. This is particularly important on the Sun and PC platforms (whereas in VMS, the previous version will often still be present until the relevant files are purged).

The Simplifier and Proof Checker also overwrite the log and script files that they create when they are re-run on the same filename in the same subdirectory. However, there is the slight protection on the Sun platform that the previous version (eg *myproof.plg*) is renamed (to *myproof.plg-*), though the previous-previous version, if any, will still be lost. Be warned also that a proof carried out in multiple sessions will, by default, overwrite logs of previous sessions. Again, VMS preserves previous versions of all such files until purged (whether manually, or automatically via a version-limit).

The POGS tool helps with the management of proof work by providing a summary of VCs, indicating their source and current proof status. This summary also identifies any proof work done by either the Simplifier or Proof Checker that is out of date, due to VCs having been re-generated since the proofs were produced.

7.4 Definition of proof functions

Where a proof function has been used in the annotations of a subprogram, it will require definition in a proof rule file. Such definitions should ideally be the subject of independent scrutiny, both to ensure conformance with the relevant specification items (if any), and to check that the definition is well-founded.

As well as definitional rules, it may also be advantageous to specify some properties of such proof functions; in this case, these properties should be rigorously shown to follow from the definition: this can be done with the Proof Checker, with suitable editing, provided the necessary definitions are first-order. (The means of proving such property rules is described in the *SPADE Proof Checker Rules Manual*.)

Particular care must be taken when defining proof rules for functions used in abstract own variable refinement, see Section 8.7.5 for further details.

7.5 Independent assessment

To reduce the risk of early mistakes having wider impact, regular independent assessment of the annotation, coding and verification activities is to be recommended. This assessment can check, in addition to the scrutiny of proof function definitions referred to earlier:

- that property proof rules have been proven to follow from the definitional rules;
- that the data and algorithmic refinements carried out in moving from specification to code seem appropriate and properly documented, and that the required functionality has been properly expressed in the annotations;
- that all necessary proof documentation is being retained, under suitable control;
- that, by selecting a sample routine, for instance, proofs can be replayed and are therefore up-to-date and consistent with the accompanying rule files.

8 Examples

The following examples show the generation of VCs from SPARK code. Small examples, to illustrate particular points, are followed by a larger example that includes elements of a real-time, embedded controller.

8.1 Integer increment

The following procedure simply increments the integer parameter passed to it. Parameter X is both imported and exported and the postcondition may therefore make use of the ~ decoration to indicate its *initial* value.

```
package P
is
end P;

package body P
is

  procedure Inc(X : in out Integer)
  --# derives X from X;
  --# post X = X~ + 1;
  is
  begin
      X := X + 1;
  end Inc;

+++        Flow analysis of subprogram Inc performed: no
           errors found.


end P;
```

Running the SPARK Examiner, with VC-generation selected, produces the following VCs for procedure Inc.

```
For path(s) from start to finish:

procedure_inc_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
        ->
C1:    x + 1 = x + 1 .
```

There are no useful hypotheses but the single conclusion is clearly true showing that the procedure correctly implements its specification. Straightforward VCs are automatically eliminated by either the Examiner (see Section 6.7) or the SPADE Automatic Simplifier.

8.2 Swap routine

Three examples follow of a simple procedure that exchanges the values of its two parameters. The first is correct, the second shows the unprovable VCs which occur when the code is incorrectly implemented and the third shows that VCs can still be generated for code which contains flow errors.

8.2.1 Correct version

```
Line
   1  package P
   2  is
   3  end P;
   4
   5  package body P
   6  is
   7    procedure Swap(X, Y : in out Integer)
   8    --# derives X from Y &
   9    --#         Y from X;
  10    --# post    X = Y~ and Y = X~;
  11    is
  12      T : Integer;
  13    begin
  14      T := X;
  15      X := Y;
  16      Y := T;
  17    end Swap;

+++        Flow analysis of subprogram Swap performed: no
           errors found.

  18
  19  end P;
```

Again the VCs are trivially correct.

```
For path(s) from start to finish:

procedure_swap_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
        ->
C1:    y = y .
C2:    x = x .
```

8.2.2 With semantic error

This version of Swap contains a semantic error which does not affect the flow analysis of the subprogram since the dependencies described by its derives annotation remain correct.

```
Line
   1  package P
   2  is
   3  end P;
   4
   5  package body P
   6  is
   7    procedure Swap(X, Y : in out Integer)
   8    --# derives X from Y &
   9    --#         Y from X;
  10    --# post    X = Y~ and Y = X~;
  11    is
  12      T : Integer;
  13    begin
  14      T := X;
  15      X := Y;
  16      Y := T + 1; -- error, but information flow unaffected
  17    end Swap;

+++        Flow analysis of subprogram Swap performed: no
           errors found.

  18
  19  end P;
```

The following VC results, with C2 being clearly unprovable.

```
For path(s) from start to finish:

procedure_swap_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
        ->
C1:    y = y .
C2:    x + 1 = x .
```

Where VCs are not provable in their entirety, the Simplifier may still be able to prove parts of them and make other simplifications. For example, running the Simplifier on the above VC proves C1 and shows that C2 is not just unprovable but actually false.

```
For path(s) from start to finish:

procedure_swap_1.
```

```
H1:    x >= integer__first .
H2:    x <= integer__last .
H3:    y >= integer__first .
H4:    y <= integer__last .
       ->
C1:    false .
```

8.2.3 With flow errors

The final example shows the importance of considering the significance of flow errors before embarking on a possibly labour-intensive proof process. An error in the code leads to an assignment statement becoming ineffective.

```
Line
   1  package P
   2  is
   3  end P;
   4
   5  package body P
   6  is
   7    procedure Swap(X, Y : in out Integer)
   8    --# derives X from Y &
   9    --#         Y from X;
  10    --# post    X = Y~ and Y = X~;
  11    is
  12      T : Integer;
  13    begin
  14      T := X;
          ^1
!!! (  1)  Flow Error        : Ineffective statement.

  15      X := Y;
  16      Y := X;
  17    end Swap;

!!! (  2)  Flow Error        : Importation of the
           initial value of variable X is
           ineffective.
!!! (  3)  Flow Error        : The variable T is
           neither referenced nor exported.
!!! (  4)  Flow Error        : The imported value of
           X is not used in the derivation of Y.
??? (  5)  Warning           : The imported value of
           Y may be used in the derivation of Y.

  18
  19  end P;
```

VCs can still be generated but again C2 is not provable.

```
For path(s) from start to finish:

procedure_swap_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
        ->
C1:    y = y .
C2:    y = x .
```

Unlike the previous example, C2 is not actually false because there are *some* circumstances in which it is true. The Simplifier reduces the VC to

```
For path(s) from start to finish:

procedure_swap_1.
H1:    x >= integer__first .
H2:    x <= integer__last .
H3:    y >= integer__first .
H4:    y <= integer__last .
       ->
C1:    y = x .
```

showing that the VC can be proved only for the case where the values to be swapped are equal!

8.3 Procedure call

This example makes use of the (correct) Swap procedure defined above and uses it to implement a procedure that does nothing, by swapping its parameters twice. To show how a check is made that a called procedure satisfies its precondition, a precondition has been introduced for procedure Swap. To ensure that Swap can be called without breaking its precondition, a similar one is required for procedure DoNothing.

```
package P
is
end P;

package body P
is
  procedure Swap(X, Y : in out Integer)
  --# derives X from Y &
  --#         Y from X;
  --# pre     X /= Y;
  --# post    X = Y~ and Y = X~;
  is
  13      T : Integer;
  14    begin
  15      T := X;
  16      X := Y;
  17      Y := T;
  18    end Swap;

+++        Flow analysis of subprogram Swap
           performed: no errors found.

  19
  20    procedure DoNothing(X, Y : in out Integer)
  21    --# derives X from X &
  22    --#         Y from Y;
  23    --# pre     not (X = Y);
  24    --# post    X = X~ and Y = Y~;
  25    is
  26    begin
  27      Swap(X, Y);
  28      Swap(X, Y);
  29    end DoNothing;

+++        Flow analysis of subprogram DoNothing
           performed:  no errors found.

  30  end P;
```

VCs are produced for procedures `Swap` and `DoNothing`. Those for `Swap` are as before except that the precondition (`X /= Y`) appears as a hypothesis. The VCs for `DoNothing` are as follows.

```
For path(s) from start to check associated with statement of line 27:

procedure_donothing_1.
H1:    not (x = y) .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
        ->
C1:    x <> y .
```

```
For path(s) from start to check associated with statement of line 28:

procedure_donothing_2.
H1:    not (x = y) .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
H6:    x <> y .
H7:    x__1 = y .
H8:    y__1 = x .
        ->
C1:    x__1 <> y__1 .


For path(s) from start to finish:

procedure_donothing_3.
H1:    not (x = y) .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
H6:    x <> y .
H7:    x__1 = y .
H8:    y__1 = x .
H9:    x__1 <> y__1 .
H10:   x__2 = y__1 .
H11:   y__2 = x__1 .
        ->
C1:    x__2 = x .
C2:    y__2 = y .
```

The first two VCs are associated with the two calls to procedure Swap and are necessary to show that Swap's precondition is met at the point of call. The third VC is for the entire operation of DoNothing and is to show that its postcondition follows from its precondition and code.

