p

Document Set	SPARK 95		Reference	SPARK 95
Title	:	SPARK 95 - The SPADE Ada 95 Kernel (excluding RavenSPARK)		
Synopsis	:			
File Under	:	CVSROOT/userdocs/SPARK95.doc (was S.P0468.73.62)		
Contents	:	I THE RATIONALE OF SPARK II SPECIFICATION OF SPARK III COLLECTED SYNTAX OF SPARK		
Status	:	Definitive		
Issue Number	:	4.7		
Date	:	23 October 2006		
Copied To	:	On demand		
Front Sheet	:			
Originators	:	Rod Chapman, Peter Amey	**Signed**	:
Approver	:	SPARK Team Line Manager	**Signed**	:

Document Control and References

Changes history

Issue 0.1: (9th March 1999) Draft for review.

Issue 0.2: (21st October 1999) Updated draft after informal review.

Issue 1.0: (28th October 1999) Changes and up-issue to definitive after formal review.

Issue 1.1: (11th September 2001) First draft including all changes for Examiner release 6.0

Issue 2.0: (11th October 2001) After review

Issue 2.1: (9th May 2001) Description of tagged types and predefined packages.

Issue 2.2: (15th May 2002) Allow modular subtypes.

Issue 2.3 (20th June 2002) Add tagged types to preface.

Issue 3.0 (2nd July 2002) Updated after review actions

Issue 3.1 (28th October 2002) Updated after further changes to tagged types and review actions

Issue 3.5 (5th March 2003) Preparative work for RavenSPARK and other Release 7.0 features

Issue 3.6 (2nd April 2003) Cross-check grammar with RavenSPARK user manual and other updates.

Issue 4.0 (29th May 2003) After final review actions

Issue 4.09 (15th August 2003) Relaxation of the entire variable rule for exported array elements

Issue 4.1 (23rd October 2003) Minor corrections after review

Issue 4.2 (23rd November 2004) Allow full-range non-tagged record subtypes and instantiation of Unchecked_Conversion.

Issue 4.3 (5th January 2005) Issued at Definitive following review S.P0468.79.88.

Issue 4.4 (5th July 2005): Allow user-defined hidden exception handlers.

Issue 4.5 (17th October 2005): Clarify rules for ‘Succ and ‘Pred.

Issue 4.6 (1st December 2005): Changed Line Manager and added new Preface.

Issue 4.7 (23rd October 2006): Added obsolescent Ada 83 floating-point attributes as implementation-defined, and documented in Annexes J and K.

Changes forecast

None

p

SPARK 95 - The SPADE Ada 95 Kernel (excluding RavenSPARK)

Edition 4.7

Praxis High Integrity Systems
20, Manvers St.
Bath BA1 1PX

October 2006
SPARK 95

PREFACE TO THE FIRST EDITION

This document describes an annotated sublanguage of Ada 95, intended for use in safety-critical applications.

SPARK 95 is described here in terms of the complete Ada 95 language: this document is intended to be read in conjunction with the International Standard "Ada 95 Reference Manual", ANSI/ISO/IEC 8652, and in Part 2 of this document, the section numbers correspond to those of the Ada 95 manual. The document is not intended to be a tutorial on the SPARK language, as this purpose is admirably served by the book by John Barnes (Barnes, 2003).

Following the overview of SPARK in Part 1, Part 2 catalogues the differences between SPARK 95 and Ada 95. Part 3 gives the collected syntax of SPARK 95, laid out in a manner which facilitates its comparison with the Ada syntax. Throughout this document, a marginal marking * signals a modification of an Ada syntax rule, and the marking + indicates that a syntax rule belongs to SPARK only.

The first version of SPARK (based on Ada 83) was produced at the University of Southampton (with MoD sponsorship) by Bernard Carré and Trevor Jennings. Subsequently the language was progressively extended and refined, first by Program Validation Limited and then by Praxis Critical Systems Ltd.

The authors welcome comments on SPARK from all interested parties.

Gavin Finnie

Praxis Critical Systems, October 1999

PREFACE TO THE SECOND EDITION

The second edition of this document accompanies release 6.0 of the SPARK Examiner. The well-established policy of making incremental and backwards-compatible enhancements to the SPARK language has been followed with this release. There are changes both to the compilable core of SPARK and its annotation language; the latter has been extended to simplify the description of interactions between a SPARK program and its external environment.

We continue to welcome comments on SPARK from all interested parties

SUMMARY OF MODIFICATIONS LEADING TO THE SECOND EDITION

Modular types (Section 3.5.4) Modular types are now included in SPARK with 3 restrictions:

1 The Modulus of a type must be a positive power of 2.

2 Subtypes of modular types are not permitted.

3 Unary arithmetic operators are not permitted.

Exit statements and loop labels (Section 5.7) Loop statement identifiers may now appear in exit statements; however, the restriction that the exit may apply only to the most closely enclosing loop remains.

Global modes on function subprograms (Section 6.1.2) For consistency with procedure globals and with parameters, the mode **in** may now appear in function global annotations.

Predefined types (Section A.1) The following types are now regarded as predefined in package Standard: Duration, Long_Integer and Long_Float. The latter two definitions are for the convenience of users whose compiler also provides them.

External Variables (Section 7) Modes **in** or **out** can now optionally appear in own variable and refinement clauses. The presence of a mode indicates that the own variable is regarded as providing a channel of communication between the SPARK program and its environment. Such variables are called *external variables*. External variables are treated as being *volatile* (i.e. referenced values may change without an intervening update and repeated updates are not regarded as ineffective). The use of external variables greatly simplifies the capture of desired system behaviour in SPARK annotations.

Null Derives (Section 6.1.2) A new form of the derives annotation can be used to show that no export within the visible part of a SPARK program is derived from the imports of that subprogram. For example: `--# derives null from X, Y, Z;` The null derives form is especially useful in conjunction with external variables of mode **in**.

Peter Amey

Praxis Critical Systems, September 2001

PREFACE TO THE THIRD EDITION

The third edition of this document accompanies release 6.2 of the SPARK Examiner.

We continue to welcome comments on SPARK from all interested parties

SUMMARY OF MODIFICATIONS LEADING TO THE THIRD EDITION

Tagged types (section 3.9) These are now permitted in SPARK under certain restrictions.

Modular types (Section 3.5.4) Subtypes of modular types are now permitted.

Type assertion annotation A new class of annotation—the *type assertion*—has been introduced with this release. This annotation allows the base type of a signed integer type declaration to be indicated to the Examiner. This supplies additional useful information to the Examiner when generating VCs to show the absence of Overflow_Check.

Configuration file The configuration file is a new mechanism, which replaces the existing target-data file mechanism, that allows the detail of packages Standard and System to be given to the Examiner.

Roderick Chapman

Praxis Critical Systems, October 2002

PREFACE TO THE FOURTH EDITION

The fourth edition of this document accompanies Release 7.1 of the SPARK Examiner. Release 7.0, and later versions, provide, for the first time, support for concurrent programming in SPARK. Full details of the concurrency extensions to SPARK are described in the manual SPARK - The SPADE Ada 95 Kernel (including RavenSPARK). Concurrency features are not included in this manual which describes only sequential SPARK.. We continue to welcome comments on SPARK from all interested parties.

SUMMARY OF MODIFICATIONS LEADING TO THE FOURTH EDITION

Most of the effort involved in Release 7.0 of the SPARK Examiner has been focussed on implementing the Ravenscar Profile to support concurrent programming in SPARK; there were no changes to the core sequential SPARK language. Release 7.1 provides private subprograms and relaxes restrictions on the use of array elements as actual parameters.

Some modest Examiner changes have been made as follows:

Duration. Duration was initially not a predefined identifier in SPARK because the absence of any form of tasking made it irrelevent. It was later added to the language at the request of some users; unfortunately, this proved a problem for other users who were re-using the identifier for other purposes. To satisfy both groups, the predefinition of Duration is now controlled by a command line switch.

Proof involving unconstrained parameters. Significant improvements have been made to proof involving calls to subprograms with unconstrained formal parameters.

Full details of changes leading to Release 7.1 of the Examiner can be found in the Examiner Release Note.

Peter Amey

Praxis Critical Systems, October 2003

PREFACE TO EDITION 4.3

Edition 4.3 of this document accompanies release 7.2 of the SPARK Examiner.

SUMMARY OF MODIFICATIONS LEADING TO EDITION 4.3

Full-range subtypes of non-tagged records are now allowed in SPARK.

Declarations of constants of type String are now allowed in SPARK without requiring a declaration of a constraining string subtype.

Instantiations of the predefined generic function Unchecked_Conversion are now allowed in SPARK.

Some significant improvements to the Examiner have been made with this release:

The VC Generator has been improved to generate hypotheses for local variables being within their designated subtype. VC Generation of **for** loops that have a dynamic range has also been implemented. Finally, the Examiner can generate proof rules for composite constants under the control of both a new command-line switch and a new annotation. Please see the release 7.2 release note for more details of these, and other, changes.

Rod Chapman

Praxis High Integrity Systems, December 2004

PREFACE TO EDITION 4.6

Edition 4.6 of this document accompanies release 7.3 of the SPARK Examiner.

SUMMARY OF MODIFICATIONS LEADING TO EDITION 4.6

SPARK now allows a body to have a hidden exception handler part.

The rules regarding the use of the ‘Succ and ‘Pred attributes have been clarified.

Significant improvements to the Examiner and Simplifier are included with this release. For a summary, please see the accompanying toolset Release Note.

Rod Chapman

Praxis High Integrity Systems, December 2005

SUMMARY OF MODIFICATIONS LEADING TO EDITION 4.7

SPARK now allows the use of obsolete Ada 83 floating-point attributes in SPARK 95 mode.

Robin Neatherway

Praxis High Integrity Systems, October 2006

Contents

I The Rationale of SPARK

[Here we present unchanged the Rationale from the original SPARK[1] Report. Subsequent development of SPARK has rendered certain parts of it inaccurate, and these are indicated in this Edition by footnotes referring to additional notes which immediately follow the Rationale.]

> " It is not too late! I believe that by careful pruning of the Ada language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic in use. The sponsors of the language have declared unequivocally, however, that there shall be no subsets. This is the strangest paradox of the whole strange project. If you want a language with no subsets, you must make it *small.* "
>
> *From Professor Hoare's 1980 ACM Turing Award Lecture*.

1 TERMS OF REFERENCE

The designers of programming languages are presented with many, often conflicting, requirements; support for high-integrity programming is only one of them. As an extreme example, in the case of 'C' - aimed at convenience of use and efficiency for low-level systems programming - we must suppose that safety was not a major preoccupation. The design of Ada was obviously more professional, but its expressive power and generality were only achieved at great cost in complexity.

Here our requirements of a programming language are quite limited, in terms of its applicability, but very strict. We are mainly concerned with software to perform system control functions. The integrity of the software is vital: it must be verifiable. We can assume that the programs are to be developed by professionals, supported by whatever tools are available, and that if necessary substantial resources will be expended in achieving high integrity of software prior to its application; but the problems involved in proving its fitness of purpose must be tractable, in practical terms.

We assume that software is to be developed systematically, through the construction of the following objects:

- A definition of requirements.
- A program specification.
- A program design.
- A program text, written in the chosen high-order language.
- A translation of this text into binary code, for a particular processor.

---

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

It should be demonstrable — by logical reasoning — that each object here is functionally consistent with its "parent": that the specification meets the defined requirements, that the design conforms to the specification, and so on.

Some issues — such as the problems of requirements capture, the well-foundedness of different specification methods and their applicability — cannot be addressed here, although of course they are quite as important as the rest. But clearly, to assess the value of a programming language for safety-critical work we must consider much more than the possible abuses of **goto** statements and pointers. The extent to which a formal specification, in VDM or 'Z' for instance, can be steered towards a design appropriate for implementation in the language, and the ease with which a design can be refined into program code, are important - both to help obtain the functional consistency we require, and to facilitate its verification. Well-tried, effective tools must exist to support program development, and a trustworthy compiler is essential.

As well as these logical considerations we also have sociological ones: the general intelligibility of the language, the size of the community of its users, their mastery of the technology at their disposal. All these matters must eventually be taken into account.

2 THE NEED FOR A "SAFE SUBSET" OF ADA

The Ada language *in its complete form* is not suitable for rigorous program development, for two closely-related reasons:

Inadequacy of its definition: The first requirement of a language for rigorous programming is that its definition be precise, and logically coherent. The language itself must not contain any ambiguities which would allow the construction of programs of uncertain meaning. The definition must also be complete.

In general the initial conception of a programming language is largely informal, and usually its first definition is not entirely coherent. However, if the language has sufficient merit it may undergo a process of refinement. The discovery of its deficiencies and the best ways of overcoming them may come partly through practical experience of using it, partly through attempts to construct its formal definition. But for high-integrity work a formal definition of the language must eventually be established, as the essential basis of its rigorous use.

The official definition of Ada is not entirely clear or logically consistent; despite the enormous importance attached to the language, it still has serious defects (McGettrick, 1982; Goodenough, 1987). A large amount of work has been done on the formal semantics of Ada, ever since 1980 (Bjorner and Oest, 1980), but as far as we know no satisfactory formal definition has yet been completed. For these reasons formal verification may be impossible, and we can never be sure of the integrity of compiled code.

Excessive complexity: We believe that, in trying to shift the burden of programming from the programmer to the compiler as far as possible, the designers of Ada have been much too ambitious. Even if all the problems of logical coherence of the language were overcome in some way, the programming extravagances which it allows would still make correctness proofs very difficult, even impossible to establish in practice. We again quote from Hoare's Turing Award Lecture (Hoare, 1980):

"The original objectives of the language included reliability, readability of programs, formality of language definition, and even simplicity. Gradually these objectives have been sacrificed in favour of power, supposedly achieved by a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. We relive the history of the design of the motor car. Gadgets and glitter prevail over fundamental concerns for safety and economy."

These problems are obviously related: it is the richness of Ada which makes its formalisation so difficult. Even if a sound formal definition were produced, the language it defined could not be the Ada of the Reference Manual, since the latter has logical defects. And although formalisation might move some of the arguments about the language onto a more logical terrain, it could not resolve them satisfactorily: the enormous complexity of the formal definition would preclude the social processes essential to its justification and refinement (DeMillo et al., 1979).

Unfortunately we cannot expect Ada to evolve significantly in the direction we would wish. Any features, once offered, are hard to take away. Besides, there is a major inhibiting factor, well summarised by Goodenough (1987):

"Almost all languages go through revisions as a result of initial implementation. Eventually languages tend to converge to a fairly standard interpretation which gets enshrined in a standard. However in the case of Ada we had the standard almost before the implementations, so that problems are coming to light after standardisation rather than before."

The Ada Language Maintenance Committee from time to time approves "Ada commentaries" which attempt to resolve "issues" arising from the Reference Manual; a few of these effectively make small changes to the official definition. The language is to undergo a major review in 1988, but in view of the enormous investments which have already been made and the Language Maintenance Committee's policies to date, we cannot anticipate any radical simplifications. For this reason we do not believe that the *complete* Ada language will ever be appropriate for safety-critical programming.

On a much more positive note, Ada has some very desirable features, not possessed by any other language likely to have widespread use. The designers of Ada were greatly influenced by experience with Pascal and its derivatives, and had the advantage of hindsight. At the centre of Ada is the core of Pascal, with some minor but nevertheless valuable improvements (for example in the form of the **case** and iterative constructs). Around this Ada has features, employed in Euclid and Modula for instance, to allow data abstraction and facilitate systematic program design. Some of these (with certain restrictions), notably

- packages,
- private types,
- functions with structured values,
- the library system,

we regard as essential extensions to Pascal, for the rigorous construction of large programs from their specifications.

The question which naturally arises is whether it is possible to extract from the complete language a logically coherent "kernel", containing those features we require and no more. Sometimes, when a language is inappropriate for high-integrity work, its foundations are so insecure that the search for a

safe subset would be pointless; this applies to FORTRAN and 'C' for instance. However we believe that the core of Ada is sound. The contentious issues, the complexity and impediments to formal definition stem from its more advanced features - such as tasks and exception-handling, to name the most problematic. As confirmation of this, by 1980 formal definitions had already been produced for subsets of Ada (Bundgaard and Schultz, 1980; Pederson, 1980), which contained all but its most troublesome features (principally separate compilation, generics, tasks and exceptions). Even in this work, *"the main problem in defining the static semantics turned out to be the handling of derived subprograms and the arranging of a proper model of the scope of predefined operators"*.

We see little merit in derived types[2], and consider that overloading of all kinds should be avoided as far as possible. The following sections describe a kernel which we believe to be coherent, and whose formal definition should be very much simpler even than the 1980 subset definitions.

3 THE DEVELOPMENT OF SPARK - GENERAL STRATEGY

SPADE-Pascal was developed essentially by excising from ISO-Pascal those features which were considered particularly "dangerous", or which could give rise to intractable validation problems - such as variant records, and the use of functions and procedures as parameters - and then resolving the remaining difficulties by introducing "annotations" (formal comments). A different strategy had to be followed here, for two reasons. Firstly, whereas ISO-Pascal is a small language, which we had mostly wanted to retain, Ada is very large and we wished to prune it severely. Secondly, whereas the formal basis of Pascal had been established, and its defects catalogued in a number of published papers, Ada is less well understood and the flaws in the language have not been delineated as precisely. It therefore seemed essential to adopt a "constructive" approach initially, sketching out a kernel of required features rather than excising unsatisfactory constructs one by one.

To obtain the expressive power we required, without introducing unnecessary complexity, it was decided that we should aim to adopt essentially the Pascal core of Ada (although of course the Pascal features are not precisely matched in Ada — in fact Ada improves on some of them), supplemented by the Ada features mentioned above which support systematic program development (principally packages, private types, functions with structured values and the library system). This subset was then refined, by (a) imposing a number of restrictions, and (b) incorporating a system of annotations, somewhat similar in form to the annotations of SPADE-Pascal (Carré and Debney, 1985) and Anna - the language for annotating Ada programs developed by Luckham *et al* (1987). The following section discusses in some detail our criteria, in the refinement of this subset.

4 CONSIDERATIONS IN THE REFINEMENT OF THE ADA SUBSET

4.1 Logical Soundness

The need for a sound language definition has already been stressed. It is for this reason that from the outset we excluded the use of Ada **tasks**[3]**,** whose proper formal definition has not yet been achieved.

---

[2] See Additional Note 7

[3] See Additional Note 8

Without a calculus to reason about Ada tasking - which allows extremely complex interactions between concurrent processes, with a high degree of non-determinism - it should not be employed in safety-critical systems.

The meaning of a program must be completely determined by its text; it must not be affected by the manner in which the program is compiled. As an example of a violation of this rule, in full Ada different legal **orders of elaboration of compilation units** can give different results. (This particular problem is overcome in SPARK by the restrictions imposed on variable definitions within packages and on initialization expressions.) Unfortunately, ambiguities are often due not to particular features, but to their use in combination. We have therefore not attempted to catalogue the "problem areas", but we believe that the language simplifications made below - for a variety of reasons - together eliminate all the problems of logical coherence.

4.2 Complexity of Formal Language Definition

We attach great importance to complexity of formal definition, as a basis for accepting or discarding language features, because it is a good indicator of the difficulty of reasoning about programs (as opposed to an informal language definition, which can be of beguiling simplicity). The complexity of the formal definition of a language also directly determines the complexity of its support tools, such as compilers, which should themselves be error-free; in choosing our subset of Ada, we have aimed to reduce its complexity to such a level that the construction of a correct (formally verified) compiler would be technically feasible in a reasonable period of time - though in our opinion it would still be a major undertaking.

Our ultimate concern of course is the fitness of purpose of the binary code version of a program, executed on a chip. If its integrity is vital it would be most unwise to place complete confidence in the compiler which produced it, however carefully the compiler was written. Another argument for simplicity of the high-level language therefore is the need for a simple correspondence between program source code and its compiled version, to allow correctness of the latter to be checked. (We envisage that a compiler employed for safety-critical work will have, as part of its documentation, a precise definition of the mapping which it performs; and that the code which it produces will be "instrumented", with formal comments, to facilitate its verification. Compilers of this kind are not yet available, but some are being developed; the potential simplicity of mappings to binary was therefore taken into account in designing SPARK.)

Analysis of compiled code may be necessary not only because the translation of program source code is unreliable but because a program may contain implementation-dependent features, in particular **address clauses** or **machine code insertions,** whose analysis is outside our province but which may be erroneous. (The SPARK Examiner will accept address clauses, but issue warning messages when it encounters them. It will also accept procedures consisting of machine code, if their specifications contain the required descriptive annotations - see Section 6 of Part 2 - but again it will warn the user of their presence.)

For these reasons we considered the following features of Ada to be undesirable, in the context of safety-critical programming.

Exceptions, designed for dealing with errors or other exceptional situations, might at first sight seem very desirable for safety-critical programming. However, it is easier and more satisfactory to write a

program which is exception-free, and prove it to be so, than to prove that the corrective action performed by exception-handlers would be appropriate under all possible circumstances[4]; if checks with recovery actions are required in a program, these can be introduced without resorting to exception handlers, and in general the embedded code will be easier to verify. Since exception-handling also seriously complicates the formal definition of even the sequential part of Ada we believe it should be omitted.

The concept of **generic units** is an interesting one, which does find significant applications in the Ada Input-Output library. However, it is another feature which seriously complicates the formal definition of Ada. Also, the code re-usability which it aims to provide is not achieved as easily as one might imagine: it is still necessary to prove correctness of every instantiation of a generic object. The proofs may be simplified by first establishing some properties of the generic object in abstract terms (assuming for instance that the operators which it employs obey certain axioms), and then showing that each instantiation is a valid concrete interpretation. But if the generic unit is non-trivial, the required proofs may remain non-trivial also. Furthermore, generics cause overloading, which we are anxious to avoid. We do not believe that, in our application area, the complexity introduced by generic units is justified.

As was mentioned earlier, **derived types**[5] which involve the implicit declaration of user-defined subprograms seriously complicate the formal definition of Ada, and cause overloading. SPARK does not allow the use of derived types other than integer and real types.

All Ada features which require *dynamic storage allocation* were ruled out, for several reasons. The specification and modelling of **access type** manipulations, which can involve aliasing, is extremely difficult; for programs using access types it may also be very difficult to achieve security (i.e. the detection of all language violations - see Section 4.5 below), let alone verification. Other features which require dynamic storage allocation such as **dynamically constrained arrays, discriminants** and **recursion** may be less troublesome in this regard, but quite generally, dynamic storage allocation makes the problem of verifying compiled code impossibly difficult: for this a simple correspondence between program variables and memory addresses is essential. The use of dynamic storage allocation is also dangerous in that it is always very difficult, and usually impossible, to establish memory requirements. In SPARK all constraints are statically determinable.

A number of other features of minor importance have also been removed, which incur a penalty in complexity simply to support lazy programming: SPARK does not allow the use of **default expressions** of **subprogram parameters,** or **mixing positional and named parameters within the same subprogram call. Default expressions** of **record components** are also banned.

So far we have considered only the reduction of complexity by the piece-meal elimination of language features. Further simplifications rely on the use of *annotations* (whose consistency with program code will be checked by SPADE tools).

Ada's **scope, visibility and overloading rules** are extremely complicated; they are very likely to cause confusion to the programmer and they impede verification quite unnecessarily. In SPARK the notions of scope and visibility are much simpler, the rules concerning the use and reuse of identifiers

---

[4] See Additional Note 1

[5] See Additional Note 7

being essentially those of SPADE-Pascal. To make this possible we have banned **overloading of character literals, enumeration literals and subprograms, block statements** and **use clauses,** and restricted the application of **renaming declarations.**

Any Ada feature which does not appear in SPARK can still be employed if the subprogram which makes use of it has its body hidden by means of a "hide" directive - see Section 6.3 of Part 2. It is possible to model calls of hidden subprograms by employing their annotations (which are mandatory), but it is impossible to check consistency of these annotations with the code implementation until code is revealed. The facility for hiding subprogram bodies is intended to be used principally to support top-down program design; its use to hide undesirable features is obviously not recommended. The SPARK Examiner issues warning messages whenever it encounters hidden subprogram bodies.

4.3 Expressive Power

The systematic development of a sizeable program involves decomposition of the programming problem, based on the recognition of useful abstractions. Most familiar is the decomposition of a programming problem into subproblems, each to be solved separately by independent functional units, through *procedural abstraction*.

Using *specifications* of the procedures for solving the subproblems, we can change the level of detail to be considered when we wish to combine them. In effect, procedural abstraction extends the virtual machine defined by a programming language by adding to it new *operations*.

Less familiar perhaps, but quite as important to software development, is *data abstraction* — the addition to the virtual machine of new kinds of *data objects*, together with operations to create, modify, insert and extract information from those objects.

Pascal supports procedural abstraction but not data abstraction. Ada offers a useful improvement to Pascal's facility for procedural abstraction (by allowing function subprograms to return structured objects), and it supports data abstraction through **packages** and **private types.** This is a most important contribution to safety in programming, whose inclusion we considered essential.

As with many other features of Ada, we found the rules governing the use of packages too permissive; we have imposed restrictions which simplify the **use** clause and reduce the contexts in which it can be placed[6]. For purposes of data abstraction it is not necessary to employ package variables, and we seriously considered the possibility of disallowing these, to avoid possible side effects. However, the package feature would then lose another of its important applications, in controlling access to variables; the solution was to render all package variables visible *to SPADE* (i.e. to give them the appearance of global variables) by means of annotations.

To ensure that we had retained all the properties of packages which were essential for our purposes, a number of VDM and Z specifications (such as the Z specification case studies (Hayes, 1987)) were implemented in SPARK. The well-known problems of refinement of specifications were encountered, but we believe the package features retained, *together with our annotations for strengthening package specifications*, form a useful basis for data abstraction.

---

[6] See Additional Note 2

4.4 Security

We say that a programming language is *insecure* if a program can violate the definition of the language in any way which it is impossible, or even very difficult, to detect *prior to the program's execution.*

It is important to note the distinction between our view of language security and the conventional one. It has always been considered important to detect and report all language violations. However, until recently language violations (as well as the methods of formal language definition) have been viewed entirely in terms of the practical capabilities of compilers: errors have been classified as "compilation errors" (covering syntactic and "static semantic" errors) and "run-time errors" (including for instance range violations of values of dynamically-evaluated expressions). The over-riding concern has been to ensure that language violations are eventually captured; whilst it has been considered very desirable to detect them at compilation time - and indeed this need is reflected in many features of Ada - the detection of errors at run-time has been regarded as a tolerable alternative.

In a safety-critical real-time system, a "run-time error" can be quite as hazardous as any other kind of malfunction: *all* language violations must be detected prior to program execution. The distinction between "compile-time" and "run-time" errors is losing interest - save to the extent that this classification indicates the *difficulty* of detecting different kinds of language errors prior to program execution. (Loosely speaking, "compilation errors" can be detected in the course of compilation by fast (i.e. polynomial-time) deterministic algorithms, whereas establishing the absence of "run-time errors" such as range errors prior to program execution usually requires formal proof (German, 1978), which can be much more difficult). The shift in emphasis from statically and dynamically decidable properties to a more general notion of well-formation of programs, possibly giving rise to proof obligations in the course of program construction, appears explicitly in recent languages such as NewSpeak (Currie, 1984) and Verdi (Craigen, 1987; Saaltink, 1987).

The need to detect all language violations prior to run-time was taken into account in the design of SPARK. The insecurities in Ada are of several different kinds, which had to be overcome in different ways. Here we only indicate the nature of the problems and their solutions; full details will be given in Part 2.

An important example of an insecurity in Ada is the fact that it is possible to employ an illegal order of compilation, without an indication of this language violation being given. Elimination of this problem from SPARK was a direct consequence of our simplification of compilation unit inter-dependencies, already mentioned in Section 4.1 above.

Aliasing through parameter passing is undesirable in any programming language, but in Ada it may cause a program to give different results with different compilers, depending on whether they pass **in out** parameters by reference or by copying in and copying back. The execution of a program is said to be "erroneous" in the Language Reference Manual if its effect depends on the choice of parameter-passing mechanism, but this language violation cannot usually be detected. As in SPADE-Pascal, the mandatory annotations in subprogram declarations together with the rules governing the choice of actual parameters (see Section 6 of Part 2) will allow the SPARK Examiner to provide complete protection against aliasing through parameter passing, which eliminates the insecurity mentioned above.

Finally we outline the way in which language violations associated with dynamic semantics (i.e. "run-time errors") are to be trapped, prior to program execution, in employing SPARK. With the

language restrictions imposed above (banning for instance the use of access types) the problem here reduces to that of *range-checking.*

It is obviously essential to prove that dynamically computed values of expressions to be assigned to variables or subprogram parameters, or to be employed as array indices, meet their type constraints. (And of course even where an assignment is to a variable or parameter of type INTEGER, we should check that the assigned value will always lie in the range INTEGER'RANGE.) Wherever a range check is required, the verification-condition generator of the SPARK Examiner will generate theorems whose proofs (obtained with the help of the SPADE Proof-Checker) establish that the range constraints are met.

To make check-statements and the associated theorems immediately comprehensible, SPARK - like SPADE-Pascal - does not allow the use of **anonymous (sub)types** or **(sub)type aliasing**[7] (so that all relevant ranges have simple names, chosen by the program author) and we disallow redefinition of constant and (sub)type identifiers within their scope. Further restrictions to simplify the generation and proof of the theorems associated with range-checking -- which is very similar to the process of program verification in general – will be outlined in the next section.

4.5 Verifiability

For verification purposes, each (package and subprogram) unit of a program is provided with a specification, which defines the effect of executing its code on its environment. The problem of verifying a program is thereby reduced to that of separately verifying each of its units (i.e. proving that the code of each unit is in consonance with its specification): to verify a unit which employs other units, we only need models of the units which it employs directly - which can be based on their specifications rather than their code. In this way the verification task remains tractable even for large programs. Here we consider first those language issues which are relevant to the *separability* of program units for verification purposes; we then consider language features which affect the simplicity of verification of the units themselves.

Ada supports the notion of separate verification of program units, for instance through the concept of packages, with their specifications. However, putting the principle into practice required substantial simplifications to the language. The simplification of scope and visibility rules mentioned above was considered to be essential for the practical study of the interaction between any subprogram and its environment. (Global-definitions also help by reducing the set of variables to be considered, in analysing any procedure, to those which it employs (directly or indirectly) rather than all those whose scopes contain the procedure.) The simplifications of program unit inter-dependencies and annotations of these units were also found essential, for without them the specifications of program units would be inadequate for verification purposes. *Side-effects,* which would invalidate the separate analysis of program units, are not allowed in SPARK: function subprograms cannot have side-effects (all non-local variables which they employ[8] must be passed as parameters), and any other "invisible" modifications of variables of global significance (such as assignments to package variables) must be rendered visible through annotations.

---

[7] See Additional Note 3

[8] See Additional Note 4

With regard to verification of individual units, apart from the restrictions mentioned above a few minor restrictions on *control structure* have been found desirable, to facilitate analysis of all kinds; they impede programmers very little, since Ada is rich in this respect. In SPARK, (1) **goto** statements are illegal, (2) **exit** statements always apply to their innermost enclosing loops, (3) if an **exit** statement contains a **when** clause then its closest-containing compound statement must be a **loop** statement, (4) if an **exit** statement does not contain a **when** clause then its closest-containing compound statement must be an **if** statement without an else or elsif clause, whose closest-containing compound statement is a **loop** statement, (5) a function subprogram contains exactly one **return** statement, which must be the last statement in its body, and (6) procedure subprograms do not contain **return** statements.

With these restrictions it becomes easy to detect data-flow and information-flow errors, side-effects and errors such as aliasing errors, as well as other syntactic and "static semantic" errors. It is also guaranteed that code is "reducible", i.e. that every loop has a single entry point, simplifying the generation of verification conditions from loop invariants.

Since data-flow errors are easily detected, there is no merit here in initialising all variables explicitly in their declarations, as is sometimes recommended (Currie, 1984). Indeed, this can involve the assignment of meaningless initial values, making it difficult to detect failure to perform proper initializations by data flow analysis and obscuring program proof. Because of this - and to reduce the number of methods of assignment - **explicit initializations in declarations are prohibited** in SPARK[9].

To simplify formal proof, and render the SPADE Proof Checker as efficient as possible, we associate a notion of *scope* with *proof rules*. These always appear as annotations within packages; to SPADE, a proof rule is visible only inside the package which contains it and - if the rule is in a visible part - in those places where the package is used. (The proof of correctness of SPARK programs will be the subject of a separate report.)

4.6 Bounded Space and Time Requirements

In real-time control applications it is essential that the memory requirements of a program should not exceed that available. This is one of the reasons why we have removed all language features which require dynamic storage allocation. All constraints are statically determinable. It may be necessary to bound the depth of procedure calls and to calculate the space required for these, but this problem should be tractable.