The Simplifier can be applied to these VCs giving

```
For path(s) from start to check associated with statement of line 27:

procedure_donothing_1.
*** true .          /* all conclusions proved */


For path(s) from start to check associated with statement of line 28:

procedure_donothing_2.
*** true .          /* all conclusions proved */
```

```
For path(s) from start to finish:

procedure_donothing_3.
*** true .          /* all conclusions proved */
```

8.4 Structured variables

The following example extends the idea of a Swap procedure to illustrate the use of the update notation in SPARK proof contexts. This version of Swap takes an array and exchanges the two elements selected by indices passed in as parameters. The postcondition can be read as *array A is the original array A with its I^{th} element replaced by its original J^{th} element and its J^{th} element replaced by its original I^{th} element*.

```
Line
   1  package P
   2  is
   3    type Index is range 1..10;
   4    type Atype is array(Index) of Integer;
   5  end P;
   6
   7  package body P
   8  is
   9    procedure SwapElement(I, J : in     Index;
  10                          A    : in out Atype)
  11    --# derives A from A, I, J;
  12    --# post A = A~[I => A~(J); J => A~(I)];
  13    is
  14      temp : Integer;
  15    begin
  16      temp := A(I);
  17      A(I) := A(J);
  18      A(J) := temp;
  19    end SwapElement;

+++        Flow analysis of subprogram SwapElement
           performed: no errors found.

  20  end P;
```

The VCs show the modelling of the array accesses and updates using FDL's `element` and `update` functions.

```
For path(s) from start to finish:

procedure_swapelement_1.
H1:    true .
H2:    i >= index__first .
H3:    i <= index__last .
H4:    j >= index__first .
H5:    j <= index__last .
```

```
H6:     for_all (i___1: index, (element(a, [i___1]) >=
           integer__first) and (element(a, [i___1]) <=
           integer__last)) .
           ->
C1:    update(update(a, [i], element(a, [j])), [j], element(
           a, [i])) = update(update(a, [i], element(a, [j])), [
           j], element(a, [i])) .
```

which the Simplifier completely reduces.

Note that this partial proof ignores the possibility of `SwapElement` being called with parameters `I` and `J` outside the range of their type `Index` and the constraint error that would result. VCs to demonstrate that such conditions do not occur can be produced by adding a precondition and suitable `check` statements to the above code. More conveniently, such checks can be inserted automatically by the SPARK Examiner with Run-time Checker.

8.5 Loop invariants

This example uses the naive integer square root algorithm introduced earlier to illustrate the generation of VCs for SPARK code containing a loop.

```
Line
package P
is
end P;

package body P
is
  procedure IntRoot(N    : in     Natural;
                    Root :    out Natural)
  --# derives Root from N;
  --# pre  N >= 0;
  --# post (Root * Root <= N) and
  --#      (N < (Root + 1) * (Root + 1));
  is
    R    : Natural := 0;
    S, T : Natural := 1;
  begin
    loop
      --# assert (T = 2 * R + 1)          and
      --#        (S = (R + 1) * (R +  1)) and
      --#        (R * R <= N);
      exit when S > N;

      R := R + 1;
      T := T + 2;
      S := S + T;
    end loop;
    Root := R;
  end IntRoot;
```

```
+++        Flow analysis of subprogram IntRoot
           performed:  no errors found.

  29  end P;
  30
```

VCs are generated from the start of the subprogram to the assert statement (loop invariant); from the assert statement round the loop back to the assert statement; and from the assert statement to the subprogram end. Note that the precondition `N >= 0` is strictly unnecessary because this restriction is already enforced by the choice of type `Natural` for the input parameter to the procedure

For path(s) from start to assertion of line 18:

```
procedure_introot_1.
H1:    n >= 0 .
H2:    n >= natural__first .
H3:    n <= natural__last .
        ->
C1:    1 = 2 * 0 + 1 .
C2:    1 = (0 + 1) * (0 + 1) .
C3:    0 * 0 <= n .


For path(s) from assertion of line 18 to assertion of line 18:

procedure_introot_2.
H1:    t = 2 * r + 1 .
H2:    s = (r + 1) * (r + 1) .
H3:    r * r <= n .
H4:    not (s > n) .
        ->
C1:    t + 2 = 2 * (r + 1) + 1 .
C2:    s + (t + 2) = (r + 1 + 1) * (r + 1 + 1) .
C3:    (r + 1) * (r + 1) <= n .


For path(s) from assertion of line 18 to finish:

procedure_introot_3.
H1:    t = 2 * r + 1 .
H2:    s = (r + 1) * (r + 1) .
H3:    r * r <= n .
H4:    s > n .
        ->
C1:    r * r <= n .
C2:    n < (r + 1) * (r + 1) .
```

The Simplifier easily proves all three VCs.

8.6 Quantifiers

In Section 3.2.3 we note that quantified expressions are a convenient way to express constraints about array data structures. One of the examples given is the following function, which returns the index of the first occurrence in a table (array) of an input value `Sought`.

```
type Index is range 1..1000;
type Table is array(Index) of Integer;

function FindSought(A      : Table;
                    Sought : Integer) return Index;
--# pre for some M in Index => ( A(M) = Sought );
--# return Z => (( A(Z) = Sought) and
--#    (for all M in Index range 1 .. (Z-1) =>
--#        (A(M) /= Sought)));
```

The precondition of the function ensures that it is only called if `Sought` is definitely present in the table.

The following code gives a body for the function, which uses a loop to search through the table from its first element until `Sought` is found.

```
function FindSought(A      : Table;
                    Sought : Integer) return Index
is
  Z : Index := Index'first;
begin
  while A(Z) /= Sought and Z < Index'last loop
    --# assert for all I in Index range 1 .. Z
    --#           => (A(I) /= Sought);
    Z := Z + 1;
  end loop;
  return Z;
end FindSought;
```

The `assert` statement gives the loop invariant that `Sought` does not occur in any of the values already searched, ie any values whose index is less than or equal to Z.