We have not made any provision to bound execution time, for instance by limiting the number of iterations around program loops, as has been proposed for NewSpeak. We believe that to ensure that execution times are satisfactory, bounds on numbers of loop iterations should be obtained by proof methods, similar in nature to proofs of termination.

5 OUR ASSESSMENT OF SPARK

In our opinion the language proposed here would be satisfactory for developing high-integrity software. The Ada subset without annotations would still not be sufficiently secure (because of

---

[9] See Additional Note 5

possible aliasing problems for instance), but with the SPARK annotations we believe that all language violations other than range errors could be detected statically, in polynomial time. Range analysis would inevitably involve proof obligations, but these could easily be generated.

We have not yet constructed a formal definition of the language[10], but our restrictions have removed the major difficulties in producing this - in fact the formalisation of SPARK would be on relatively well-trodden ground, apart from the definition of annotations and their role.

The simplicity of the language implies that tools for static analysis of SPARK could be relatively simple and robust. Simplicity of the SPARK scope and visibility rules, the absence of side effects, and the "separability" of subprograms would also make the generation of range validity conditions and verification conditions relatively straightforward.

Some readers may be dismayed to see so many features of Ada removed, and feel that SPARK is "too small". It is by no means the largest subset of Ada which would be amenable to analysis by the techniques employed in SPADE, but it is significantly larger than SPADE-Pascal, which has been found adequate for a substantial number of safety-critical projects. The additional features which appear in SPARK (such as packages and private types) make programming *simpler* and *safer*, rather than complicate the verification task. Of course, the extent to which Ada must be simplified for high-integrity programming will be a matter of opinion: our preoccupation with simplicity is based on experience of what can be achieved with a high degree of confidence in practice, rather than what can be proved in principle.

Pedagogical considerations also suggest to us that drastic simplifications must be made. Safety-critical work demands complete mastery of a programming language, so that the programmer can concentrate on what he or she wants to say rather than struggle with the means of expression. In this regard, SPARK is presented here as a complicated set of restrictions of a very large language, to allow direct comparison with full Ada; however, a much lighter description of SPARK could be produced, which would make the language as easy to learn as Modula-2. Initial training based on a SPARK manual, bringing out the essential ideas of high-integrity programming in Ada, might be very worthwhile.

Finally, we must stress that the use of an annotated subset such as SPARK will not solve all our problems. Formal specification and proof of programs may become a little easier but it remains very difficult. And we do not have any formally-verified Ada compilers. We believe that the development of a high-integrity SPARK compiler is feasible; in the meantime, the use of SPARK would mostly employ the relatively well exercised paths through an Ada compiler, and authors of validation tests could concentrate on the more important features of the language. On those rare occasions where we have used the term "safe subset", for "safe" read "less dangerous".

6 ADDITIONAL NOTES ON RATIONALE

Subsequent development of SPARK has rendered inaccurate certain parts of the preceding Rationale, and these are indicated in this Edition by footnotes referring to the following list.

---

[10] See Additional Note 6

1 The SPARK Examiner can now generate verification conditions that can be used to prove the absence of run time errors and show that the program is exception free.

2 Rather than simplifying the **use** clause and reducing the contexts in which it can be placed, SPARK does not permit the basic **use** clause at all. SPARK 95 permits the **use type** clause in certain contexts.

3 Subtype aliasing is permitted for scalar types and non-tagged record types.

4 Function subprograms are allowed to read non-local variables (but not to update them).

5 Initializations in declarations are now optional in SPARK, but the initial values must be constant.

6 A formal definition of (most of) the SPARK language has now been constructed (Program Validation Ltd, 1994).

7 SPARK 95 now includes a subset of tagged types and type extension. Although these are a form of derived type, additional SPARK rules limit the complexity that results.

8 The advent of the Ravenscar Profile now provides a method of constructing concurrent Ada programs with deterministic behaviour. SPARK now optionally includes support for the Ravenscar Profile which is fully described in the manual SPARK - The SPADE Ada 95 Ravenscar Kernel (including RavenSPARK).

7 DEVELOPMENT OF SPARK 95

The original SPARK language was based on Ada 83. The standardisation of Ada 95 provided an opportunity to review the SPARK language to ensure that it remained a true Ada subset and to seek to exploit beneficial new features of Ada.

The SPARK 95 language therefore includes not only the necessary changes to SPARK83 to maintain compatibility with Ada, but also those major new language features of Ada 95 that are consistent with the objectives and philosophy of SPARK. We have also made some extensions to the language of SPARK annotations, in order to provide increased flexibility in the use of information flow analysis.

SPARK 95 includes the following major language extensions introduced by Ada 95:

- Child packages

  The introduction of child packages in Ada 95 greatly assists the development of large programs in a modular and hierarchic manner. Private child packages in particular are a significant addition to SPARK since they provide a natural way of achieving the encapsulation and top-down refinement of program state.

- Use Type Clauses

  SPARK has a stricter visibility model than Ada and prohibits the **use** clause. Ada 95 has introduced the **use type** clause which makes operators of a specified type directly visible. This has significant benefits in SPARK, eliminating the need for (often tedious) renaming of predefined operators for types from another unit.

- Modular Types

SPARK 95 introduces modular types from Ada95, but with a number of restrictions, most notably that a type's modulus must be a power of 2. Modular types are particularly useful in low-level interfacing code, checksumming and cryptographic algorithms.

- Tagged Types

  Tagged record type and extensions of those types are included in SPARK 95 with certain restrictions principally the exclusion of class-wide operations and dynamic dispatch.

Changes to Ada 95 rules for parameters, principally the readability of parameters of mode `out` have also allowed extended global annotations and optional information flow analysis.

In SPARK 83 every procedure subprogram must have a **derives** annotation which firstly identifies its imports and exports (this information being needed for language conformance checking and data flow analysis) and secondly states the dependency relations between those imports and exports (this information being needed for information flow analysis).

In SPARK 95, the form of the **global** annotation for a procedure has been extended so that optionally a mode (**in**, **out** or **in out**) may be specified for each global. Along with the modes of formal parameters, this information then fully identifies imports and exports.

The result is a clearer separation in SPARK between the annotations required for language security and those required for deeper analysis, giving the user more flexibility. It means that the **derives** annotation and information flow analysis are no longer mandatory. Selected parts of a SPARK program can then be analysed in such a way that full language conformance checking and data flow analysis are performed but full information flow analysis is not carried out.

8 DEVELOPMENT OF RavenSPARK

The development of RavenSPARK is described in the manual SPARK - The SPADE Ada 95 Ravenscar Kernel (including RavenSPARK).

II Specification of SPARK

This section catalogues the differences between SPARK 95 and Ada 95. Anything not mentioned here is the same in SPARK 95 as it is in Ada 95.

SPARK 95 is a sub-language of Ada 95, supplemented with annotations (formal comments). Since annotations always begin on each line with the Ada comment symbol "--", all SPARK programs comply with the Ada standard.

The section numbers used here correspond to those of the International Standard "Ada 95 Reference Manual", ANSI/ISO/IEC 8652, January 1995. We shall refer to this language reference manual as the Ada *LRM*. The context-free syntax of the SPARK language is described in the same form as in the Ada *LRM*; syntax rules marked with an asterisk (*) are variants of rules of standard Ada and those marked with a plus (+) are additional rules. Section III of this Report contains a collected syntax of SPARK.

2 LEXICAL ELEMENTS

2.4 Numeric Literals

2.4.2 Based Literals

Based real literals shall not be employed.

```
*  based_literal ::=
       base # based_numeral # [exponent]
   base ::= numeral
   based_numeral ::=
       extended_digit { [underline] extended_digit }
   extended_digit ::= digit | A | B | C | D | E | F
```

2.7 Comments

In SPARK, if the first two adjacent hyphens in a line are immediately followed by a sharp symbol (**#**), then these symbols are considered to be the start of an *annotation* (see Section 2.11), which influences the legality of a SPARK program.

2.8 Pragmas

Pragmas are only allowed at the following places in a SPARK program:

- at any place where a declaration or a statement would be allowed;
- where a body would be allowed in a declarative part;
- between a context clause and its following library unit or secondary unit;
- at any place where a compilation unit would be allowed.

The presence or absence of a pragma, other than pragma Elaborate_Body and pragma Import, has no effect on the legality of a SPARK text. The pragma Elaborate_Body is discussed in Section 10.2.1 and the pragma Import in Annex B.1.

2.9 Reserved Words

In addition to the reserved words of Ada, the identifiers listed below are reserved words in SPARK.

assert	from	inherit	own
		initializes	
check	global	invariant	post
			pre
	hide		
derives	hold	main_program	some

Further identifiers are reserved for use by certain SPARK language tools (see Annex M) and should not be used if generation of verification conditions is required.

2.10 Allowable Replacements of Characters

SPARK employs standard characters; replacement characters shall not be used.

2.11 Annotations

In SPARK, if the first two adjacent hyphens in a line are immediately followed by a sharp symbol (**#**[11]), then these symbols and all subsequent symbols in the line, up to the next pair of adjacent hyphens (if such a pair is present), form part of an *annotation.* An annotation can extend over any number of lines, but every non-blank continuation line must begin (after any leading spaces) with the three characters **--#.**

Like other kinds of Ada comments, SPARK annotations will be ignored by an Ada compiler. However, a SPARK annotation is a formal statement, which conveys information to the SPARK Examiner.

Some examples of annotations are given below.

```
--# global in  A, B, C;
--# derives A from B, C;
--# global out  CurrentSymbol, Input;       -- Comment embedded
--# derives CurrentSymbol from Input;       -- in an annotation.
```

The inclusion of the following kinds of annotations is imposed by certain language rules of SPARK:

global definitions	(see Section 6.1.2)
dependency relations	(see Section 6.1.2)
inherit clauses	(see Section 7.1.1)
own variable clauses	(see Section 7.1.3)
initialization specifications	(see Section 7.1.4)
refinement definitions	(see Section 7.2.1)
main program annotations	(see Section 10.1.1)

The presence or absence of such annotations influences the legality of a SPARK program.

---

[11] To maximise compatibility with other software tools, the SPARK Examiner allows the annotation introduction character to be defined by the user

3 DECLARATIONS AND TYPES

3.1 Declarations

SPARK does not have declarations associated with discriminants, tasks, generics or exceptions, which are not allowed in the language.

Subprogram declarations and package declarations are not considered to be basic declarations in SPARK, but subprogram declarations are nevertheless still permitted in the visible and private parts of packages (see Section 7.1), and package declarations are still permitted in declarative parts (see Section 3.11). Subprogram declarations are also permitted in declarative parts when immediately followed by pragma Import (see Sections 3.11 and B.1). SPARK does not have abstract subprogram declarations.

Renaming declarations too are not considered to be basic declarations in SPARK. They are still permitted in the visible parts of packages (see Section 7.1), and in declarative parts (see Section 3.11), but are subject to restrictions in application (see Section 8.5).

* basic_declaration ::=
type_declaration | subtype_declaration
| object_declaration | number_declaration
defining_identifier ::= identifier

SPARK does not permit an enumeration literal to be declared with a character literal as its name (see Section 3.5.1), nor a function to be declared with an operator symbol as its name (see Section 6.1).

3.2 Types and Subtypes

The following classes of types are all excluded from SPARK: decimal fixed point , access, task and protected.

3.2.1 Type Declarations

Type declarations and definitions are restricted to those types which are supported by SPARK. Hence there are no incomplete type declarations, discriminant parts, task type declarations, protected type declarations, access type definitions or derived type definitions other than type extensions of tagged record types.

* type_declaration ::=
full_type_declaration |
private_type_declaration |
private_extension_declaration
* full_type_declaration ::= **type** defining_identifier **is** type_definition ;

* type_definition ::=
enumeration_type_definition | integer_type_definition
| real_type_definition | array_type_definition

```
    |  record_type_definition          | modular_type_definition
    |  record_type_extension
+   record_type_extension ::= new type_mark with record_definition ;
```

3.2.2 Subtype Declarations

SPARK does not have discriminant constraints, digits constraints or delta constraints.

```
    subtype_declaration ::=
          subtype defining_identifier is subtype_indication ;
    subtype_indication ::= subtype_mark [ constraint ]
    subtype_mark ::= subtype_name
    constraint ::= scalar_constraint | composite_constraint
*   scalar_constraint ::= range_constraint
*   composite_constraint ::= index_constraint
```

All constraints shall be statically determinable in SPARK.

If the subtype indication has no constraint, then the given type_mark must denote a scalar type or a record type. Thus an Ada subtype declaration of the form

```
subtype T1 is T2;
```

is only allowed in SPARK if `T2` denotes a scalar subtype or a record subtype. In the former case, it may be regarded as equivalent to

```
subtype T1 is T2 range T2'First .. T2'Last;
```

A subtype indication for a Boolean subtype must not include a constraint, since only full-range Boolean subtypes are permitted in SPARK.

3.3 Objects and Named Numbers

In SPARK, the result of evaluating a function call or an aggregate is not considered to be an object. Together with the absence of certain language constructs in SPARK, this means that an object is restricted to being one of the following:

- the entity declared by an object declaration;
- a component of another object.

Similarly, the following (and no others) represent constants in SPARK:

- an object declared by an object declaration with the reserved word **constant**;
- a formal parameter of mode **in**;
- a loop parameter;
- a selected component or indexed component of a constant.

The only indefinite subtypes in SPARK are unconstrained array subtypes.

3.3.1 Object Declarations

SPARK and Ada object declarations differ in the following respects. In SPARK,

1 In both constant and variable declarations, the nominal subtype shall be given by a subtype mark and shall not be unconstrained. (The only exception to this is the admission of declarations of constants of type string).

2 There are no aliased objects.

3 There are no single task declarations or single protected declarations.

4 The expression initializing an object shall not contain any of the following constructs:

- a name denoting an object which has not been declared by a constant or named number declaration;
- a function call which is not a call of a predefined operator or attribute;
- an indexed component;
- a selected component whose prefix denotes a record object.

* object_declaration ::= defining_identifier_list : [**constant**] subtype_mark [**:=** expression] **;**
defining_identifier_list ::= *defining_*identifier { , *defining_*identifier }

Rule (1) above prevents the declaration of objects of an anonymous nominal subtype. For instance, in Ada the following declaration would be valid:

```
Index : Integer range 1 .. 10;
```

Here the object `Index` has been declared with an anonymous subtype of Integer. In SPARK the declaration of `Index` would be of the form:

```
subtype Index_Range is Integer range 1 .. 10;
Index : Index_Range;
```

Furthermore, since SPARK requires the nominal subtype to be constrained, the actual subtype of an object declared by an object declaration is always the same as its nominal subtype and the object is only 'constrained by its initial value' in the case of constant string declarations.

Rule (4) above means that an initial value assigned by an object declaration is always statically determinable in SPARK.

3.4 Derived Types and Classes

SPARK does not have derived type definitions other than record type extensions. Nevertheless, numeric types are still considered to be implicitly derived from a corresponding root numeric type (see 3.5.4 and 3.5.6).

3.4.1 Derivation Classes

SPARK has no class-wide types, but does have the implicitly defined universal types for the integer, real and fixed point classes.

3.5 Scalar Types

In SPARK, the range in a range constraint shall be static. Furthermore, no static range shall be a null range, i.e. the upper bound of a static range shall be greater than or equal to the lower bound of the range.

* range_constraint ::= **range** *static*_range
range ::= range_attribute_reference
| simple_expression **..** simple_expression

SPARK does not have the following attributes: 'Image, 'Wide_Image, 'Wide_Value, 'Wide_Width, 'Width, 'Value.

The attribute reference S'Base (for a scalar subtype S) is allowed only as the prefix of the name of another attribute reference: for example, S'Base'First.

In SPARK, the attributes 'Succ and 'Pred are not defined for Real types nor the type Boolean.

3.5.1 Enumeration Types

In SPARK, enumeration literals are not regarded as parameterless functions, but simply as names denoting the distinct values of the associated enumeration type.

Enumeration literals shall not be overloaded, i.e. the same enumeration literal shall not occur in two enumeration type definitions which are both directly visible at any point.

Since the character literals belong to the enumeration type Character in package Standard, character literals cannot be used as enumeration literals in a user-defined enumeration type definition.

enumeration_type_definition ::=
(enumeration_literal_specification { **,** enumeration_literal_specification } **)**
* enumeration_literal_specification::= *defining*_identifier

3.5.2 Character Types

User-defined character types are not permitted in SPARK.