The VCs generated for this function body, *after* simplification by the Simplifier, are:

```
For path(s) from start to assertion of line 22:

function_findsought_1.
H1:    for_some(m_ : index,m_ >= 1 and m_ <= 1000
                           and element(a,[m_]) = sought) .
H2:    for_all(i___1 : index,
               element(a,[i___1]) >= integer__first
               and element(a,[i___1]) <= integer__last) .
H3:    sought >= integer__first .
H4:    sought <= integer__last .
H5:    element(a,[1]) <> sought .
       ->
```

```
C1:    for_all(i : index,
               i >= 1 and i <= 1 -> element(a,[i]) <> sought) .
```

The first VC above is for the code from the start of the function to the loop invariant. The conclusion C1 is easily proved as the constraints on the quantified variable i (ie it equals 1) mean that the required property follows immediately from hypothesis H5.

```
For path(s) from assertion of line 22 to assertion of line 22:

function_findsought_2.
H1:    for_all(i : index,i >= 1 and i <= z ->
                         element(a,[i]) <> sought) .
H2:    element(a,[z + 1]) <> sought .
H3:    m < 999 .
       ->
C1:    for_all(i : index,i >= 1 and i <= z + 1 ->
                         element(a,[i]) <> sought) .
```

The second VC above is for the code from the loop invariant round the loop and back to the invariant. The conclusion C1 follows from hypotheses H1 and H2.

```
For path(s) from start to finish:

function_findsought_3.
H1:    for_some(m_ : index,m_ >= 1 and m_ <= 1000
                         and element(a,[m_]) = element(a,[1])) .
H2:    for_all(i___1 : index,
               element(a,[i___1]) >= integer__first
               and element(a,[i___1]) <= integer__last) .
H3:    element(a,[1]) >= integer__first .
H4:    element(a,[1]) <= integer__last .
       ->
C1:    for_all(m_ : index,m_ >= 1 and m_ <= 0 ->
                          element(a,[m_]) <> element(a,[1])) .
```

The third VC above is for the code from the start to the end of the function, when the loop is not entered (ie Sought is the first element of the table). The conclusion C1 is easily proved as the constraints on the quantified variable, i, are contradictory and hence the quantification is vacuously true. (Note that the VC generated by the Examiner actually has two conclusions, but the Simplifier proves the first one.)

```
For path(s) from assertion of line 22 to finish:

function_findsought_4.
H1:    for_all(i : index,i >= 1 and i <= z ->
                         element(a,[i]) <> sought) .
H2:    element(a,[z + 1]) = sought or 999 <= z .
       ->
C1:    element(a,[z + 1]) = sought .
C2:    for_all(m_ : index,m_ >= 1 and m_ <= z ->
                          element(a,[m_]) <> sought) .
```

The final VC above is for the code from the loop invariant to the end of the function, when the while-loop condition no longer holds. Conclusion C2 follows immediately from H1, but conclusion C1 is unprovable as H2 is not quite strong enough. The real problem is that the loop invariant does not include the function's precondition and so the precondition does not appear as a hypothesis in the VC. Adding the precondition to the loop invariant results in another hypothesis in the VC (the same as H1 in the first VC above) which combined with the existing hypotheses is sufficient to prove C1. This illustrates the importance of including enough information in invariants.

8.7 Abstract own variable refinement for a Stack

This section uses a simple example to provide an introduction to abstract own variable refinement proofs. The presentation of the example consists of:

- an abstract stack specification using an abstract proof type;
- the corresponding concrete implementation;
- the VCs that are generated to prove the refinement;
- an illustration of how the abstract stack specification can be used in other packages that make use of the stack.

8.7.1 Abstract Stack Package Specification

A complete abstract package specification for a stack is as follows.

```
package Stack
--# own State : StackType;            -- type announced abstract variable
--# initializes State;
is

  --# type StackType is abstract;     -- SPARK proof annotation
                                      -- (declares a new proof type)
  function IsEmpty return Boolean;
  --# global State;

  function NotFull return Boolean
  --# global State;

  --# function Append(S : StackType; X : Integer) -- use of proof
  --#   return StackType;                         -- type

  procedure Push(X : in Integer);
  --# global in out State;
  --# derives State from State, X;
  --# pre  NotFull(State);            -- abstract precondition
  --# post State = Append(State~, X); -- abstract postcondition

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

```
    --# derives State, X from State;
    --# pre  not IsEmpty(State);         -- abstract precondition
    --# post State~ = Append(State, X); -- abstract postcondition

  procedure Clear;
    --# global out State;
    --# derives State from ;
    --# post IsEmpty(State);            -- abstract postcondition

  end Stack;
```

Although the type `StackType` can only be used in SPARK proof annotations, the Examiner generates an 'illegal redeclaration' error if an Ada object with the same name is declared in any scope where the proof type is visible. Furthermore, as the type announcement for the variable `State` uses an abstract proof type, there must not be a concrete declaration of `State` in the package. In addition if there is more than one abstract variable, then each must have its own abstract proof type (because of the meaning that the built-in abstraction relation gives to abstract types – see Section 8.7.3).

In the above example the abstract proof type has the same meaning for the Examiner as if the user had typed:

```
package Stack
--# own State : StackType;
--# initializes State;
is
  type StackType is private;
  ...
private
  --# hide Stack;
end Stack;
```

So the only information given about the type `StackType` is that it is possible to compare two variables of that type for equality.

8.7.2 Concrete Stack Implementation

The package body for the stack example is straightforward.

```
  package body Stack
  --# own State is Ptr, Vector;
  is
    MaxStackSize : constant := 100;

    type    Ptrs    is      range 0..MaxStackSize;
    subtype Indexes is Ptrs range 1..Ptrs'last;
    type    Vectors is array (Indexes) of Integer;

    Ptr    : Ptrs := 0;
    Vector : Vectors := Vectors'(Indexes => 0);
    function IsEmpty return Boolean
    --# global Ptr;
    --# return (Ptr = 0);                -- concrete return annotation
    is
    begin
      return Ptr = 0;
    end IsEmpty;

    function NotFull return Boolean
    --# global Ptr;
    --# return Ptr < MaxStackSize;       -- concrete return annotation
    is
    begin
      return Ptr < MaxStackSize;
    end NotFull;

    procedure Push(X : in Integer)
    --# global in out Vector, Ptr;
    --# derives Ptr from Ptr &
    --#         Vector from Vector, Ptr, X;
    --# pre  Ptr < MaxStackSize;         -- corresponds to NotFull
    --# post Ptr = Ptr~ + 1 and
    --#      Vector = Vector~[Ptr => X]; -- corresponds to Append
    is
    begin
      Ptr := Ptr + 1;
      Vector(Ptr) := X;
    end Push;

    procedure Pop(X : out Integer)
    --# global in     Vector;
    --#        in out Ptr;
    --# derives Ptr from Ptr &
    --#         X   from Vector, Ptr;
    --# pre  Ptr > 0;
    --# post Ptr = Ptr~ - 1 and X = Vector(Ptr~);
    is
    begin
      X := Vector(Ptr);
      Ptr := Ptr - 1;
    end Pop;

    procedure Clear
    --# global out Ptr;
    --# derives Ptr from ;
    --# post Ptr = 0;
    -- flow error: inconsistency with abstract flow annotation as Vector is unchanged
    is
    begin
      Ptr := 0;
    end Clear;
  end Stack;
```

Note that the subprograms have concrete proof annotations, in addition to the abstract annotations in the package specification. This is exactly analogous to the abstract and concrete `global` and `derives` flow annotations.

8.7.3 Refinement VCs

When generating VCs for the package body the Examiner produces the following FDL definition for the proof type `StackType`.

```
type stacktype = record
    vector : vectors;
    ptr : ptrs
end;
```

The Examiner also generates an abstraction relation between the concrete and abstract variables, where the concrete variables (`Vector` and `Ptr`) are the appropriate fields of the abstract variable (`State`).[9]

Generating the refinement VCs is a simple matter of:

- extracting the abstract and concrete pre- and postconditions for 'weaken pre' and 'strengthen post' VCs (as appropriate);
- adding abstraction relation hypotheses for any concrete variables which are imported and/or exported (as given by the flow annotations in the package body).

Consider the procedure `Push`. The '`-vcs`' (`/VCS` on VAX or NT) option of the Examiner produces three VCs. (One VC for the code against the concrete pre/post specification and two refinement VCs.) The precondition refinement VC is as follows.

```
procedure_push_2.
H1:    stack__notfull(state) .
H2:    vector = fld_vector(state) .
H3:    ptr = fld_ptr(state) .
H4:    x >= integer__first .
H5:    x <= integer__last .
H6:     for_all (i___1: ptrs, ((i___1 >= indexes__first) and (
          i___1 <= indexes__last)) -> ((element(vector, [
          i___1]) >= integer__first) and (element(vector, [
          i___1]) <= integer__last))) .
H7:    ptr >= ptrs__first .
H8:    ptr <= ptrs__last .
        ->
C1:    ptr < maxstacksize .
```

[9] This is a functional data refinement, ie the abstraction relation is a total function from concrete to abstract, and hence simplifies the resulting refinement VCs.

Hypothesis H1 is the abstract precondition and conclusion C1 the concrete precondition. H2 and H3 are the abstraction relation hypotheses that relate the abstract and concrete variables. (H4 to H8 are the standard 'variables in type' hypotheses, typically used for RTC proofs.)

The postcondition refinement VC is as follows.