The type Wide_Character is not predefined in SPARK.

3.5.3 Boolean Types

In SPARK, the type Boolean is still considered to be predefined as

```
type Boolean is ( False, True );
```

However the ordering operators (see Section 4.5.2) are not defined for Boolean types.

3.5.4 Integer and Modular Types

* integer_type_definition ::= signed_ integer_type_definition

signed_integer_type_definition ::=
range *static*_simple_expression **..** *static*_simple_expression
modular_type_definition ::= **mod** *static*_simple_expression

Although SPARK does not have derived type definitions, an integer type is still considered to be implicitly derived from *root_integer* in SPARK. Therefore, all the predefined integer operators and basic operations are applicable to the new type.

Modular types are allowed with the following restrictions:

- The Modulus of a type must be a positive power of 2.
- Unary arithmetic operators (unary -, +, **abs**) are not permitted. The unary "**not**" operator is allowed, as are all binary arithmetic and logical operators.

3.5.5 Operations of Discrete Types

The operations of the enumeration type Boolean include the predefined equality and inequality operators, but not the ordering operators nor the attributes 'Pos and 'Val.

3.5.6 Real Types

real_type_definition ::= floating_point_definition | fixed_point_definition

Although SPARK does not have derived type definitions, real types (both floating-point and fixed-point) are still considered to be implicitly derived from *root_real*. Therefore, the predefined operators and basic operations of such types are still available in SPARK. In addition, many of the attributes that Ada associates with floating-point and fixed-point types are incorporated in SPARK (see Appendix A).

It is important to note that the real types provide only approximations to the real numbers, and that both floating-point and fixed-point arithmetic are implementation-dependent.

3.5.7 Floating Point Types

In SPARK, the expression specifying the requested decimal precision shall be a *simple* expression.

* floating_point_definition ::=
digits *static*_simple_expression [real_range_specification]
real_range_specification ::=
range *static*_simple_expression .. *static*_simple_expression

3.5.9 Fixed Point Types

SPARK does not have decimal fixed point types and hence has no decimal fixed point definitions.

In SPARK, the expression specifying the delta of a fixed point type shall be a *simple* expression.

* fixed_point_definition ::= ordinary_fixed_point_definition
* ordinary_fixed_point_definition ::=
delta *static*_simple_expression real_range_specification

3.6 Array Types

SPARK and Ada array type definitions differ in the following respects. In SPARK,

1 A discrete subtype definition shall be a subtype mark only, and not specified in terms of an anonymous subtype.

2 A component definition shall be a subtype mark only, and not specified in terms of an anonymous subtype.

3 A component definition shall not contain the reserved word **aliased**

array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition
unconstrained_array_definition ::=
    array (index_subtype_definition { **,** index_subtype_definition }) **of**
        component_definition
index_subtype_definition ::= subtype_mark **range <>**
constrained_array_definition ::=
    array (discrete_subtype_definition { **,** discrete_subtype_definition } **) of**
        component_definition
* discrete_subtype_definition ::= *discrete*_subtype_mark
* component_definition ::= subtype_mark

Rules (1) and (2) above, together with the restrictions on object declarations, prevent the occurrence of anonymous subtypes in declarations of array objects. For instance, in Ada a constrained array variable might be declared as follows:

```
Upper_Case_Table : array (1 .. 10) of Character range 'A' .. 'Z';
```

whereas in SPARK this variable declaration would take the following form:

```
subtype Index_Range is Integer range 1 .. 10;
subtype Capital_Letter is Character range 'A' .. 'Z';
type Upper_Case_Array is array (Index_Range) of Capital_Letter;
Upper_Case_Table : Upper_Case_Array;
```

These rules also prevent the use of anonymous subtype array constants. Thus in SPARK an array (variable or constant) must belong to a named subtype.

3.6.1 Index Constraints and Discrete Ranges

SPARK and Ada index constraints and discrete ranges differ in the following respects. In SPARK,

1 An index constraint shall be specified by subtype marks only, and not in terms of anonymous subtypes.

2 In the SPARK grammar, an index constraint is not defined in terms of discrete ranges, but the latter are still used elsewhere. It will be noted, from the definition below, that all discrete ranges are static in SPARK.

* index_constraint ::= **(***discrete*_subtype_mark { **,** *discrete*_subtype_mark } **)**
* discrete_range ::= *discrete*_subtype_indication | *static*_range

Thus, whereas an acceptable Ada declaration for an array variable might be of the form

```
Ten_Characters : String(1 .. 10);
```

in SPARK the same object would be declared as follows:

```
subtype Index_Range is Integer range 1 .. 10;
subtype String_10 is String(Index_Range);
Ten_Characters : String_10;
```

3.6.3 String Types

The type Wide_String is not predefined in SPARK.

String literals are only compatible with the unconstrained array type String.

The concatenation operator, "&", is defined for type String (though not for other one-dimensional array types), but its use is severely restricted (see Section 4.5.3).

The ordering operators <, <=, >= and > are defined for type String but not for any other one-dimensional arrays.

In SPARK, all subtypes of String shall have a lower index bound equal to 1.

3.7 Discriminants

Discriminants are not supported in SPARK.

3.8 Record Types

SPARK and Ada record type definitions differ in the following respects. In SPARK,

1 a record type definition shall not contain the reserved words **abstract** or **limited**;

2 a record definition cannot be **null record** unless it is tagged[12];

3 a component list cannot have a variant part;

4 a component list cannot be the reserved word **null** unless the record is tagged;

5 a component item cannot be a representation clause;

6 a component declaration cannot have a default expression;

```
*   record_type_definition ::= [tagged] record_definition
*   record_definition ::=
        record
            component_list
        end record | null record
*   component_list ::= component_item { component_item } | null
*   component_item ::= component_declaration
*   component_declaration ::=
        defining_identifier_list : component_definition ;
```

[12] Note that a tagged null record serves only as a basis for type extension; direct use of the null record is not possible.

Rules (2) and (4) above mean that there are no untagged *null records* in SPARK.

Since a component definition must be a subtype mark (see 3.6), a record component cannot have an anonymous subtype.

3.8.1 Variant Parts and Discrete Choices

Variant parts are not supported, but discrete choices are used in aggregates and case statement alternatives.

In SPARK, the syntactic category *discrete_choice* differs from Ada in the following respects:

1 an expression as a discrete choice shall be a simple expression and shall be static;

2 the category does not include the choice **others**, which is instead directly incorporated into the syntax for array aggregates (Section 4.3.3) and case statements (Section 5.4).

```
  discrete_choice_list ::= discrete_choice { | discrete_choice }
* discrete_choice::= static_simple_expression | discrete_range
```

3.9 Tagged Types and Type Extensions

Tagged types and type extensions are subject to the following additional rules:

1 Abstract types may not be used.

2 Controlled types may not be used.

3 Tagged types and type extensions may only be declared in the specification of library unit packages.

4 At most one *tagged type* or *type extension* may be declared in any package.

5 A subprogram declaration may not have the same name as a potentially inheritable subprogram unless it successfully overrides it.

6 Actual parameters matching formals of tagged types must be objects (or ancestor type conversions of objects) not general expressions.

7 The operand of an ancestor type conversion must be an object (not an expression).

8 When completing a private extension the type named in the private part must be exactly the same as that named in the visible part (Ada requires only that it has to be derived from the same root). This is simply a matter of simplicity and clarity.

9 The ancestor part of an extension aggregate may not be a type mark.

10 The primitive operations of a tagged type or type extension do not include functions that return the tagged type: i.e. a function result may not be a *controlling operand*.

Note also that SPARK prohibits class-wide operations including dynamic dispatch (equivalent to pragma Restrictions (No_Dispatch) in RM H.4 (19).

3.10 Access Types

Access types are not allowed.

3.11 Declarative Parts

Declarative parts in SPARK and Ada differ in the following respects. In SPARK,

1 subprogram declarations are not allowed in a declarative part (except when immediately followed by pragma Import - see Annex B.1), but subprogram bodies are permitted;

2 there are restrictions on the position of renaming declarations (which are not basic declarations in SPARK);

3 use clauses are not basic declarative items in SPARK, but the **use type** clause is permitted in an *embedded package declaration* (see following syntax).

```
*   declarative_part ::=
        { renaming_declaration }
        { declarative_item          |  embedded_package_declaration
                                    |  external_subprogram_declaration }
    declarative_item ::= basic_declarative_item  |  body
*   basic_declarative_item ::=
        basic_declaration  |  representation_clause
+   embedded_package_declaration ::=
        package_declaration
        { renaming_declaration | use_type_clause }
+   external_subprogram_declaration ::=
        subprogram_declaration
        pragma Import ( pragma_argument_association,  pragma_argument_association
                                    { , pragma_argument_association } ) ;
    body ::= proper_body  |  body_stub
*   proper_body ::= subprogram_body  |  package_body
```

4 NAMES AND EXPRESSIONS

4.1 Names

SPARK and Ada names differ in the following respects. In SPARK,

1 character literals, operator symbols and type conversions are not names;

2 slices are not allowed;

3 there are no (explicit or implicit) dereferences.

```
*  name ::= direct_name
        | indexed_component
        | selected_component
        | attribute_reference
        | function_call
*  direct_name ::= identifier
*  prefix ::= name
```

SPARK excludes most whole-array operations on unconstrained array objects, in order that rules relating to index bounds may be statically checked. Consequently, the name of an unconstrained array object (formal parameter) shall only appear in the following contexts:

1 as the prefix of an attribute reference;

2 as the prefix of an indexed component;

3 as an actual parameter in a call to a subprogram where the corresponding formal parameter is also unconstrained;

4 as an operand, of type String, of one of the relational operators =, /=, <, <=, > or >= ;

5 in a procedure or function annotation (see Section 6.1.1).

4.1.2 Slices

Slices are not allowed in SPARK.

4.1.3 Selected Components

In SPARK a selector name cannot be a character_literal or an operator_symbol.

```
   selected_component ::= prefix . selector_name
*  selector_name ::= identifier
```

Since a selector name cannot be an operator symbol, operator subprograms can only be called using an infix notation.

The prefix of an expanded name shall not denote a loop statement; nor, except within the parent unit name of a subunit **separate** clause, shall it denote a subprogram. (Hence, apart from this exception, the prefix of an expanded name in SPARK must denote a package.)

4.1.4 Attributes

The SPARK syntax for attribute designators excludes the alternative Access since the corresponding attribute is not supported.

```
    attribute_reference ::= prefix ' attribute_designator
*   attribute_designator::= identifier [ ( expression [ , expression ] ) ] | Delta | Digits
    range_attribute_reference ::= prefix ' range_attribute_designator
    range_attribute_designator::= Range [ ( static_expression ) ]
```

The attributes supported by SPARK are listed in Annex K.

4.2 Literals

In SPARK, character literals are not regarded as parameterless functions, but simply as constructs denoting values of the predefined type Character.

The rules governing the use of string literals in SPARK were given in Section 3.6.3.

The literal **null** does not exist in SPARK.

4.3 Aggregates

An aggregate is not a primary in SPARK, which implies that whenever an aggregate is used in an expression it must be qualified by an appropriate type mark to form a qualified expression. An unqualified aggregate is permitted as an "aggregate item", but only inside an aggregate for a multi-dimensional array type (see Section 4.3.3).

As in Ada, the type denoted by the qualifying type mark determines the required type for each of the aggregate components. In SPARK, it also statically determines any subtype constraints; hence, for each component value that is an array, the upper and lower bounds (for each index position) shall be equal to those imposed by the corresponding component subtype.

The evaluation of an aggregate is not considered to create an object in SPARK, but simply a value of the appropriate type.

4.3.1 Record Aggregates

In a record aggregate each named component association can only have a single aggregate choice, and **others** cannot be used. As reflected in the SPARK syntax, positional and named component associations shall not be mixed within the same record aggregate (Ada already forbids this mixing for array aggregates). SPARK does not permit the reserved words **null record** in an aggregate unless it is an extension aggregate.

```
*   record_aggregate ::= positional_record_aggregate | named_record_aggregate
+   positional_record_aggregate ::= ( expression { , expression } )
+   named_record_aggregate ::=
        ( record_component_association  { , record_component_association } )
```

* record_component_association ::= *component*_selector_name **=>** expression

4.3.2 Extension aggregates

The ancestor part of an extension aggregate may not be a type mark.

As for record aggregates above, the grammar prevents mixing of named and positional associations.

* extension_aggregate ::= (ancestor_part **with** record_component_association_list) |
(ancestor_part **with null record**)
* ancestor_part ::= expression
\+ record_component_association_list ::= named_record_component_association
| positional_record_component_association
\+ positional_record_component_association ::= expression { , expression }
\+ named_record_component_association ::=
record_component_association { , record_component_association }

4.3.3 Array Aggregates

In the SPARK grammar the syntactic category *discrete_choice* does not include the choice **others**, which is instead directly incorporated in the array aggregate syntax.

array_aggregate ::= positional_array_aggregate | named_array_aggregate
* positional_array_aggregate ::=
(aggregate_item , aggregate_item { , aggregate_item })
| (aggregate_item { , aggregate_item } , **others =>** aggregate_item)
* named_array_aggregate ::=
(array_component_association { , array_component_association }
[, **others =>** aggregate_item])
| (**others =>** aggregate_item)
* array_component_association ::= discrete_choice_list **=>** aggregate_item
\+ aggregate_item ::= expression | array_aggregate

In SPARK, all choices of named associations in an array aggregate shall be static. This follows from the static nature of discrete ranges and discrete choices in SPARK (see Sections 3.6.1 and 3.8.1).

Since an aggregate must be qualified by an appropriate type mark, the bounds of an array aggregate are always known from the context in SPARK. The number of components in a positional array aggregate, or the range of choices in a named array aggregate, must be (statically) consistent with the bounds associated with the qualifying type mark.

4.4 Expressions

SPARK and Ada expressions differ in the following respects. In SPARK,

1 **null** is not a primary;

2 allocators and aggregates are not primaries;

3 character literals and type conversions are primaries.

expression ::=
relation { **and** relation } | relation { **and then** relation }
| relation { **or** relation } | relation { **or else** relation }

```
    |   relation { xor relation }
relation ::=
        simple_expression [ relational_operator simple_expression ]
    |   simple_expression [ not ] in range
    |   simple_expression [ not ] in subtype_mark
simple_expression ::=
    [ unary_adding_operator ] term { binary_adding_operator term }
term ::= factor { multiplying_operator factor }
factor ::= primary [** primary] | abs primary | not primary
*   primary ::=
        numeric_literal         | character_literal   | string_literal
    |   name                    | type_conversion
    |   qualified_expression    | (expression)
```

Rule (3) above is required because character literals and type conversions are not names in SPARK.

4.5 Operators and Expression Evaluation

4.5.1 Logical Operators and Short-circuit Control Forms

The logical operators **and**, **or** and **xor** for one-dimensional arrays of Boolean components are defined only when both operands have the same upper and lower bounds.

4.5.2 Relational Operators and Membership Tests

In SPARK, the ordering operators `<`, `<=`, `>`, `>=` are not defined for Boolean types or any array type except String.

The equality operators `=` and `/=` for array types other than String are defined only when, for each index position, the operands have equal bounds.

The equality operators `=` and `/=` are defined for floating point types, but their use is discouraged and elicits a warning from the SPARK Examiner.

4.5.3 Binary Adding Operators

The concatenation operator, "&", has a restricted use in SPARK. It is defined only for result type String and each operand must be either a string literal, a static character expression, or another concatenation.

4.5.5 Multiplying Operators

In SPARK, a multiplication or division with operands of fixed point types shall be qualified or explicitly converted to identify the result type.

4.6 Type Conversions

The only (explicit) view conversions in SPARK are those involving ancestor conversion of an extended type; furthermore, such an ancestor conversion must be a view conversions (implying that its operand is an object). All other type conversions are value conversions.

```
*   type_conversion ::= subtype_mark ( expression )
```

In SPARK a type conversion where the operand type and the target type are array types must satisfy the following additional conditions (which are statically determinable in SPARK):

1 The target subtype shall be a constrained array subtype.

2 For each index position, both the upper and the lower bounds of the operand array shall be equal to those of the target subtype.

The operand of a type conversion shall not be a character literal or a string literal, nor such an expression enclosed in parentheses.

In SPARK a type conversion other than a view conversion cannot be an actual parameter in a subprogram call whose corresponding formal parameter is of mode **in out** or **out**.

4.7 Qualified Expressions

In SPARK, the type mark of a qualified expression shall not denote an unconstrained array type.

If the type mark denotes a constrained array subtype, then for each index position the upper and lower bounds of the operand shall be equal to those associated with that subtype.

4.8 Allocators

Allocators are not allowed in SPARK.

4.9 Static Expressions and Static Subtypes

In SPARK, the definition of static expression is extended to include enumeration literals explicitly. (In Ada their static nature is defined indirectly in terms of their status as static functions but this does not apply in SPARK.)

A SPARK program shall not contain a static expression whose value violates a range constraint or an index constraint.

5 STATEMENTS

5.1 Simple and Compound Statements - Sequences of Statements

SPARK imposes the following restrictions on the use of simple and compound statements:

1 Statements cannot be labelled, however, loop statement identifiers may be used (see Section 5.5).

2 The following kinds of simple statements cannot be employed:

entry call statements

goto statements

requeue statements

delay statements

abort statements

raise statements

3 The following kinds of compound statements cannot be employed:

block statements

accept statements

select statements

4 Code statements can be employed in SPARK programs, under the same conditions as in Ada (see Section 13.8 of the Ada *LRM*). In the definition of SPARK this feature is provided by the code_insertion - see Section 6.3.

```
   sequence_of_statements ::= statement { statement }
*  statement ::=
       simple_statement  |  compound_statement
*  simple_statement ::= null_statement
       |  assignment_statement      |  procedure_call_statement
       |  exit_statement            |  return_statement

*  compound_statement ::=
           if_statement             |  case_statement
       |   loop_statement
   null_statement ::= null;
   statement_identifier ::= direct_name
```

5.2 Assignment Statement

In SPARK, an implicit subtype conversion ('sliding') never occurs in an array assignment: for each index position, the upper and lower bounds of the value of the right-hand side expression shall be equal to those associated with the subtype of the array variable. This follows from the rules on type conversion (Section 4.6).

(Note that the rules of Section 4.1 prevent assignment to an unconstrained array parameter.)

There are additional rules concerning the use of external variables, or functions which reference external variables, (see Section 7) in assignment statements:

1 External variables of mode **out** may not be referenced.

2 External variables of mode **in** may not be updated.

3 External variables, and functions which globally reference external variables, may not form part of assigned expressions; they may only appear directly in simple assignment statements.

Rule 3 is to prevent ordering effects in the reading of external devices.

5.4 Case Statements

In the SPARK grammar the syntactic category *discrete_choice* does not include the choice **others**, which is instead directly incorporated in the case statement syntax.

```
*  case_statement ::=
       case expression is
           case_statement_alternative
       {   case_statement_alternative }
       [   when others => sequence_of_statements ]
       end case;
   case_statement_alternative ::=
       when discrete_choice_list  => sequence_of_statements
```

Although type conversions are not names in SPARK, the rule in *LRM* 5.4(7) regarding the range of values to be covered by the discrete choices still applies. Hence if the expression in a case statement is a type conversion whose subtype mark denotes a static and constrained scalar subtype (as all subtype marks do in SPARK), then the range of values covered shall be exactly those belonging to that subtype.

5.5 Loop Statements

A loop statement may be named by a loop statement identifier, as in Ada. There are restrictions on the use of loop indentifiers in exit statements, see Section 5.7. A loop identifier may not be used as a prefix of a loop parameter (c.f. Section 8.3).

A loop parameter specification shall include an explicit subtype mark for the range over which the loop parameter will iterate. This prevents the loop parameter from having an anonymous subtype.

```
loop_statement ::=
    [ loop_statement_identifier : ]
        [ iteration_scheme ]
          loop
                sequence_of_statements
          end loop [ loop_statement_identifier ] ;
iteration_scheme ::= while condition
    |   for  loop_parameter_specification
*   loop_parameter_specification ::=
    defining_identifier in [ reverse ] discrete_subtype_mark [ range range ]
```

5.6 Block Statements

Block statements cannot be used in SPARK.

5.7 Exit Statements

In SPARK, the following restrictions are imposed on the use of exit statements:

1 An exit statement always applies to the most closely enclosing loop statement.

2 An exit statement may name a loop label which, if present, must match the label of the most closely enclosing loop statement.

3 If an exit statement contains a when clause then its closest-containing compound statement shall be a loop statement.

4 If an exit statement does not contain a when clause then its closest-containing compound statement shall be an if statement, which has no elsif or else clauses, and whose closest-containing compound statement is a loop statement; in this case the exit statement shall be the last statement within the if statement.

```
*   exit_statement ::= exit [ simple_name ] [ when condition ] ;
```

5.8 Goto Statements

SPARK does not allow the use of goto statements.

6 SUBPROGRAMS

6.1 Subprogram declarations

SPARK and Ada subprogram declarations differ in the following respects. In SPARK,

1 a declaration of a procedure or a function may contain a corresponding annotation (see Section 6.1.1);

2 a function designator must be an identifier (not an operator symbol);

3 a subprogram cannot have a parent unit name (only packages may be child units);

4 a formal parameter cannot have a default expression;

5 there are no abstract subprogram declarations;

6 there are no access parameters.

```
*  subprogram_declaration ::=
       procedure_specification ; procedure_annotation
     | function_specification ; function_annotation
+  procedure_specification ::=
     procedure defining_identifier parameter_profile
+  function_specification ::=
     function defining_designator parameter_and_result_profile
*  designator ::= identifier
*  defining_designator ::= defining_identifier
   defining_program_unit_name ::= [ parent_unit_name . ] defining_identifier
   operator_symbol ::= string_literal
   defining_operator_symbol ::= operator_symbol
   parameter_profile ::= [ formal_part ]
   parameter_and_result_profile ::= [ formal_part ] return subtype_mark
   formal_part ::=
     ( parameter_specification { ; parameter_specification } )
*  parameter_specification ::=
     defining_identifier_list : mode subtype_mark
   mode ::= [ in ] | in out | out
```

In SPARK, the subtype mark following **return** in the profile of a function shall not denote an unconstrained array subtype.

6.1.1 Procedure and Function Annotations

A procedure annotation may have up to two constituents, as follows.

```
+  procedure_annotation ::=
     [ global_definition ]
     [ dependency_relation ]
```

The purpose of global definitions and dependency relations is explained in the next section. It will be seen there that SPARK imposes certain rules of consistency between the global definition and dependency relation of a procedure, its body and its environment. Hence these constituents of procedure annotations affect the legality of SPARK texts.

A function subprogram annotation does not contain a dependency relation and its global definition (if any) may have a simpler form, omitting global modes, for reasons explained below.

```
+   function_annotation ::=
        [ global_definition ]
```

6.1.2 Global Definitions and Dependency Relations

To explain the role of subprogram annotations we shall employ the terminology of Sections 3.3 and 6.1 of the Ada *LRM*, which define the *reading* and *updating* of values of objects, and the *modes* of formal parameters of subprograms, as follows:

"...The value of an object is *read* when the value of any part of the object is evaluated or when the value of an enclosing object is evaluated. The value of a variable is *updated* when an assignment is performed to any part of the variable, or when an assignment is performed to an enclosing object."

"...The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter; **in**, **in out** or **out**."

We shall also employ the terms *local* and *global* in the same way as the Ada *LRM* (see its Section 8.1): " .... A declaration is *local* to a declarative region if the declaration occurs immediately within the declarative region. An entity is *local* to a declarative region if the entity is declared by a declaration that is local to the declarative region. A declaration is *global* to a declarative region if the declaration occurs immediately within another declarative region that encloses the declarative region. An entity is *global* to a declarative region if the entity is declared by a declaration that is global to the declarative region."

In Ada, appropriate use of parameter modes in a subprogram specification provides some protection, controlling to some extent the reading and updating of global variables by the subprogram. In SPARK, the transactions between a subprogram and its environment are specified and controlled much more precisely, by adding annotations to subprogram specifications and imposing some additional rules. To explain these, it is useful to consider briefly the design processes we employ, first for procedures and then for function subprograms.

The purpose of a procedure is to perform an *(updating) action*, involving the computation of values and assignments of them to variables which are external to the procedure. It can return a value to its calling environment by updating a global variable *directly;* alternatively, it can return a result *indirectly* by updating a formal parameter of mode **in out** or **out**. For any procedure P, we describe the global variables and formal parameters employed to convey its results to its calling environment as the *exported variables* of P. To derive its results the procedure P may itself need to read values

previously derived in its calling environment. The global variables and formal parameters used to convey these values we describe as the *imported variables* of P.

Note on terminology: imported variables may be formal parameters of mode **in**, which in Ada terms are constants rather than variables. In the remainder of this subsection, we use the term *variable* to include formal parameters as well as objects declared by a variable declaration.

In the early stages of the design of a procedure one chooses its exported variables, and determines which (initial values of) imported variables may be required by the procedure, to derive (the final value of) each exported variable. In using SPARK, this information is given with the procedure specification. If any of the imported or exported variables are global variables rather than formal parameters of the procedure, their names and modes of use within the procedure are given in a *global definition;* also, the imported variables from which each exported variable is derived may be specified in a *dependency relation.*

The syntax of global definitions and dependency relations is given below. It will be noted that these definitions and relations are specified in terms of *entire variables*, i.e. variables that are not subcomponents of composite variables.

```
+   global_definition ::=
        --# global [global_mode]  global_variable_list ; { [global_mode] global_variable_list ; }
+   global_mode ::= in | in out | out
+   global_variable_list ::= global_variable { , global_variable }
+   global_variable ::= entire_variable
+   entire_variable ::= [ package_name . ]  direct_name
+   dependency_relation ::=
        --# derives [dependency_clause { & dependency_clause } [& null_dependency_clause]] ;
    |   --# derives null_dependency_clause ;
+   dependency_clause ::=
        exported_variable_list from [ imported_variable_list ]
+   exported_variable_list ::= exported_variable { , exported_variable }
+   exported_variable ::= entire_variable
+   imported_variable_list ::= *  |  [ * , ] imported_variable { , imported_variable }
+   imported_variable ::= entire_variable
+   null_dependency_clause ::= null from imported_variable { , imported_variable }
```

The modes in a global definition are analagous to those in a formal parameter specification. For example,

```
--# global in I; in out J, K; out L;
```

states that the global variable `I` is an import of the procedure, `J` and `K` are both imports and exports, and `L` is an export.

An example of a dependency relation is

```
--# derives A, B, C from X, Y, Z;
```

which states that the final value of each of the exported variables `A`, `B` and `C` is derived from the initial values of the imported variables `X`, `Y` and `Z`.

The use of * in an imported variable list is a convenient abbreviation for the case where each exported variable in the corresponding exported variable list depends on itself, i.e. its exported value is derived from its imported value. If present, it must be the first (or only) item in the imported variable list. The * notation is permitted even if the variable it represents is already present in the imported variable list.

For example, the dependency relation

```
--# derives A, B, State from *, State;
```

is an abbreviation for

```
--# derives A     from A, State &
--#         B     from B, State &
--#         State from State;
```

Note that annotations can be broken across multiple lines as described in section 2.11.

Where annotations indicate the importing of external variables (see Section 7) it is sometimes the case that the subprogram simply consumes values from the external variable without using the values read to derive any entity within the SPARK program itself. For example, a "busy wait" procedure can be viewed as simply consuming clock cycles without exporting any value to its calling environment. An extension to the derives annotation, a *null_dependency_clause*, is used to describe such cases.

For example, a delay procedure which reads an external clock and waits X clock ticks before returning can be specified as follows:

```
procedure Delay (X : in Clock.Ticks);
--# global in Clock.State; – an external variable
--# derives null from X, Clock.State;
```

The appearance of the identifier **null** can be taken as meaning that nothing *visible inside the SPARK program* is derived from the imports associated with the null export.

Whereas a procedure subprogram is designed to update variables in its calling environment, in SPARK the execution of a function subprogram is not allowed to have any *side-effects*, i.e. it shall not update any global variables, directly or indirectly. In our terminology, *a function subprogram cannot have any exported variables.* It may have imported variables other than its formal parameters, i.e. in its execution it may read some global variables directly - in which case the function specification must be followed by a global definition naming all those variables. However, since a function has no exported variables, its associated global definition may take a simpler form which simply lists the names of the variables. The mode of these variables may be given, but it can only be mode "in", for consistency with imported formal parameters. Moreover, since the data flow between a function subprogram and its calling environment is completely prescribed by its designator, formal part and global definition, the specification of a function subprogram does not contain a dependency relation. Note that a function without either formal parameters or a global definition is permitted; such a function effectively has a constant return value.

A function may globally import an external variable of mode **in** but this imposes limitations on how calls of the function may be used. See Sections 5.2, 6.4.1, 6.5 and 7.

An example illustrating the use of these annotations is given in Figure 1.

The following rules apply to the formal parameters, global definition and dependency relation of a subprogram (or main program - see Section 10.1.1):

1 A name cannot appear more than once in the same global definition.

2 The name of a variable V can only appear in the global definition of a subprogram or main program P if

- P and V are local to the same declarative region (including the case where V is a formal parameter of a subprogram or main program that immediately encloses P), or
- the name of V appears in the global definition of a subprogram or main program that immediately encloses P, or
- V is an own variable (see Section 7.1.3) of a package Q, and P and Q are local to the same declarative region, or
- V is an own variable of a package that immediately encloses P, or
- V is an own variable of a private child of a package that immediately encloses P or an own variable of a public descendant of such a private child, or
- V is inherited (see Section 7.1.1) by a package that immediately encloses P, or
- P is the main program and V is inherited by P.

3 If the name of a variable V appears in the global definition of a procedure subprogram P, and V is a formal parameter of mode **in** of a subprogram or main program that immediately encloses P, or the name of V appears with mode **in** in the global definition of such a subprogram or main program, then the mode of V in the global definition of P shall also be **in**.

4 A name in the global definition of a subprogram or main program P shall not be redeclared immediately within P or within a loop statement whose nearest enclosing program unit is P.

5 Every variable name that appears as an imported variable in the dependency relation of a procedure subprogram or main program P shall either denote a formal parameter of P of mode **in** or **in out** or shall appear in the global definition of P with mode **in** or **in out**.

6 Every variable name that appears as an exported variable in the dependency relation of a procedure subprogram or main program P shall either denote a formal parameter of P of mode **in out** or **out** or shall appear in the global definition of P with mode **in out** or **out**.

7 External variables with mode **out** may only appear as exports in a dependency relation and may only have global mode **out**. Conversely, external variables with mode **in** may only appear as imports and may only have global mode **in**.

8 If a procedure subprogram or main program P has a dependency relation, every formal parameter of P and every variable in the global definition of P shall appear at least once in the dependency relation, as an imported variable or an exported variable. Moreover, every formal parameter of mode **in out** and every variable which appears in the global definition with mode **in out** shall appear in the dependency relation as both an imported variable and an exported variable.

9 A name cannot appear more than once as an exported variable of a dependency relation. A name cannot appear more than once in the same imported variable list (but * is permitted even if the variable it represents is already present in the list).

```
procedure Interchange(A : in out Vector; M, N : IndexRange)
--#  derives A from A, M, N;
-- This procedure transfers the contents of A(1) .. A(M) into A(N+1) .. A(N+M) --
while simultaneously transferring the contents of A(M+1) .. A(M+N) into
-- A(1) .. A(N) without using an appreciable amount of auxiliary memory.   It
-- is an Ada version of ALGORITHM 284: INTERCHANGE OF TWO BLOCKS OF DATA by W.
-- Fletcher, Comm. ACM, vol. 9 (1966), p. 326.  The ACM publication explains
-- the algorithm.
is
  D, I, J, K, L, R : Integer;
  T : Float;

  function GCD(X, Y : Integer) return Integer
  is
    C, D, R : Integer;
  begin
    C := X; D := Y;
    while D /= 0 loop
      R := C mod D;
      C := D; D := R;
    end loop;
    return C;
  end GCD;

  procedure Swap(TempVal : in out Float; Index : IndexRange)
  --#  global in out A;
  --#  derives A       from A, TempVal, Index &
  --#          TempVal from A, Index;
  -- This procedure exchanges the values of TempVal and A(Index).
  is
    T : Float;
  begin
    T := A(Index); A(Index) := TempVal; TempVal := T;
  end Swap;
```

```
begin -- Interchange
  D := GCD(M, N);
  R := (M + N) / D;
  I := 1;
  while I <= D loop
    J := I; T := A(I);
    K := 1;
    while K <= R loop
      if J <= M then
         J := J + N;
      else
        J := J - M;
      end if;
      Swap(T, J);
      K := K + 1;
    end loop;
    I := I + 1;
  end loop;
end Interchange;
```

Figure 1: An extract from a SPARK program, illustrating the use of global definitions and dependency relations.