```
procedure_push_3.
H1:    stack__notfull(state~) .
H2:    ptr~ < maxstacksize .
H3:    vector~ = fld_vector(state~) .
H4:    vector = fld_vector(state) .
H5:    ptr~ = fld_ptr(state~) .
H6:    ptr = fld_ptr(state) .
H7:    x >= integer__first .
H8:    x <= integer__last .
H9:     for_all (i___1: ptrs, ((i___1 >= indexes__first) and (
          i___1 <= indexes__last)) -> ((element(vector~, [
          i___1]) >= integer__first) and (element(vector~, [
          i___1]) <= integer__last))) .
H10:   ptr~ >= ptrs__first .
H11:   ptr~ <= ptrs__last .
H12:    for_all (i___1: ptrs, ((i___1 >= indexes__first) and (
          i___1 <= indexes__last)) -> ((element(vector, [
          i___1]) >= integer__first) and (element(vector, [
          i___1]) <= integer__last))) .
H13:   ptr >= ptrs__first .
H14:   ptr <= ptrs__last .
H15:   ptr = ptr~ + 1 .
H16:   vector = update(vector~, [ptr], x) .
        ->
C1:    state = append(state~, x) .
```

Hypotheses H15 and H16 are the concrete postcondition and conclusion C1 the abstract postcondition. H1 and H2 are the abstract and concrete preconditions. H3 to H6 are the abstraction relation hypotheses that relate the abstract and concrete variables, for both the initial and final states. (H7 to H14 are the standard 'variables in type' hypotheses.)

To formally prove the refinement VCs, proof rules must be supplied that give definitions of the abstract proof functions in terms of the concrete variables. As these rules are only used internally to the Stack package, they can assume the FDL definition of StackType as a record. For example:

```
stack__isempty(S) may_be_replaced_by fld_ptr(S) = 0 .

stack__notfull(S) may_be_replaced_by
   fld_ptr(S) < maxstacksize .

append(S,X) may_be_replaced_by
   mk__stacktype(vector := update(fld_vector(S),
         [fld_ptr(S)+1], X),
            ptr    := fld_ptr(S)+1) .
```

The last two of these rules are sufficient to enable the two refinement VCs for Push to be proved.[10]

8.7.4 Use of Abstract Stack in Other Packages

We can use the stack package to give an alternative implementation of the swap procedure introduced in Section 8.2.

```
procedure Swap (X, Y : in out Integer)
--# global in out Stack.State;
--# derives Stack.State, X, Y from Stack.State, X, Y;
--# pre  Stack.NotFull(Stack.Append(Stack.State, X));
--# post X = Y~ and Y = X~;
is
begin
  Stack.Push(X);
  Stack.Push(Y);
  Stack.Pop(X);
  Stack.Pop(Y);
end Swap;
```

Note that with this implementation, Swap must have a precondition to ensure that there is room to push at least two elements onto the stack. The precondition uses the (public) proof functions of Stack to state the required property of the abstract Stack.State. We shall now consider some of the VCs of Swap to illustrate how the abstract specification of Stack can be used in proof.

The first VC for Swap is to establish the precondition of the first call to Stack.Push.

```
procedure_swap_1.
H1:    stack__notfull(stack__append(stack__state,x)) .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
       ->
C1:    stack__notfull(stack__state) .
```

Intuitively this VC is true as if a stack with a number appended is not full, then the stack without the element appended cannot be full either.

To actually complete the proof (eg using the Proof Checker) a proof rule has to be given for the required property of the abstract proof functions. For example:

```
stack__notfull(S) may_be_deduced_from
      [stack__notfull(stack__append(S, X))].
```

[10] Any reader wondering why the first two rules contain 'stack__isempty' and 'stack__notfull', instead of just 'isempty' and 'notfull', should see section 6.6.2.

As with any user supplied proof rule, there is obligation to verify that this property is valid. That is, to show that the property is true for the concrete implementation of the `Stack` package. This can be expressed as the following VC (with suitable FDL declarations of variables s and x):

```
H1:  stack__notfull(stack__append(s, x)) .
     ->
C1:  stack__notfull(s) .
```

Using the previously defined proof rules, for proofing the refinement VCs internal to the `Stack` package, this expands to:

```
H1:  fld__ptr(mk__stacktype(vector :=
        update(fld_vector(s),
           [fld_ptr(s)+1], x),
              ptr    := fld_ptr(s)+1))
                 < maxstacksize.
     ->
C1:  fld__ptr(s) < maxstacksize .
```

which after simplification is trivially true.

The third VC for Swap is to establish the precondition of the first call to `Stack.Pop`. After simplification by the Simplifier this is:

```
procedure_swap_3.
H1:    stack__notfull(stack__append(stack__state,x)) .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
H6:    stack__notfull(stack__state) .
H7:    stack__notfull(stack__append(stack__state,x)) .
       ->
C1:    not stack__isempty(stack__append(stack__append
                                     (stack__state,x),y)) .
```

This can be proved immediately with the following proof rule:

```
not stack__isempty(stack__append(S, X)) may_be_deduced.
```

Again this proof rule needs to be justified in terms of the concrete definitions of the stack proof functions.

The final VC for Swap is the establishment of its postcondition. After simplification by the Simplifier this is:

```
procedure_swap_5.
H1:    stack__notfull(stack__append(stack__state,x)) .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
H6:    stack__notfull(stack__state) .
H7:    stack__notfull(stack__append(stack__state,x)) .
```

```
H8:    stack__append(stack__append(stack__state__4,y__4),x__3)
        = stack__append(stack__append(stack__state,x),y) .
H9:    not stack__isempty(stack__append(stack__append
                              (stack__state__4,y__4),x__3)) .
H10:   not stack__isempty(stack__append(stack__state__4,y__4)) .
       ->
C1:    x__3 = y .
C2:    y__4 = x .
```

This can be proved with the following proof rules:

```
X = Y may_be_deduced_from
  [stack__append(S,X) = stack__append(T,Y)].
V = W may_be_deduced_from
  [stack__append(stack__append(S,V),X) =
     stack__append(stack__append(T,W),Y)].
```

8.7.5 Guidance on refinement proof rules

For any reasoning involving refinement to be sound, each user supplied proof rule, for the proof functions used in a package's abstract specification, **must** be satisfied by the concrete implementation. As we have seen above, there are two kinds of proof rules for refinement proof functions:

1 The internal definitions of abstract functions in terms of concrete variables.

2 The external properties of abstract functions used by callers.

The internal definitions can be proved to be consistent with the actual concrete implementation, by discharging the refinement integrity VCs generated by the Examiner. However, the external properties should be proved consistent with the internal definitions and this can be difficult to establish. For example, consider the following possible rule for the proof function `Stack.Append`.

```
S = T may_be_deduced_from
    [stack__append(S,X) = stack__append(T,Y)].
```

This formalises the intuitive property that:

if two stacks (S and T) are the same after each has had an item appended, then they must have been the same before the items were appended.[11]

However, is this consistent with the implementation in the package body of `Stack`? To try to prove this consistency we can state the proof rule as the following VC:

```
H1:  stack__append(s,x) = stack__append(t,y) .
     ->
C1:  s = t .
```

[11] A proof rule like this is needed if we wish to prove that the procedure Swap satisfies the additional postcondition: `Stack = Stack~.`

Using the internal definition of `stack__append` this expands to:

```
H1:  mk__stacktype(vector := update(fld_vector(s),
                                    [fld_ptr(s)+1], x),
                  ptr    := fld_ptr(s)+1)
      =
     mk__stacktype(vector := update(fld_vector(t),
                                    [fld_ptr(t)+1], y),
                  ptr    := fld_ptr(t)+1) .
     ->
C1:  s = t .
```

By the definition of record equality `H1` and `C1` can both be replaced by two predicates:

```
H1:  update(fld_vector(s), [fld_ptr(s)+1], x)
      = update(fld_vector(t), [fld_ptr(t)+1], y) .
H2:  fld_ptr(s) + 1 = fld_ptr(t) + 1 .
     ->
C1:  fld_vector(s) = fld_vector(t) .
C2:  fld_ptr(s) = fld_ptr(t) .
```

This simplifies to:

```
H1:  update(fld_vector(s), [fld_ptr(s)+1], x)
      = update(fld_vector(t), [fld_ptr(s)+1], y) .
     ->
C1:  fld_vector(s) = fld_vector(t) .
```

which cannot be proved as the values of the `vector` components of `s` and `t` may differ at the position `fld_ptr(s) + 1`. Hence the proof rule is not strictly consistent with the implementation – as the initial values overwritten by appending X and Y can be different, with the final stacks still being equal.

We can (informally) argue that the potential discrepancy, in the initial values of the two stacks' arrays, is irrelevant as it is impossible to observe any values in the arrays beyond the current end of the stacks (as defined by the values of the `ptr` variables). However, in more complex refinements it may be much harder to justify that any such observable behaviours, on which the validity of proof rules depend, really do hold.

Great care therefore needs to be taken when defining proof rules for proof functions used in refinement. Ideally all such rules should be proved against the internal (concrete) definitions of the proof functions, as used in the refinement VC proofs within a package.

8.8 Behavioural subtyping of extended types

This section provides an example of type extension and shows the additional VCs that are generated to show that operations on extended types are true behavioural subtypes as described in Section 2.7.5.1. The Example is made up of an Accumulator which keeps a running total of values added to it and an extension that also keeps a count of the number of items added.

8.8.1 Root “class” specification

```
package Accumulator
is
   type T is tagged record
      TheValue : Integer;
   end record;

   subtype Small is Integer range 1 .. 10;

   procedure Clear (O : out T);
   --# derives O from ;
   --# post O.TheValue = 0;

   procedure Put (O     : in out T;
                  Value : in     Integer);
   --# derives O from O, Value;
   --# pre Value in Small;
   --# post O.TheValue = O~.TheValue + Value;

   function Get (O : T) return Integer;
   --# return O.TheValue;

end Accumulator;
```

8.8.2 Root “class” implementation

```
package body Accumulator
is
   procedure Clear (O : out T)
   is
   begin
      O.TheValue := 0;
   end Clear;

   procedure Put (O     : in out T;
                  Value : in     Integer)
   is
   begin
      O.TheValue := O.TheValue + Value;
   end Put;

   function Get (O : T) return Integer
   is
   begin
      return O.TheValue;
   end Get;
end Accumulator;
```

8.8.3 Type extension specification

```
with Accumulator;
--# inherit Accumulator;
package AccumulatorWithCount
is
   type T is new Accumulator.T with record
      CallCount : Integer;
   end record;

   subtype Large is Integer range 1 .. 100;

   -- overriding operations
   procedure Clear (O : out T);
   --# derives O from ;
   --# post O.TheValue = 0 and O.CallCount = 0;
   -- Note that this postcondition is written in the form:
   --    "root postcondition" and "new term"

   procedure Put (O     : in out T;
                  Value : in     Integer);
   --# derives O from O, Value;
   --# pre Value in Large;
   --# post O = O~[TheValue => O~.TheValue + Value;
   --#              CallCount => O~.CallCount + 1];
   -- Note that this postcondition is written in a completely different form to
   -- that used on Accumulator.Put

   -- Get is inherited unchanged

   -- New function
   function GetCallCount (O : T) return Integer;
   --# return O.CallCount;

end AccumulatorWithCount;
```

8.8.4 Type extension implementation

```
package body AccumulatorWithCount
is
   procedure Clear (O : out T)
   is
   begin
      O.TheValue := 0;
      O.CallCount := 0;
   end Clear;
```

```
procedure Put (O     : in out T;
                 Value : in     Integer)
   is
   begin
      O.TheValue := O.TheValue + Value;
      O.CallCount := O.CallCount + 1;
   end Put;

   function GetCallCount (O : T) return Integer
   is
   begin
      return O.CallCount;
   end GetCallCount;
end AccumulatorWithCount;
```

8.8.5 Additional VCs for overridden Clear operation

The unsimplified VCs for behavioural subyping of `AccumulatorWithCount.Clear` are:

```
For checks of subclass inheritance integrity:

procedure_clear_2.
*** true .          /* trivially true VC removed by Examiner */


procedure_clear_3.
H1:    fld_thevalue(fld_inherit(o)) = 0 .
H2:    fld_callcount(o) = 0 .
        ->
C1:    fld_thevalue(fld_inherit(o)) = 0 .
```

The Clear operation does not have a precondition so the first of these VCs, to show it “requires less” is trivially true. The second, to show that it “promises more”, is also clearly true since H1 and C1 are identical.

8.8.6 Additional VCs for overridden Put operation

The unsimplified VCs for behavioural subyping of `AccumulatorWithCount.Put` are:

```
For checks of subclass inheritance integrity:

procedure_put_2.
H1:    value >= accumulator__small__first .
H2:    value <= accumulator__small__last .
        ->
C1:    value >= large__first .
C2:    value <= large__last .
```

```
procedure_put_3.
H1:    o = upf_callcount(upf_inherit(o~,
           upf_thevalue(fld_inherit(o~),
           fld_thevalue(fld_inherit(o~)) + value)),
           fld_callcount(o~) + 1) .
        ->
C1:    fld_thevalue(fld_inherit(o)) =
           fld_thevalue(fld_inherit(o~)) + value .
```

In this case we do have a check that the precondition of the overriding operation is not stronger than the operation it overrides. This is readily provable since the "small" integer subtype is smaller than the "large" integer subtype as shown by the Examiner-generated proof rules:

```
put_rules(7): accumulator__small__first may_be_replaced_by 1.
put_rules(8): accumulator__small__last may_be_replaced_by 10.
put_rules(12): large__first may_be_replaced_by 1.
put_rules(13): large__last may_be_replaced_by 100.
```

Because of the different form used to express the postcondition of the extended version of Put, the postcondition VC is less obviously true than was the case with Clear; however, it is readily discharged by the Simplifier (as are all the other VCs generated by this example).

8.9 Unconstrained formal parameters

Generally it does not make any sense to make use of *exported* parameters in *pre*conditions. Where, however, the parameter is of an unconstrained array type we may wish to assert properties that the actual parameter must have for the call to function correctly.

For example, consider a string copy routine which copies one uncosntrained parameter to another. We may wish to assert that the exported string is the same length as the imported one.

```
procedure Copy (S : in     String;
                D :    out String)
--# derives D from S;
--# pre D'Length = S'Length; -- note use of export D in precondition
is
begin
   for I in Positive range S'Range loop
      D (I) := S(I);
   end loop;
end Copy;
```

If we use this routine to initialize a string of a particular subtype:

```
subtype Index5 is Positive range 1 ..5;
subtype String5 is String (Index5);
```

```
procedure MakeOk (D : out String5)
--# derives D from ;
is
begin
   Copy ("Hello", D);
end MakeOk;
```

then the Examiner generates a precondition check associated with the call to `Copy`.

```
procedure_makeok_1.
H1:    true .
        ->
C1:    index5__last - index5__first + 1 = positive__5__last -
           positive__5__first + 1 .
```

This is readily provable using the rules generated for the user-defined type `Index5` and the Examiner-generated type `positive__5` (which is created by counting the number of characters in the string literal).

```
makeok_rules(11): index5__first may_be_replaced_by 1.
makeok_rules(12): index5__last may_be_replaced_by 5.
makeok_rules(16): positive__5__first may_be_replaced_by 1.
makeok_rules(17): positive__5__last may_be_replaced_by 5.
```

8.10 Dealing with ports

SPARK includes notations designed to simplify descriptions of systems which import values from or export values to the external environment. Own variables with modes (similar to parameter modes) are used for this purpose; they are known as *external variables*. Section 8.10.1 below describes a method of dealing with ports that pre-dates external variables. It is followed, in Section 8.10.2, by what is essentially the same material except that it is implemented using external variables. A larger example follows these two sections.

8.10.1 Traditional method

When values are read from input ports, care needs to be taken in the way such operations are annotated so as to indicate their behaviour to the SPARK Examiner. The usual way this is done is to consider the input port to be monitoring a “stream” of input values. Reading a value removes it from the head of the stream so that a subsequent read appears, to the Examiner, to return a (possibly) different value. The usual way of annotating such port read operations is to introduce an abstract own variable to represent the input stream. For example:

```
package Sensor
--# own Stream;
--# initializes Stream;
is
  procedure Read(X : out Integer);
  --# global Stream;
  --# derives X, Stream from Stream;
end Sensor;
```

The SPARK Examiner with VC Generator is a concrete proof tool and it is not possible to express a postcondition for the Read operation in terms of its abstract type (which might be regarded, for example, as an arbitrarily large sequence of integers). A method must be found where the postcondition can be expressed in terms of variables having concrete Ada types.

It is possible to regard the abstract own variable (eg `Stream`) as representing the most recently read value itself. Type announcement can be used to give the own variable the correct type and a suitable postcondition added.