6.2 Formal Parameter Modes

The rules of SPARK, in particular the rules to prohibit aliasing in the execution of procedures (see Section 6.4), prevent the possibility of assigning to an object via one access path and then reading its value via a distinct access path. This ensures that the effect of the program will not depend on whether the parameter is passed by copy or by reference.

6.3 Subprogram Bodies

SPARK and Ada subprogram bodies differ in the following respects. In SPARK,

1 A designator shall appear at the end of every subprogram body (repeating the designator of the subprogram specification).

2 The SPARK grammar rule for code_insertion allows the inclusion of code statements, as in Ada. If a subprogram implementation consists of code statements, the SPARK Examiner will report this fact, but it will effectively ignore them.

3 Rules governing the form and placement of return statements in subprograms are given in Section 6.5.

The implementation of a subprogram may be *hidden* from the SPARK Examiner, by means of a **hide** directive (see Annex M), though this is not part of the core SPARK language.

```
*   subprogram_body ::=
        procedure_specification
      [ procedure_annotation ]
        is
        subprogram_implementation
    |   function_specification
      [ function_annotation ]
        is
        subprogram_implementation
+   subprogram_implementation ::=
            declarative_part
        begin
                sequence_of_statements
        end designator ;
    |   begin
            code_insertion
        end designator ;
+   code_insertion ::= code_statement { code_statement }
```

A declaration of a procedure subprogram may contain a procedure annotation (as defined in Section 6.1). If such a procedure annotation contains a global definition, in which one or more variables are abstract (as defined in Chapter 7), then a second procedure annotation (which is a refinement of the first - c.f. Section 7.2.1) shall occur, in the body stub if this exists or in the procedure body otherwise; whereas if the procedure annotation in the procedure declaration contains no abstract variables, the procedure shall have only one procedure annotation.

If a subprogram declaration does not exist for a procedure, there shall be only one procedure annotation, occurring in the body stub if this exists, or in the procedure body otherwise.

A declaration of a function subprogram may contain a global definition (see Section 6.1). If it does, and one or more variables in this definition are abstract, then a second global definition (which is a refinement of the first) shall occur, in the body stub if this exists or in the subprogram body otherwise; whereas if the function subprogram declaration does not contain a global definition with abstract variables, the subprogram shall have no further global definitions.

If a subprogram declaration does not exist for a function subprogram, there shall exist at most one global definition for the subprogram, occurring in the body stub if this exists or in the subprogram body otherwise.

In a SPARK subprogram body, parameters and global variables of the subprogram which are not exported by it shall not be updated, directly or indirectly.

Using knowledge of the imported and exported variables of a subprogram, its body may be analysed to determine whether it is free of certain anomalies. (For a description of the analyses see Bergeretti and Carré (1985)). These anomalies may be classified as follows:

1 A statement S in which the value of a variable V is read when it is undefined, that is, V is not imported by the subprogram and no path from the start of the subprogram to S includes a statement that updates V. A program containing such a statement is not a legal SPARK program.

2 A statement S in which the value of a variable V is read when it *may be* undefined, that is, V is not imported by the subprogram and there exists a path from the start of the subprogram to S that does not include a statement that updates V. This represents a programming error unless the programmer can show, by reasoning (formally or otherwise) about the program's dynamic semantics, that all such paths are non-executable.

3 An unset exported value, that is, an exported variable which is not updated on any path through the subprogram. This renders the program illegal in SPARK.

4 A potentially unset exported value, that is, an exported variable which is not updated on all paths through the subprogram. This represents a programming error unless the programmer can show by reasoning that the paths on which the variable is not updated are non-executable.

5 Information flow relations of the subprogram body which are not consistent with its dependency relation (given explicitly for a procedure subprogram, and implicitly for a function subprogram). This indicates a difference between the stated intention of the subprogram and its implementation.

6 Others such as ineffective statements and invariant expressions in conditions, which may be unintended by the programmer and could be indicative of programming errors.

The SPARK Examiner performs the analysis described above and reports any discrepancies.

6.4 Subprogram Calls

In SPARK, positional and named parameter associations shall not both be used in the same subprogram call.

* procedure_call_statement ::=
*procedure*_name [actual_parameter_part] ;
* function_call ::=

```
        function_ name [ actual_parameter_part ]
*   actual_parameter_part ::= ( parameter_association_list )
+   parameter_association_list ::=
        named_parameter_association_list  |  positional_parameter_association_list

+   named_parameter_association_list ::=
        formal_parameter_selector_name => explicit_actual_parameter
            { , formal_parameter_selector_name => explicit_actual_parameter }
+   positional_parameter_association_list ::=
        explicit_actual_parameter { , explicit_actual_parameter }
    explicit_actual_parameter ::= expression | variable_name
```

The rules below prevent aliasing of variables in the execution of procedure subprograms. See Section 6.1.2 for the definitions of imported, exported and entire variables. (If a procedure subprogram has two procedure annotations as a consequence of refinement (c.f. Chapter 7), then in applying the rules to calls of a procedure P occurring outside the package in which P is declared, the annotation in the declaration should be employed; whereas in applying the rules to calls within the body of this package, the annotation in the procedure body or body stub should be used.)

1 If a variable V named in the global definition of a procedure P is exported, then neither V nor any of its subcomponents can occur as an actual parameter of P.

2 If a variable V occurs in the global definition of a procedure P, then neither V nor any of its subcomponents can occur as an actual parameter of P where the corresponding formal parameter is an exported variable.

3 If an entire variable V or a subcomponent of V occurs as an actual parameter in a procedure call statement, and the corresponding formal parameter is an exported variable, then neither V nor any subcomponent of V can occur as another actual parameter in that statement.

Where one of these rules prohibits the occurrence of a variable V or any of its subcomponents as an actual parameter, the following constructs are also prohibited in this context:

1 a type conversion whose operand is a prohibited construct;

2 a qualified expression whose operand is a prohibited construct;

3 a prohibited construct enclosed in parentheses.

In SPARK every call of a subprogram, in the compilation unit where its proper body or body stub is declared, shall follow that declaration. A subprogram declared in a package shall not be called in a private child of that package or in any descendant of such a private child. These rules, together with the rules of Sections 3.11 and 7.1.1, imply that subprograms cannot be called recursively.

6.4.1 Parameter Associations

Only an explicit type conversion which is a view conversion (see section 4.6) may be used as an actual parameter in a subprogram call where the corresponding formal parameter is of mode **in out** or **out.**

If a formal parameter is of a constrained array subtype, the upper and lower bounds of the corresponding actual parameter (for each index position) must be equal to those of the formal parameter. (This follows from the rules on type conversion in Section 4.6.)

An external variable (see Section 7) or a function which references an external variable, may not be used as an actual parameter.

Actual parameters matching formals of tagged types must be objects (or view conversions of objects) not general expressions.

6.5 Return Statements

The last statement in the sequence of statements of a function subprogram shall be a return statement, which shall include an expression.

* return_statement ::= **return** expression ;

No other occurrences of return statements are allowed in SPARK.

If the result subtype of a function is a constrained array subtype, the expression in the return statement in the function subprogram body must have upper and lower bounds (for each index position) equal to those of that subtype. (This follows from the rules on type conversion in Section 4.6.)

External variables, and functions which reference external variables, may not be used within expressions in return statements although they may appear alone in such statements. This restriction is to avoid the introduction of ordering effects in the reading of external devices.

6.6 Overloading of Operators

In SPARK user-defined operators are not permitted.

Renaming declarations are allowed for operators (see Section 8.5).

7 PACKAGES

SPARK has several important concepts associated with packages which do not exist in Ada.

Package Inheritance The *inheritance* of one package by another (or by the main program) is achieved by naming the package to be inherited in the *inherit clause* (an annotation) of the recipient package, in much the same way as the with clause is employed to specify dependencies between compilation units. (The conditions under which inheritance is permitted are specified in Section 7.1.1). Within a package, the visibility of declarations occurring outside the package is restricted to entities declared in those packages which it inherits. The principal reason for employing this form of inheritance is that, whilst the Ada package features provide satisfactory control of visibility from the exterior, of the contents of a package, visibility from within of declarations outside a package is largely determined by the context of the package declaration; it cannot easily be controlled or even be made explicit. The rules of inheritance provide a relatively simple yet quite precise means of specifying, and controlling, the access to external entities (the consistency of inherit clauses with code being checked by the SPARK Examiner).

Own Variables of Packages The concepts of “own variables” and “refinement” (discussed below) are particularly relevant to the design of Ada programs in terms of “abstract state machines” or ASM's (Booch, 83), a machine being implemented by a package, with its state being represented by variables declared within the package.

To introduce these concepts, the text below consists of a package P, whose body contains a declaration of a variable V, and a procedure A which reads and updates this variable. When the procedure A is called, by a procedure to which P is visible, the variable V is read and updated, despite the fact that it is not visible at the place where A is declared, or where it is called. In other words, the call of A has a *side-effect*.

```
package P is
  .....
  procedure A;
  --# global in out ???;
  --# derives ??? from ???;
  .....
end P;

package body P is
  .....
  V : XXX;
  .....

  procedure A;
  -- This procedure reads and updates the variable V.
  .....
  end A;
  .....
 end P;
```

To make explicit the fact that the procedure A reads and updates V (which is essential, if we are to verify that V is employed as intended), we must first make evident the existence of this "state variable", in those parts of the program whose execution may directly or indirectly cause it to be read or updated. This can be achieved by naming the state variable V in an *own variable clause*, an annotation placed immediately before the visible and private parts of the package.

```
package P
--# own V;
is
  .....
  procedure A;
  --# global in out V;
  --# derives V from V;
  .....
end P;

package body P is
  .....
  V : XXX;
  .....

  procedure A;
  -- This procedure reads and updates the variable V.
  .....
  end A;
  .....
end P;
```

An own variable clause of a package has the visibility, *within annotations*, of a declaration occurring in the the visible part of the package. If an own variable clause of a package P names a variable V, its effect (with regard to annotations) is to raise the declaration of V to a place in the visible part of P that precedes all required occurrences of the name of this variable.

With this convention, the dependency relation of the procedure A in our example can be constructed, according to the rules of Section 6.1. This in turn will allow information flow analysis of those parts of the program which employ the package P, taking into account the reading and updating of V.

Refinement The above example of the use of an own variable annotation is unusually simple, in that the "abstract state machine" package P has only one variable, V. In practice, the following circumstances arise.

- In writing the specification of an abstract state machine (ASM) package, we should have clear notions of the purpose of the procedures and function subprograms to be declared in this specification, and of how these subprograms are to read and/or update the "state" (or values of the variables) of the ASM. However, until the package body has been designed in detail, we may not know exactly how the ASM state will be represented: the implementation of the ASM may eventually require a number of objects of scalar or composite types, and possibly even some further ASM's. Thus it may be very difficult, initially, to list all the state variables and to

define precisely the dependency relations of the subprograms declared in the package specification.

- Even if the specification of ASM's to this level of detail could be achieved at an early stage, it would often be counter-productive. For in analysing code that employs an ASM (i.e. code containing calls of subprograms of the ASM), it is important to take into account all the uses and modifications of the state of the ASM, but the details of its concrete representation are not relevant: these are significant only in analysing the code of the ASM itself. Inclusion of these details in the specification of the ASM would make the dependency relations of subprograms declared there cumbersome, and furthermore, it would greatly complicate the dependency relations of all subprograms that used the ASM, directly or indirectly.

Both these difficulties can be overcome by employing the method of *refinement.* In constructing the specification of an ASM we can describe its state and annotate its visible subprograms in terms of own variables that may be either *concrete* (i.e. variables declared immediately within the package, as in the above example), or *abstract* (in which case their names play the role of visible representatives of variables or collections of variables or even ASM's eventually to be declared in the body of the ASM being specified). When the body of an ASM is written, each abstract own variable of its specification appears there, as the subject of a *refinement clause*, in an annotation of the form

```
--# own V1 is W11, W12, ... , W1p &
--#     V2 is W21, W22, ... , W2q &
.................................
--#     Vn is Wn1, Wn2, ... , Wnr ;
```

Here the refinement clauses give for each abstract own variable Vi a list of its *constituents* Wij, these being names either of variables declared immediately within the package body, or of (abstract or concrete) own variables of ASM's declared immediately within this body or as private children of the package (or public descendents of such private children).

As a very simple example of refinement, the following is the specification of a stack ASM in which the state of the ASM is represented by a single abstract own variable called “State”. Note that the dependency relations of the subprograms are given in terms of this abstract variable.

```
package Stack
--# own State;
is
  function Empty return Boolean;
  --# global  State;

  procedure Clear;
  --# global out State;
  --# derives State from  ;

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

```
  procedure Push(X : in Integer);
  --# global in out State;
  --# derives State from State, X;
end Stack;
```

In the body of the package, given below, the abstract variable State is represented by two concrete variables, Pointer and Vector, which are declared within this body. The text of the body begins with a refinement annotation, associating State with Pointer and Vector.

If a subprogram declaration in the visible part of a package has a global definition containing one or more abstract own variables of the package, then its body, which appears in the package body, must also have a global definition (called the *refinement* of the original one). Each abstract variable occurring in the original definition is replaced by one or more of its constituents in the new one. In the case of a procedure subprogram, the new global definition is accompanied by a new dependency relation - again called a refinement of the original one - describing the dependencies between the imports and exports of the procedure, as represented by its (concrete) parameters and its (abstract or concrete) global variables. Rules of consistency of refinements, of global definitions and dependency relations, are given in Section 7.2.1.

```
package body Stack
--# own State is Pointer, Vector;
is
  StackSize : constant Integer := 100;
  subtype PointerType is Integer range 0 .. StackSize;
  Pointer : PointerType;
  subtype IndexRange is Integer range 1 .. StackSize;
  type VectorType is array (IndexRange) of Integer;
  Vector : VectorType;

  function Empty return Boolean
  --# global Pointer;
  is
  begin
    return Pointer = 0;
  end Empty;
```

```
  procedure Clear
  --# global out Pointer, Vector;
  --# derives Pointer from  &
  --#         Vector  from  ;
  is
  begin
    Pointer := 0;
    Vector := VectorType'(IndexRange => 0);
  end Clear;

  procedure Pop(X : out Integer)
  --# global in out Pointer; in Vector;
  --# derives Pointer from Pointer &
  --#         X       from Pointer, Vector;
  is
  begin
    X := Vector(Pointer);
    Pointer := Pointer - 1;
  end Pop;

  procedure Push(X : in Integer)
  --# global in out Pointer, Vector;
  --# derives Pointer from Pointer &
  --#         Vector  from Pointer, Vector,   X;
  is
  begin
    Pointer := Pointer + 1;
    Vector(Pointer) := X;
  end Push;
end Stack;
```

External Variables Where an own variable represents a connection between the SPARK program and its external environment it may be given a mode indicating whether it is to be regarded as an input (mode **in**) from the environment or an output to it (mode **out**). Mode **in out** is not permitted. Own variables with such modes are called external variables. Special rules apply to external variables in order to capture correctly their volatile nature. Refinement constituents may themselves be given modes to indicate that they are external variables.

External variables are regarded as *volatile*: successive reads of a external variable are considered to return potentially different values and successive writes to external variables are not regarded as ineffective. This behaviour applies regardless of whether values are referenced or updated indirectly via subprogram calls or directly via assignment or return statements.

External variables, and functions which reference external variables, may only be used directly in assignment and return statements. They may not be used in expressions, conditionals or as actual parameters.

An example of the use of external variables in a device driver package is given in Figure 3 starting on page 60.

7.1 Package Specifications and Declarations

SPARK and Ada package structure differ in the following respects. In SPARK,

1 a package declaration contains an optional inherit clause (described in Section 7.1.1 below) before the package specification;

2 a package specification consists of the package name, a package annotation (described in Sections 7.1.2 - 7.1.4), a visible part and an optional private part;

3 a package specification must end with the name of the package.

```
*   package_declaration ::= [ inherit_clause ] package_specification ;
+   private_package_declaration ::= [ inherit_clause ] private package_specification ;
*   package_specification ::=
        package defining_program_unit_name
            package_annotation
        is
            visible_part
      [ private
            private_part ]
        end [ parent_unit_name . ] identifier
+   visible_part ::=
        { renaming_declaration }
        { package_declarative_item }
+   private_part ::=
        { renaming_declaration }
        { package_declarative_item }
+   package_declarative_item::=
        basic_declarative_item  |  subprogram_declaration
                                |  external_subprogram_declaration
```

The visible and private parts of a package specification each consist of a list of basic declarative items and subprogram declarations, optionally preceded by renaming declarations for operators inherited by the package (See Section 8.5). A subprogram declaration may be qualified by an immediately following pragma Import (see Annex B.1).

The private part of a package specification may be *hidden* from the SPARK Examiner by means of a hide directive (see Annex M), though this is not part of the core SPARK language.

It is to be noted that in SPARK, a package specification cannot contain package declarations, but packages can still be declared within package bodies.

A further illustration of the specification of SPARK packages is given in Figure 2.

7.1.1 Inherit Clauses

A package declaration (or a main program) may begin with an inherit clause.

```
+   inherit_clause ::=
        --# inherit package_name { , package_name } ;
```

Note that annotations can be broken across multiple lines as described in section 2.11.

A package (or main program) P is said to *inherit* a package Q if the inherit clause of P contains a name (or a prefix) denoting Q. In addition, all packages (and the main program) are deemed to inherit the package Standard without it being named in an inherit clause.

A package (or main program) P can inherit a package Q only if

- the declaration of P is within the scope of Q or Q is a library package contained in the program library, and
- every package (or main program) whose body contains the declaration of P, but not that of Q, inherits Q.

In addition, a package P can inherit a private child of a package Q only if P is also a private child of Q or a descendant of such a child (or if Q is package Standard).

Furthermore, mutual inheritance is forbidden, that is, it is not permissible to have a sequence of packages P1, ..., P*n*, such that each package inherits its successor in the sequence and P*n* inherits P1.

We define the set of packages *owned* by a library package (other than package Standard) to be the private children of P and their public descendants. If a package is owned by a library package P, then the only packages it may inherit are P itself, other packages owned by P, and packages inherited by P.

A name in a package (or main program) P can only denote an entity declared outside the declarative region of P if it is

- a package inherited by P, or
- an entity declared in the visible part of a package inherited by P, or
- an entity declared in the private part of a package which is inherited by P, and of which P is a private descendant, or
- an entity declared in the private part of a package which is inherited by P, and of which P is a public descendant, provided the name does not occur in the visible part of P, or
- an entity declared in the private part or body of a package which is inherited by P, and whose body includes the declaration of P, or
- an own variable of a package which is inherited by P (wherever that variable may be declared within the inherited package), when its name occurs in an annotation within P (see Section 7.1.3).

We may describe any such entity as being inherited by the package P.

An entity E of a package Q (other than package Standard), inherited by a package P, shall be denoted in P by Q.E. Denotations of entities of package Standard are not prefixed with their package name. A child package is denoted by a direct name at places where its declaration is directly visible (for example within the body of its parent).

7.1.2 Package Annotations

A package annotation may contain an own variable clause (described in the next Section), and if so, it may also contain an initialization specification (see Section 7.1.4).

```
+  package_annotation ::=
       [ own_variable_clause [ initialization_specification ] ]
```

7.1.3 Own Variable Clauses

The names that occur in an own variable clause of a package are called the *own variables* of that package. Where such an own variable has a mode it is called an *external variable*.

```
+  own_variable_clause ::=
       --# own own_variable_list ;
+  own_variable_list ::= mode own_variable { , mode own_variable }
+  own_variable ::= direct_name
```

A name cannot occur more than once in the same own variable clause.

Every own variable of a package shall occur either

- in a variable declaration immediately within the declarative region of the package, or
- as a subject of a refinement definition of the package (c.f. Section 7.2.1),

but not both. Own variables which occur in variable declarations are described as concrete own variables, whereas own variables which are subjects of refinement definitions are said to be *abstract.*

The name of a variable whose declaration occurs immediately within the declarative region of a package shall be a (concrete) own variable of the package if and only if it is not a constituent of a refinement definition of the package (c.f. Section 7.2.1). All variables declared in a package specification shall be (concrete) own variables of the package.

All subjects of a refinement definition of a package shall be (abstract) own variables of that package. The name of an abstract own variable of a package shall not be the subject of any declaration (variable or otherwise) which occurs immediately within the declarative region of the package.

An own variable clause is only visible within annotations. Subject to this restriction the visibility of an own variable clause, outside its package, is that of a declaration in the visible part of the package. Within the specification of its package, the own variable clause is visible in all annotations; and within the body of its package, the clause is visible only within the refinement definition (if this exists). An own variable clause of a library package P is not visible within a descendant package owned (see Section 7.1.1) by P.

The following rules ensure that inconsistencies between external variable modes are not introduced by refinement:

1 Refinement constituents may not be of mode **in out**.

2 Refinement constituents of moded own variables must have the same mode as their subject.

3 Refinement constituents of unmoded own variables can have any or no mode (excluding **in out**).

4 Refinement constituents which refine to own variables of private child packages must have the same mode as that given to the own variable in the child package.

5 Own variables of embedded packages which have been announced in a previous refinement clause must have the same mode as was given in that refinement clause.

7.1.4 Package Initializations and Initialization Specifications

An initialization specification is an annotation in a package specification whose purpose is to indicate which variables are to be initialized by the elaboration either of the package specification or of its body, ie are to be updated by a package initialization or initialized at their declarations.

```
+ initialization_specification ::=
      --# initializes own_variable_list ;
```

External variables may not appear in an initialization specification (such variables are considered to be implicitly initialized by the environment to which they are connected).

A variable whose declaration occurs immediately within the declarative region of a package shall be updated by the sequence of statements of the package initialization or initialized at its declaration (but not both) if and only if

- it is an own variable of the package, named in its initialization specification, or
- its name occurs, without a mode, in the constituent list of a refinement clause, whose subject is named in the initialization specification.

If an abstract own variable of a package occurs in an initialization specification of the package, and any refinement constituent of this variable is an own variable of another package, declared immediately within the body or as (a public descendant of) a private child of the first one, then the constituent shall be named in an initialization specification of the embedded or descendant package.

Conversely, every own variable which occurs in the initialization specification of a package declared immediately within the body of another package or as (a public descendant of) a private child of another package shall occur as a constituent of a refinement definition of the enclosing or ancestor package, and the subject of the refinement clause to which the consituent belongs shall occur in an initialization specification of the enclosing or ancestor package.

7.2 Package Bodies

SPARK and Ada package bodies differ in the following respects. In SPARK,

1 a package body must end with the name of the package;

2 a package body may begin with a refinement definition (described in Section 7.2.1).

```
*  package_body ::=
       package body defining_program_unit_name
     [ refinement_definition ]
       is
         package_implementation
       end [ parent_unit_name . ] identifier ;
+  package_implementation ::=
       declarative_part
     [ begin
         package_initialization ]
+  package_initialization ::=
       sequence_of_statements
```

The package implementation consists of a declarative part, possibly followed by a package initialization. By use of the **hide** directive (not part of the core SPARK language - see Annex M), either the package initialization or the entire package implementation may be *hidden* from the SPARK Examiner.

A variable can only be updated by the sequence of statements of a package initialization if its declaration occurs immediately within the declarative region of the package and it is not an external variable. In a package initialization, user-defined subprograms cannot be called and no variable declared immediately within another package can be read or updated.

7.2.1 Refinements

Every abstract own variable of a package shall be the subject of exactly one refinement clause of a refinement definition of the package (i.e. a refinement definition which occurs in the package body, before its declarative part).

```
+  refinement_definition ::=
       --# own refinement_clause { & refinement_clause } ;
+  refinement_clause ::=
       subject is constituent_list
+  subject ::= direct_name
+  constituent_list ::= mode constituent { , mode constituent }
+  constituent ::= [ package_name . ] direct_name
```

The constituents of the clauses of a refinement definition of a package shall together comprise

- the set of all names of variables whose declarations occur immediately within the declarative region of the package, but which are not own variables of the package, together with
- the set of all own variables of packages whose declarations occur immediately within the body of the package, and
- the set of all own variables of private children of the package and their public descendants; such own variables must be denoted in the refinement definition using a full hierarchic prefix, ie starting with the name of the appropriate root library package.

No name shall appear more than once in a refinement definition.

If and only if a subprogram declaration in a package specification has a global definition containing one or more abstract own variables of the package, then the body of the subprogram, which occurs in the package body, shall also have a global definition, called a *refinement* of the original one.

A refinement *G'* of a global definition *G*, which does not contain any global modes, shall be reducible to *G* by replacing all constituents of refinement clauses by the subjects of those clauses and removing any duplicates that result.

A refinement *G'* of a global definition *G*, which does include global modes, shall be reducible to *G* by replacing all constituents of refinement clauses by the subjects of those clauses, and setting the mode of each subject as follows:

1 For each subject whose constituents appear in *G'* with two or more different modes, the mode of the subject is set to **in out**.

2 For each subject whose constituents appear in *G'* only with mode **out** but which have at least one constituent absent from *G'*, the mode of the subject is set to **in out**.

3 For each subject in *G* which is not an external variable but which has constituents which are external variables and which appear in *G'*, the global mode of the subject is set to **in out**.

4 Otherwise the mode of the subject is set to the (common) mode of its constituents in *G'*.

For a procedure subprogram whose declaration includes a dependency relation, a refinement of a global definition shall be accompanied by a dependency relation, again called a *refinement* of the original one. A refinement *D*' of a dependency relation *D* shall be reducible to *D* by the successive application of the five following operations. (In this description, the set of constituents of a refinement of an own variable *V*i is denoted by *C(Vi)*).

1 For each export *E* of *D'* in turn, if *E* is a constituent of a refinement of an own variable *Vi* then for every constituent *W* in *C(Vi)* which is not an export of *D'*, a dependency clause with export *W* and import *W* is added to *D'*.

2 If *E* is a constituent which is an external variable of mode **out** whose subject is not an external variable, then *E* is added as an import.

3 For each import *I* in *D'* where *I* is a refinement constituent which is an external variable of mode **in** whose subject is not an external variable, add a new dependency clause to *D'* showing that *I* is derived from *I*.

4 All dependency clauses whose exports belong to the same set of constituents *C(Vi)* are combined into a single clause, whose export is *V*i and whose imports are all the imports of the original clauses.

5 Wherever the imports of a clause include members of a set of constituents *C(Vi)*, these are removed and replaced by *Vi*.

7.3 Private Types and Private Extensions

SPARK and Ada private type declarations differ in that a private type declaration in SPARK cannot have a discriminant part.

* private_type_declaration ::=

type defining_identifier **is [tagged] [limited] private** ;
* private_extension_declaration ::=
type defining_identifier **is new** *ancestor*_subtype_indication **with private** ;

7.3.1 Private Operations

In SPARK attributes of a private type are not allowed unless the corresponding full type declaration is visible.

7.4 Deferred Constants

SPARK does not permit a deferred constant declaration to be completed by a pragma Import.

7.6 User-Defined Assignment and Finalization

SPARK does not have controlled types and hence there are no user-defined initialization, assignment or finalization operations. The package Ada.Finalization is not predefined in SPARK.

```
package RealNumbers is
  type Real is digits 6;
end RealNumbers;

with RealNumbers;
--# inherit RealNumbers;
package RandomNumbers
--# own Seed;
--# initializes Seed;
is
  procedure Random(X : out RealNumbers.Real);
  --# global in out Seed;
  --# derives X, Seed from Seed;
end RandomNumbers;

package body RandomNumbers is
  subtype Pos_31 is Integer range 0 .. 2**30 - 1;
  Seed : Pos_31;

  procedure Random(X : out RealNumbers.Real)
  is
  --# hide Random
    ... implementation of Random
  end Random;

begin
  Seed := 2**15 - 1;
end RandomNumbers;

with RealNumbers,
     RandomNumbers,
     SPARK_IO;
use type RealNumbers.Real;
--# inherit RealNumbers,
--#         RandomNumbers,
--#         SPARK_IO;
--# main_program;
procedure Main
--# global in out RandomNumbers.Seed, SPARK_IO.File_Sys;
--# derives RandomNumbers.Seed, SPARK_IO.File_Sys
--#    from *, RandomNumbers.Seed;
is
  X : RealNumbers.Real;
begin
  RandomNumbers.Random(X);
  SPARK_IO.Put_Integer(SPARK_IO.Standard_Output,
                       Integer(X * 10.0), 0, 10);
end Main;
```

Figure 2: An illustration of the specification of SPARK packages.

The following example uses external variables and refinement to describe and implement a complex input/output device. The device has internal state that records the last value sent to it. Its behaviour when *value* is written is as follows:

if value = last value sent then
do nothing
else
store value in last value
write value to out register
busy wait until ack received at status port.

The abstract specification of the device is:

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

And its body:

```
package body Device
--# own State is        OldX,       --  state variable constituent
--#              in     StatusPort, --  external variable constituent
--#                 out Register;   --  external variable constituent
is
  OldX : Integer := 0; --  only component that needs or permits  initialization

  StatusPort : Integer;
  for StatusPort'Address use .........;
  Register : Integer;
  for Register'Address use ..........;

  procedure WriteReg (X : in Integer)
  --# global out Register;
  --# derives Register from X;
  is
  begin
    Register := X;
  end WriteReg;
```

```
  procedure ReadAck (OK : out Boolean)
  --# global in StatusPort;
  --# derives OK from StatusPort;
  is
    RawValue : Integer;
  begin
    RawValue := StatusPort; -- only assignment is allowed here
    OK := RawValue = 16#FFFF_FFFF#; -- ack value
  end ReadAck;

  procedure Write (X : in Integer)
  --# global in out OldX;
  --#           out Register;
  --#        in     StatusPort;
  --# derives OldX,
  --#         Register from OldX, X &
  --#         null from StatusPort; -- see Section 6.1.2
  is
    OK : Boolean;
  begin
    if X /= OldX then
      OldX := X;
      WriteReg (X);
      loop
        ReadAck (OK);
        exit when OK;
      end loop;
    end if;
  end Write;
end Device;
```

Figure 3: An illustration of the use of external variables

8 VISIBILITY RULES

8.3 Visibility

In SPARK a user-defined subprogram shall not overload any other subprogram; however, an inherited root subprogram may be *overridden*. To prevent unintenional overloading, a subprogram declaration may not have the same name as a potentially inheritable subprogram unless it successfully overrides it

The associations between declarations and occurrences of identifiers and the places where particular identifiers can occur are governed by the scope and visibility rules of Ada, with the following additional restrictions:

1 In subprogram implementations and subprogram calls, the occurrences of variable and formal parameter names shall be consistent with the global definitions and dependency relations of the subprograms (as prescribed in Section 6.1.2 and Section 6.3).

2 In a package (or main program), the occurrences of identifiers that denote entities declared outside the package (or main program) shall be subject to the rules of inheritance given in Section 7.1.1.

3 In a package initialization, no variables shall be read or updated other than those declared immediately within that package (see Section 7.2).

4 At a place where a declaration of an entity is directly visible, its denotation shall not have a prefix, unless the entity is inherited there and is not a package, in which case it shall be denoted as a selected component of the package in which it is declared.

5 An identifier cannot be redeclared at a place where a declaration of it is already directly visible, unless

- the new place of declaration is in a subprogram and the visible declaration is a variable declaration or a parameter specification that occurs outside that subprogram, or
- the new place of declaration is in a package and the visible declaration occurs outside that package, or
- the new declaration is a component declaration in a record type definition.

6 An identifier cannot be redeclared at a place where it denotes a package inherited by the closest surrounding package or main program.

7 Neither the identifier Standard nor any identifier which is predefined in the package Standard shall be redeclared.

8.4 Use Clauses

In SPARK **use** (package) dauses are not allowed, and **use type** clauses are subject to certain restrictions.

use_type_clause ::= **use type** subtype_mark { , subtype_mark } ;

A **use type** clause may appear only in a context clause (Section 10.1.2) or in an embedded package declaration (Section 3.11).

The type determined by a subtype mark of a **use type** clause shall not be a limited private type.

A subtype mark shall not appear in a **use type** clause if all primitive operators of the associated type are already directly visible within the scope of the **use type** clause.

The type determined by a subtype mark of a **use type** clause in an embedded package declaration shall be a type declared in the associated package.

8.5 Renaming Declarations

In SPARK, the only renaming declarations are those for subprograms and (child) packages.

```
*  renaming_declaration ::=
        package_renaming_declaration
     |  subprogram_renaming_declaration
```

8.5.1 Object renaming declarations

SPARK does not have object renaming declarations.

8.5.2 Exception renaming declarations

SPARK does not have exception renaming declarations.

8.5.3 Package renaming declarations

In SPARK, package renaming declarations are used strictly for renaming child packages with their original names (devoid of ancestor-name prefixes), at places where those packages are not directly visible.