```
package Sensor
--# own Stream : Integer;
--# initializes Stream;
is
  procedure Read(X : out Integer);
  --# global Stream;
  --# derives X, Stream from Stream;
  --# post X = Stream~;
end Sensor;
```

This annotated form can then be used to define other parts of the code. For example, consider a procedure which returns the Boolean value `true` if the sensor reading exceeds a certain value.

```
procedure AboveLimit(Result :    out Boolean)
--# global Sensor.Stream;
--# derives Result, Sensor.Stream from Sensor.Stream;
--# post Result <-> (Sensor.Stream~ > 100);
is
  Limit : constant Integer := 100;
  Reading : Integer;
begin
  Sensor.Read(Reading);
  Result := Reading > Limit;
end AboveLimit;
```

VCs can now be generated

```
For path(s) from start to finish:

procedure_abovelimit_1.
H1:    true .
H2:    sensor__stream >= integer__first .
H3:    sensor__stream <= integer__last .
H4:    reading__1 = sensor__stream .
        ->
C1:    (reading__1 > limit) = (sensor__stream > 100) .
```

and simplified

```
For path(s) from start to finish:

procedure_abovelimit_1.
*** true .          /* all conclusions proved */
```

The method can be extended to deal with procedures which need more than one read of a port by introducing a proof function to model the way reading from a port is considered to be consuming a sequence of possible input values and then using this to re-specify the read operation as follows:

```
package Sensor
--# own Stream : Integer;
--# initializes Stream;
is
  --# function NextReading(S : Integer) return Integer;

  procedure Read(X : out Integer);
  --# global Stream;
  --# derives X, Stream from Stream;
  --# post X = Stream~ and Stream = NextReading(Stream~);
end Sensor;
```

Consider a procedure that determines whether the sensor value is rising:

```
procedure Rising(Result :    out Boolean)
--# global Sensor.Stream;
--# derives Result, Sensor.Stream from Sensor.Stream;
--# post Result <-> Sensor.NextReading(Sensor.Stream~) >
--#                   Sensor.Stream~;
is
  Reading1,
  Reading2 : Integer;
begin
  Sensor.Read(Reading1);
  Wait.Ten;
  Sensor.Read(Reading2);
  Result := Reading2 > Reading1;
end Rising;
```

Giving VCs

```
procedure_rising_1.
H1:    true .
H2:    sensor__stream >= integer__first .
H3:    sensor__stream <= integer__last .
H4:    reading1__1 = sensor__stream .
H5:    sensor__stream__1 = sensor__nextreading(sensor__stream) .
H6:    reading2__2 = sensor__stream__1 .
H7:    sensor__stream__2 = sensor__nextreading(sensor__stream__1).
        ->
C1:    (reading2__2 > reading1__1) <-> (sensor__nextreading(
           sensor__stream) > sensor__stream) .
```

which the Simplifier again fully reduces.

Although workable, this approach is not recommended because `Stream` is not really an integer but some abstract object which might be considered to be a sequence of integers. Announcing it to be an integer can be considered sleight of hand which confuses the precision which proof permits.

In general, it is better completely to separate operations on the abstract stream of input values and the actual values returned. One way to do this is to poll sensors outside those parts of the code for which proof is to be attempted, store the read values in suitable variables which are then passed as parameters to, or accessed globally by, subprograms implementing algorithms which are free of outside-world objects. There is then no need to type announce the sensor stream to be of some concrete Ada type, it can remain abstract as it really is.

Such a pure version of `AboveLimit` would be

```
procedure AboveLimit(Reading : in     Integer;
                     Result  :    out Boolean)
--# derives Result from Reading;
--# post Result <-> (Reading > 100);
is
  Limit : constant Integer := 100;
begin
  Result := Reading > Limit;
end AboveLimit;
```

A check that the sensor was above limit would be implemented by polling the sensor and passing the value obtained to the new, clean and provable, `AboveLimit` procedure.

This technique is illustrated further in the example at Section 8.11.

8.10.2 Using external variables

When values are read from input ports, care needs to be taken in the way such operations are annotated so as to indicate their behaviour to the SPARK Examiner. The usual way this is done is to consider the input port to be monitoring a “stream” of input values. Reading a value removes it from the head of the stream so that a subsequent read appears, to the Examiner, to return a (possibly) different value. SPARK permits a mode (either `IN` or `OUT` but not `IN OUT`) to be included in the declaration of an own variable or refinement constituent. The mode indicates communication with the external environment. Once annotated in this way no further special steps are required to obtain an accurate model of the volatile behaviour of input/output. For example:

```
package Sensor
--# own in Stream;
is
  procedure Read(X : out Integer);
  --# global Stream;
  --# derives X from Stream;
end Sensor;
```

For proofs that rely on just the single (most recent) value read from an external variable, it is possible to regard the external abstract variable (e.g. `Stream`) as representing the most recently read value itself. Type announcement can be used to give the external own variable the correct type and a suitable postcondition added.

```
package Sensor
--# own in Stream : Integer;
is
  procedure Read(X : out Integer);
  --# global Stream;
  --# derives X from Stream;
  --# post X = Stream~;
end Sensor;
```

This annotated form can then be used to define other parts of the code. For example, consider a procedure which returns the Boolean value `true` if the sensor reading exceeds a certain value.

```
procedure AboveLimit(Result :    out Boolean)
--# global Sensor.Stream;
--# derives Result from Sensor.Stream;
--# post Result <-> (Sensor.Stream~ > 100);
is
  Limit : constant Integer := 100;
  Reading : Integer;
begin
  Sensor.Read(Reading);
  Result := Reading > Limit;
end AboveLimit;
```

VCs can now be generated

```
For path(s) from start to finish:

procedure_abovelimit_1.
H1:    true .
H2:    sensor__stream >= integer__first .
H3:    sensor__stream <= integer__last .
H4:    reading__1 = sensor__stream .
        ->
C1:    (reading__1 > limit) = (sensor__stream > 100) .
```

and simplified

```
For path(s) from start to finish:

procedure_abovelimit_1.
*** true .          /* all conclusions proved */
```

Where procedures need more than one read of a port, the Examiner generates proof attribute functions (see Section 6.4.5.3) to capture their volatile behaviour. For an external variable of mode `in,` the proof function `'Tail` models the way reading from a port is considered to be consuming a sequence of possible input values. We can use this to re-specify the read operation as follows:

```
package Sensor
--# own in Stream : Integer;
is
  procedure Read(X : out Integer);
  --# global Stream;
  --# derives X from Stream;
  --# post X = Stream~ and Stream = Stream'Tail(Stream~);
end Sensor;
```

Consider a procedure that determines whether the sensor value is rising:

```
procedure Rising(Result :    out Boolean)
--# global Sensor.Stream;
--# derives Result from Sensor.Stream;
--# post Result <-> Sensor.Stream'Tail(Sensor.Stream~) >
--#                   Sensor.Stream~;
is
  Reading1,
  Reading2 : Integer;
begin
  Sensor.Read(Reading1);
  Wait.Ten;
  Sensor.Read(Reading2);
  Result := Reading2 > Reading1;
end Rising;
```

Giving VCs

```
procedure_rising_1.
H1:    true .
H2:    sensor__stream >= integer__first .
H3:    sensor__stream <= integer__last .
H4:    reading1__1 = sensor__stream .
H5:    sensor__stream__1 = sensor__nextreading(sensor__stream) .
H6:    reading2__2 = sensor__stream__1 .
H7:    sensor__stream__2 = sensor__nextreading(sensor__stream__1).
        ->
C1:    (reading2__2 > reading1__1) <-> (sensor__nextreading(
           sensor__stream) > sensor__stream) .
```

which the Simplifier again fully reduces.

For values written to external variables of mode out, the `'Append` proof attribute (see Section 3.3.1.3) is used showing that the written value is appended to the sequence of values seen by the external environment.

A mixture of the approaches described in Sections 8.10.1 and 8.10.2 may be needed where an abstract own variable without a mode is refined onto constituents which are moded external variables. In this case, the proof attributes will be applicable inside the pakcage body and other, user-supplied proof functions will be needed in the abstract, external view.

8.11 An embedded controller

This example forms part of a environmental control system designed to maintain the temperature of some environment between set limits.

The system consists of a clock to determine operating times, a temperature sensor, a mode switch and an actuator that turns a heating system on or off. The requirement is for the heating to be on only when within the operating time range and when the temperature is below some target temperature.

External variables have been used for the interfaces but the control algorithm (`PerformModeOperations`) has been kept separate from the reading of sensors and writing of actuators as described at the end of Section 8.10.1.

8.11.1 The specification

$$\mathit{Times} == 0 \mathrel{..} (24 * 60 * 60 -1)$$

$$\mathit{OnTime} == 32400$$

$$\mathit{OffTime} == 61200$$

$$\mathit{Modes} ::= \mathit{off} \mid \mathit{timed} \mid \mathit{continuous}$$

$$\mathit{Settings} ::= \mathit{isOn} \mid \mathit{isOff}$$

$$\mathit{Thermostats} ::= \mathit{aboveTemp} \mid \mathit{belowTemp}$$

$$\begin{array}{|l}
\hline
\quad \mathit{ModeOperation} \\
\mathit{ticks?} : \mathit{Times} \\
\mathit{mode?} : \mathit{Modes} \\
\mathit{thermostat?} : \mathit{Thermostats} \\
\mathit{heating!} : \mathit{Settings} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\quad \mathit{ModeOff} \\
\mathit{ModeOperation} \\
\hline
\mathit{mode?} = \mathit{off} \\
\mathit{heating!} = \mathit{isOff} \\
\hline
\end{array}$$

ModeContinuous

ModeOperation

$$mode? = continuous$$
$$(heating! = isOn \land thermostat? = belowTemp) \lor$$
$$(heating! = isOff \land thermostat? = aboveTemp)$$

ModeTimed

ModeOperation

$$mode? = timed$$
$$(heating! = isOn \land thermostat? = belowTemp \land ticks? > OnTime \land ticks? < OffTime) \lor$$
$$(heating! = isOff \land \neg(thermostat? = belowTemp \land ticks? > OnTime \land ticks? < OffTime))$$

The entire operation can then be described by the schema disjunction

$$PerformModeOperation \equiv ModeOff \lor ModeContinuous \lor ModeTimed$$

8.11.2 The postcondition

From the specification it is clear that the post condition will consist of three disjunctions: one each for *ModeOff, ModeContinuous* and *ModeTimed*. We also have to consider the refinements used in producing SPARK equivalents to elements of the Z specification. For example, the variable `AboveTemp` is the refinement of the Z *thermostat? = aboveTemp.* By this process we can arrive at the postcondition for the *PerformModeOperation* operation which is

```
--# post
--#   (Mode = ModeSwitch.off and not Heating) or
--#   (Mode = ModeSwitch.cont and (Heating <-> not AboveTemp)) or
--#   (Mode = ModeSwitch.timed and
--#     (Heating <->
--#       (not AboveTemp and Time > OnTime and Time < OffTime)));
```

8.11.3 The code

The annotated code for the heating controller can now be completed.

```
-------------------------------------------------------
-- 1.  INTERFACES TO THE REAL WORLD                  --
-------------------------------------------------------

package Clock
--# own in TimeSeq;
is

  subtype Times is Integer range 0 .. 86399;

  procedure Read (T: out Times);
  --# global in TimeSeq;
  --# derives T from TimeSeq;
  -- return the current time

end Clock;


package Thermostat
--# own in TempSeq;
is

  procedure Read (AboveTemp: out Boolean);
  --# global in TempSeq;
  --# derives AboveTemp from TempSeq;
  -- AboveTemp is true iff temperature is >= required level

end Thermostat;


package ModeSwitch
--# own in ModeSwitchSeq;
is

  type Modes is (off, cont, timed);

  procedure Read (M: out Modes);
  --# global in ModeSwitchSeq;
  --# derives M from ModeSwitchSeq;
  -- reads setting of ModeSwitch and returns its current state

end ModeSwitch;

package Actuator
--# own out ActSeq;
is
```

```
procedure Write(TurnOn: in Boolean);
  --# global out ActSeq;
  --# derives ActSeq from TurnOn;
  -- activates heating if true passed in, deactivates if false
end Actuator;


  with Thermostat,
       Actuator,
       Clock,
       ModeSwitch;
  --# inherit Thermostat,
  --#         Actuator,
  --#         Clock,
  --#         ModeSwitch;
  --# main_program
  procedure HeatingSystem
  --# global     out Actuator.ActSeq;
  --#         in     Thermostat.TempSeq,
  --#                Clock.TimeSeq,
  --#                ModeSwitch.ModeSwitchSeq;
  --# derives Actuator.ActSeq           from Thermostat.TempSeq,
  --#                                        Clock.TimeSeq,
  --#                                        ModeSwitch.ModeSwitchSeq;
  is
    OnTime  : constant Clock.Times := 32400;
    OffTime : constant Clock.Times := 61200;

    OldHeating : Boolean := False;  -- initial value from Actuator package
    Heating    : Boolean;

    Mode: ModeSwitch.Modes;
    AboveTemp: Boolean;
    Time: Clock.Times;

    -------------------------------------------------------
    -- 2.  IMPLEMENTATION OF THE Z SPECIFICATION         --
    -------------------------------------------------------

    function IsInOperatingPeriod(Time : Clock.Times) return Boolean
    --# return Time > OnTime and Time < OffTime;
    is
    begin
      return Time > OnTime and Time < OffTime;
    end IsInOperatingPeriod;

    procedure PerformModeOperation
    --# global  out Heating; in Mode, AboveTemp, Time;
    --# derives Heating from Mode, AboveTemp, Time;
    --# post
    --#   (Mode = ModeSwitch.off and not Heating) or
    --#   (Mode = ModeSwitch.cont and (Heating <-> not AboveTemp)) or
    --#   (Mode = ModeSwitch.timed and
    --#      (Heating <->
    --#        (not AboveTemp and Time>OnTime and Time<OffTime)));
    is
    begin
      case Mode is
        when ModeSwitch.off   =>
           Heating := False;
        when ModeSwitch.cont  =>
           Heating := not AboveTemp;
        when ModeSwitch.timed =>
           Heating := not AboveTemp and
             IsInOperatingPeriod(Time);
      end case;
    end PerformModeOperation;

  begin -- HeatingSystem
    loop
      Clock.Read(Time);
      Thermostat.Read(AboveTemp);
      ModeSwitch.Read(Mode);
      PerformModeOperation;
      if Heating /= OldHeating then
        Actuator.Write(Heating);
      end if;
      OldHeating := Heating;
    end loop;
  end HeatingSystem;
```

8.11.4 Verification conditions

The verification conditions for `PerformModeOperation` are as follows.

```
For path(s) from start to finish:

procedure_performmodeoperation_1.
H1:   true .
H2:   mode >= modeswitch__modes__first .
H3:   mode <= modeswitch__modes__last .
H4:   true .
H5:   time >= clock__times__first .
H6:   time <= clock__times__last .
H7:   mode = modeswitch__off .
       ->
C1:   ((mode = modeswitch__off) and (not false)) or (((
          mode = modeswitch__cont) and (false <-> (not
          abovetemp))) or ((mode = modeswitch__timed) and (
          false <-> ((not abovetemp) and ((time > ontime) and (
          time < offtime)))))) .

procedure_performmodeoperation_2.
H1:   true .
H2:   mode >= modeswitch__modes__first .
```

```
H3:   mode <= modeswitch__modes__last .
H4:   true .
H5:   time >= clock__times__first .
H6:   time <= clock__times__last .
H7:   mode = modeswitch__cont .
       ->
C1:   ((mode = modeswitch__off) and (not (not abovetemp))) or (((
          mode = modeswitch__cont) and ((not abovetemp) <-> (not
          abovetemp))) or ((mode = modeswitch__timed) and ((not
          abovetemp) <-> ((not abovetemp) and ((time >
          ontime) and (time < offtime)))))) .


procedure_performmodeoperation_3.
H1:   true .
H2:   mode >= modeswitch__modes__first .
H3:   mode <= modeswitch__modes__last .
H4:   true .
H5:   time >= clock__times__first .
H6:   time <= clock__times__last .
H7:   mode = modeswitch__timed .
       ->
C1:   ((mode = modeswitch__off) and (not ((not abovetemp) and
          isinoperatingperiod(time)))) or (((mode =
          modeswitch__cont) and (((not abovetemp) and
          isinoperatingperiod(time)) <-> (not abovetemp))) or ((
          mode = modeswitch__timed) and (((not abovetemp) and
          isinoperatingperiod(time)) <-> ((not abovetemp) and ((
          time > ontime) and (time < offtime)))))) .
```

8.11.5 Simplified VCs

The VCs can be extensively simplified by the SPADE Automatic Simplifier.

```
For path(s) from start to finish:

procedure_performmodeoperation_1.
*** true .          /* all conclusions proved */


procedure_performmodeoperation_2.
*** true .          /* all conclusions proved */


procedure_performmodeoperation_3.
H1:    time >= 0 .
H2:    time <= 86399 .
       ->
C1:    not abovetemp and isinoperatingperiod(time) <-> not
           abovetemp and (time > 32400 and time < 61200) .
```

Only the third VC is not proved by the Simplifier; this is because the code implements the check for being within the operating period by means of a call to the function `IsInOperatingPeriod` rather by direct inequalities as used in the specification. Within the Proof Checker, substitution for this function with a suitable proof rule would be enough to complete the proof.

8.11.6 The use of proof functions

To illustrate a use of proof functions, the postcondition used above has here been expressed in terms of the following proof function.

```
--# function IntendedSetting(T   : Clock.Times;
--#                          M   : ModeSwitch.Modes;
--#                          Hot : Boolean) return Boolean;
```

The postcondition then becomes

```
--# post Heating <-> IntendedSetting(Time, Mode, AboveTemp);
```

The VCs generated from this revised code are as follows.

```
For path(s) from start to finish:

procedure_performmodeoperation_1.
H1:   true .
H2:   mode >= modeswitch__modes__first .
H3:   mode <= modeswitch__modes__last .
H4:   true .
H5:   time >= clock__times__first .
H6:   time <= clock__times__last .
H7:   mode = modeswitch__off .
       ->
C1:   false <-> intendedsetting(time, mode, abovetemp) .


procedure_performmodeoperation_2.
H1:   true .
H2:   mode >= modeswitch__modes__first .
H3:   mode <= modeswitch__modes__last .
H4:   true .
H5:   time >= clock__times__first .
H6:   time <= clock__times__last .
H7:   mode = modeswitch__cont .
       ->
C1:   (not abovetemp) <-> intendedsetting(time, mode,
          abovetemp) .


procedure_performmodeoperation_3.
H1:    true .
H2:    mode >= modeswitch__modes__first .
H3:    mode <= modeswitch__modes__last .
H4:    true .
```

```
H5:    time >= clock__times__first .
H6:    time <= clock__times__last .
H7:    mode = modeswitch__timed .
        ->
C1:    ((not abovetemp) and
           isinoperatingperiod(time)) <->
           intendedsetting(time, mode, abovetemp) .
```

Proof of these VCs would require use of the Proof Checker and establishment of suitable substitution rules for the proof function. Informally, the hypotheses can be read as a traversal condition for a particular path and the conclusion as an action. Consider the first VC: this can be read as "when the mode switch is off, the code can be regarded as correct provided the intended heating setting under these circumstances is false".

This approach allows an interrogative use of the VC Generator: suitable proof functions, for the exports of interest, can be determined from the derives annotation and used to write a skeletal postcondition. The VCs generated can then be interpreted as above.

For example:

```
--# derives X from X, Y &
--#         Y from Y &
--#         Z from X, Y, Z;
```

would suggest proof functions:

```
--# function Xout(X : Xtype; Y : Ytype) return Xtype;
--# function Yout(Y : Ytype) return Ytype;
--# function Zout(X : Xtype; Y : Ytype; Z : Ztype) return
      Ztype;
```

allowing construction of the following postcondition:

```
--# post X = Xout(X~, Y~) and
--#      Y = Yout(Y~)     and
--#      Z = Zout(X~, Y~, Z~);
```

(the ~ is needed because X, Y and Z are all both imported and exported).

None of this retrospective delving is a substitute for first specifying the code, then asserting (through proof contexts) its behaviour and proving it; however, it can provide a more clinical investigative tool than provided by path function generation. For example, there may be circumstances in which we wish to investigate the behaviour of a subset of the exports of a subprogram. Consider the procedure `SumAndDiff` which returns the `sum and difference of two integer numbers.`

```
procedure SumAndDiff(X, Y : in     Integer;
                     S, D :    out Integer)
--# derives S, D from X, Y;
is
begin
  S := X + Y;
  D := X - Y;
end SumAndDiff;
```

If we were just interested in the behaviour of S we could define a proof function

```
--# function Sout (X, Y : Integer) return Integer;
```

and use it to specify a post condition

```
--# post    S = Sout(X, Y);
```

VCs can now be generated which we may use to enhance our understanding of how export S behaves (but which tell us nothing about D).

```
For path(s) from start to finish:

procedure_sumanddiff_1.
H1:    true .
H2:    x >= integer__first .
H3:    x <= integer__last .
H4:    y >= integer__first .
H5:    y <= integer__last .
        ->
C1:    x + y = sout(x, y) .
```

Document Control and References

Praxis High Integrity Systems Limited, 20 Manvers Street, Bath BA1 1PX, UK.

File under

$CVSROOT/userdocs/Examiner_GenVCs.doc (was previously S.P0468.73.31)

Changes history

Issue 0.1: (18th September 1995) Initial version

Issue 0.2: (6th October 1995) After review by ION and JAH

Issue 0.3: (1st November 1995) Conversion to Praxis Standard Document by SGB

Issue 0.4: (8th November 1995) Incorporation of final comments by ION

Issue 2.0: (1st July 1997) Conversion to new standard document format

Issue 2.1: (27th August 1997) Updated for Examiner Release 3.0, made provisional

Issue 3.0: (2nd September 1997) After formal review

Issue 3.1: (21st June 2000) Draft update for Examiner Release 5.0

Issue 4.0: (10th July 2000) Updated to definitive after review.

Issue 4.1: (10th July 2001) Draft update for Release 6.0

Issue 6.0: (24th August 2001) After review by JARH

Issue 6.01: (6th February 2002) CFR 993—Added documentation of type assertion annotation

Issue 6.02: (24th May 2002) Description of FDL representation of tagged records

Issue 6.10: (24th June 2002) Updates resulting from review S.60468.79.77

Issue 6.11: (29th October 2002) Addition of Liskov-Wing rules

Issue 6.2: (30th October 2002) After review

Issue 7.0: (14th April 2003) Updated to new template format

Issue 8.0: (29th May 2003) Changes to new template, final format

Issue 8.1: (3rd Jun 2003) Addition of unconstrained formal parameters, concurrency issues and final tidy up.

Issue 8.2: (29th April 2004) Addition of % operator.

Issue 8.3: (9th July 2004) Describe private type refinement.

Issue 8.4: (2nd November 2004) Describe composite constant rule policy.

Issue 8.5: (23rd November 2004) Change company name only.

Issue 8.6: (5th January 2005) Definitive issue following review S.P0468.79.88.

Issue 8.7: (11th February 2005) Added note that explicit range is not permitted in quantified expressions over type Boolean. (SEPR 1779).

Issue 8.8: (22nd November 2005) Changed Line Manager.

Issue 8.9: (21st August 2008): Added “real” as an FDL reserved identifier (SEPR 2437)

Issue 8.10: (7th January 2009): Factored out contact details.

Issue 8.11: (2nd February 2009): Modify copyright notice.

Changes forecast

Nil.

Document references

1 SPARK Examiner User Manual

2 Generation of RTCs for SPARK Programs