```
*  package_renaming_declaration ::=
        package defining_program_unit_name
           renames parent_unit_name . package_direct_name ;
```

A package renaming declaration shall occur only immediately within the declarative part of a package or main program that inherits the renamed package.

Within the scope of the renaming declaration, the renamed package shall be denoted only by its new name.

8.5.4 Subprogram renaming declarations

In SPARK, subprogram renaming declarations are used strictly for renaming subprograms (including operators but not enumeration literals), declared immediatedly within packages, with their original names (devoid of package-name prefixes), at places where the subprograms are not directly visible.

A renaming declaration can only apply to a subprogram declared in a package P if either

1 the renaming declaration occurs in an embedded package declaration which declares P, or

2 the renaming declaration occurs immediately within the declarative part of a package or main program which inherits P, or

3 the renaming declaration applies to an operator and occurs immediately within the visible part or private part of a package which inherits P.

Where a renaming declaration applies, the renamed operator or subprogram can only be denoted by its new name. In a renaming declaration of a subprogram, the formal part (and type mark in the case of a function) of the subprogram specification shall be the same as those of the renamed subprogram.

As a consequence of the prohibition of selectors as operator symbols (see Section 4.1.3), operators resulting from explicit type declarations must be renamed when they are inherited, unless made visible via a **use type** clause.

* subprogram_renaming_declaration ::=
function defining_operator_symbol formal_part **return** subtype_mark
renames *package_*name . operator_symbol ;
| function_specification
renames *package_* name . *function_*direct_name ;
| procedure_specification
renames *package_* name . *procedure_* direct_name ;

8.5.5 Generic renaming declarations

SPARK does not have generic renaming declarations.

8.6 The Context of Overload Resolution

Subprograms, enumeration literals, character literals and string literals have unique meanings in SPARK: by the rules of the language they cannot be overloaded. With their parameters, the significance of operators and basic operations is also completely determined.

9 TASKS

Tasks and multi-tasking constructs are not allowed.

10 PROGRAM STRUCTURE AND COMPILATION ISSUES

10.1 Separate Compilation

10.1.1 Compilation Units - Library Units

SPARK and Ada compilation units differ in the following respects. In SPARK

1 A subprogram declaration or body is not a library item.

2 The main program is distinct from a subprogram body (in the syntax) and is a library unit. It is distinguished by the presence of a `main_program` annotation. It has an inherit clause, and the rules of inheritance apply to its body (see Section 7.1.1)

3 Owing to the presence of an optional inherit clause, private package declarations are expressed differently in the syntax.

4 There are no library unit renaming declarations.

```
    compilation ::= { compilation_unit }
    compilation_unit ::=
        context_clause library_item | context_clause subunit
*   library_item ::= library_unit_declaration | library_unit_body
*   library_unit_declaration ::=
        package_declaration | private_package_declaration | main_program
*   library_unit_body ::= package_body
    parent_unit_name ::= name
+   main_program ::=
        [   inherit_clause ]
            main_program_annotation
            subprogram_body
+   main_program_annotation ::=
        --# main_program ;
```

All the imported global variables of the main program shall be initialized own variables (see Section 7.1.4) of packages inherited by the main program.

10.1.2 Context Clauses - With Clauses

In SPARK a context clause contains **with** clauses and **use type** clauses only, and no **use** (package) clauses. All units named in a **with** clause must be packages.

```
    context_clause ::= { context_item }
*   context_item ::= with_clause | use_type_clause
*   with_clause ::= with library_package_name { , library_package_name } ;
```

A package name cannot appear (directly) more than once in the **with** clauses of a given context clause.

A public descendant of a package P shall not be mentioned in a **with** clause of the body of P or any of its subunits.

Each subtype mark appearing in the **use type** clause(s) of a given context clause shall determine a different type.

10.1.3 Subunits of Compilation Units

In SPARK, a body stub for a procedure or function may require an appropriate annotation (see Section 6.3). Note that no such annotation occurs in the proper body of the corresponding subunit.

```
*   body_stub ::= subprogram_body_stub | package_body_stub
*   subprogram_body_stub ::=
        procedure_specification [ procedure_annotation ] is separate ;
      | function_specification [ function_annotation ] is separate ;
*   package_body_stub ::=
      | package body defining_identifier is separate ;
    subunit ::= separate ( parent_unit_name ) proper_body
```

10.2 Program Execution

10.2.1 Elaboration Control

The rules of SPARK make the pragmas Elaborate and Elaborate_All unnecessary.

However, as in Ada, pragma Elaborate_Body may be required in a SPARK package specification to make the package require a body.

11 EXCEPTIONS

Exceptions are not supported by SPARK.

12 GENERIC UNITS

Generic units are not allowed in SPARK other than the instantiation of the predefined generic Unchecked_Conversion.

12.1 Generic Declarations

Generic declarations are not allowed in SPARK.

12.2 Generic Bodies

Generic bodies are not allowed in SPARK.

12.3 Generic Instantiation

Generic instantiation of the predefined generic function Unchecked_Conversion is permitted in SPARK. No other predefined generics are recognised in SPARK so the only permitted instantiation is instantiation of a generic function.

* generic_instantiation ::= **function** defining_designator **is**
new *generic_function*_name [generic_actual_part]
generic_actual_part ::= (generic_association {, generic_association })
generic_association ::= [*generic_formal_parameter*_selector_name =>]
explicit_generic_actual_parameter
explicit_generic_actual_parameter ::= subtype_mark

13 REPRESENTATION ISSUES

13.1 Representation Items

Representation clauses may appear in SPARK texts. The SPARK Examiner checks their syntax, which must conform to the syntax rules given in Chapter 13 of the Ada *LRM*, but it ignores their semantics. A warning message to this effect is given whenever the SPARK Examiner encounters a representation clause.

13.3 Operational and Representation Attributes

SPARK does not support the operational attribute External_Tag.

13.7 The Package System

The Ada predefined library unit System is not automatically predefined in SPARK (see Annex A), nor are any of its descendants.

However, the SPARK version of package System, including the implementation-defined values that relate to the target Ada compilation system, may be specified via the target configuration file. The SPARK version of package System that may be specified in this way is defined below:

```
package System is

   type Address is private;

   Storage_Unit : constant := integer_value;
   Word_Size : constant := integer_value;

   Max_Int : constant := integer_value;
   Min_Int : constant := integer_value;

   Max_Binary_Modulus : constant := integer_value;

   Max_Base_Digits : constant := integer_value;
   Max_Digits : constant := integer_value;

   Fine_Delta : constant := real_value;
   Max_Mantissa : constant := integer_value;

   subtype Any_Priority is Integer range integer_range;
   subtype Priority is Any_Priority range integer_range;
   subtype Interrupt_Priority is Any_Priority range integer_range;

end System;
```

13.8 Machine Code Insertions

Code statements are permitted in SPARK, within a code insertion, as described in Section 6.3.

13.9 Unchecked Type Conversions

SPARK recognises the predefined generic function Unchecked_Conversion and permits instances of this. SPARK checks the static semantics of the instantiation but does not perform any of the dynamic semantic checks relating to the size and alignment of the actual subtypes used in the instantiation.

13.11 Storage Management

The package Ada.Finalization is not predefined in SPARK and there are no user-defined storage pools.

13.13 Streams

The package Ada.Streams is not predefined in SPARK and there are no stream types.

ANNEX A PREDEFINED LANGUAGE ENVIRONMENT

In SPARK, the only predefined packages are Standard, Ada, Ada.Characters and Ada.Characters.Latin_1.

The majority of the library units predefined in Ada95, including Ada.Direct_IO, Ada.Sequential_IO, Ada.Text_IO, Ada.Unchecked_Deallocation and System, use features not supported by SPARK and are not considered to be predefined. This allows the user to supply a specification of such packages, containing only SPARK features. Conversely, it also facilitates the declaration of a package which inherits a genuine Ada predefined library unit but has a visible part compatible with the rules of SPARK. All references to the Ada predefined library unit must then occur within hidden parts, representation clauses, or code statements of the private part or the body of the declared package.

Packages Standard and System in Ada95 each include implementation-defined values in their specifications. It is possible to define via the target configuration file the SPARK version of these packages that includes the actual values as specified by the target Ada compilation system. Indeed package System in SPARK becomes a predefined package if it is defined in this way (see section 13.7).

A.1 The Package Standard

In SPARK the view of package Standard differs from that described in Annex A.1 of the Ada *LRM* in that it does not include the following declarations:

1 the type Wide_Character;

2 the package ASCII;

3 the type Wide_String;

4 any predefined exceptions.

The following Identifiers are predefined in SPARK'S view of package Standard:

1 the types Integer and Long_Integer, and subtypes Natural and Positive;

2 the types Float and Long_Float;

3 the type Duration;

4 the type Boolean;

5 the type Character;

6 the type String.

The full definition of the predefined integer and floating point types that correspond to the target Ada compilation system, including the values of the implementation-defined constraints, (e.g. the range for type Integer) may be specified in a reduced version of package Standard via the target configuration file, for example:

```
package Standard is

   type Integer is range integer-range;
   type Short_Short_Integer is range integer-range;
   type Short_Integer is range integer-range;
   type Long_Integer is range integer-range;
   type Long_Long_Integer is range integer-range;

   type Short_Float is digits integer-value range real-range;
   type Float is digits integer-value range real-range;
   type Long_Float is digits integer-value range real-range;
   type Long_Long_Float is digits integer-value range real-range;

end Standard;
```

If the predefined types are not specified in this way, their constraints are undefined in SPARK.

Note that the existence of the Short_ and Long_ forms of Integer and Float is implementation dependent, and may not be supported by a particular compiler, so these types should only be used if specified in the SPARK definition of package Standard in the target configuration file.

The type Duration is declared as a fixed-point type, but values for its attributes such as 'First, 'Last, and 'Delta are not provided, since these are implementation defined and not specifiable via the target configuration file.

A.2 The Package Ada

The package Ada is predefined in SPARK as in Ada95.

A.3 Character Handling

A.3.1 The Package Characters

The package Ada.Characters is predefined in SPARK as in Ada95.

A.3.2 The Package Characters.Handling

The package Ada.Characters.Handling is not predefined in SPARK.

A.3.3 The Package Characters.Latin_1

The package Ada.Characters.Latin_1 is predefined in SPARK as in Ada95.

A.4 String Handling

The package Ada.Strings is not predefined in SPARK.

A.5 The Numerics Package

The package Ada.Numerics is not predefined in SPARK.

A.6 Input-Output

The SPARK language has no predefined packages for input-output, since the standard Ada input-output packages contain features not supported by SPARK. The Ada95 predefined input-output packages Ada.Sequential_IO, Ada.Direct_IO, Ada.Storage_IO, Ada.Text_IO, Ada.Wide_Text_IO, Ada.Streams.Stream_IO and Ada.IO_Exceptions are thus not predefined in SPARK.

However, the SPARK Examiner provides a package Spark_IO which defines operations for file manipulation and input-output of the predefined types Character, String, Integer and Float. If required, facilities for input-output of new integer and floating point types, fixed point types and enumeration types may be provided by the user, based on procedures in Spark_IO, whose specification and body are supplied in machine-readable form with the SPARK Examiner. For further details, consult the SPARK Examiner User Manual.

A.15 The Package Command_Line

The package Ada.Command_Line is not predefined in SPARK.

ANNEX B INTERFACE TO OTHER LANGUAGES

B.1 Interfacing Pragmas

A pragma Import may only occur in two places:

1 Immediately after a subprogram declaration (in a package specification or in a declarative part).

2 Immediately after a variable declaration.

In both cases, the entity named in the pragma must be the one whose declaration the pragma immediately follows.

B.2 The Package Interfaces

The package Interfaces is not predefined in SPARK, nor are any of its descendants.

ANNEX C SYSTEMS PROGRAMMING

C.1 Access to Machine Operations

Code insertions and calls to intrinsic subprograms are supported in SPARK (see section 6.3 for code insertions and annex B.1 for pragma Import).

C.2 Required Representation Support

Representation clauses may appear in SPARK texts. See section 13 for the restrictions in their usage.

C.3 Interrupt Support

C.3.2 The Package Interrupts

The package Ada.Interrupts is not predefined in SPARK.

C.7 Task Identification and Attributes

C.7.1 The Package Task_Identification

The package Ada.Task_Identification is not predefined in SPARK.

C.7.2 The Package Task_Attributes

The package Ada.Task_Attributes is not predefined in SPARK.

ANNEX D REAL-TIME SYSTEMS

As SPARK does not support tasking, this Annex is not part of the SPARK language.

ANNEX E DISTRIBUTED SYSTEMS

The features of this Annex are outside the scope of the SPARK language.

ANNEX F INFORMATION SYSTEMS

The features of this Annex are outside the scope of the SPARK language.

ANNEX G NUMERICS

The features of this Annex are outside the scope of the SPARK language.

ANNEX H SAFETY AND SECURITY

Although clearly of interest to SPARK users, the features of this Annex are outside the scope of the SPARK language itself.

ANNEX J OBSOLESCENT FEATURES

With the exception of package ASCII, the obsolescent features described in Annex J of the Ada *LRM* are not supported by SPARK.

In line with the Ada95 AARM A5.3 (72.f), obsolescent Ada 83 floating-point attributes are now allowed in SPARK 95 as implementation-defined attributes (see Annex K).

Annex K Language-defined Attributes

The following attributes are allowed in SPARK:

S'Adjacent
S'Aft
S'Base
S'Ceiling
X'Component_Size
S'Compose
S'Copy_Sign
S'Delta
S'Denorm
S'Digits
S'Exponent
A'First(N)
A'First (for array types)
S'First (for scalar types)
S'Floor
S'Fore
S'Fraction
A'Last(N)
A'Last (for array types)
S'Last (for scalar types)
S'Leading_Part
A'Length(N)
A'Length
S'Machine
S'Machine_Emax
S'Machine_Emin
S'Machine_Mantissa
S'Machine_Overflows
S'Machine_Radix
S'Machine_Rounds
S'Max
S'Min
S'Model
S'Model_Emin
S'Model_Epsilon
S'Model_Mantissa
S'Model_Small
S'Modulus
S'Pos
S'Pred
A'Range(N)
A'Range (for array types)
S'Range (for scalar types)
S'Remainder
S'Rounding
S'Safe_First
S'Safe_Last
S'Scaling
S'Signed_Zeros
S'Size (for subtypes)
X'Size (for objects)
S'Small (for fixed-point types)
S'Succ
S'Truncation
S'Unbiased_Rounding
S'Val
X'Valid

As noted in Annex J, the following attributes are allowed in SPARK 95 mode in accordance with the Ada95 AARM A5.3 (72.f):

S'Emax

S'Epsilon

S'Large

S'Mantissa

S'Safe_Emax

S'Safe_Large

S'Safe_Small

S'Small (for floating-point types)

ANNEX L LANGUAGE-DEFINED PRAGMAS

Except for pragmas Elaborate_Body (see Section 10.2.1), and Import (see Annex B.1), the SPARK Examiner issues warning messages when it encounters pragmas, but otherwise it ignores them.

ANNEX M TOOL-DEPENDENT FEATURES

This annex describes features that are not part of the core SPARK language but are associated with SPARK language tools. Further details may be found in the relevant tool user manuals.

M.1 The hide directive

The SPARK Examiner supports a feature which permits certain parts of a program text to be hidden from it. The Examiner reports that the text has been hidden, but otherwise ignores any text in the hidden part.

The parts of a program text that may be hidden are:

1 A subprogram implementation (see Section 6.3). This permits program development by successive refinement.

2 The exception handler part of a subprogram implementation. In this case the hide directive must immediately follow the reserved word **exception**.

3 The private part of a package specification. This makes it possible to implement abstract data types in terms of concrete types not supported by SPARK, such as access types.

4 A package implementation (see Section 7.1).

5 A package initialization (see Section 7.1).

Hidden text is introduced by a hide directive, which takes the form

--# hide *program_unit*_name

The program unit name is the name of the subprogram or package whose details are to be hidden. All text after the hide directive, up to an **end** immediately followed by the same program unit name, is ignored by the SPARK Examiner.

M.2 Additional reserved words

In addition to those listed in Section 2.9, the identifiers below are reserved for use by the SPARK proof tools. The use of these identifiers in a SPARK program must be avoided if generation of verification conditions (including those for the absence of run-time errors) is required.

are_interchangeable	finish	may_be_deduced	requires
as	first	may_be_deduced_from	
assume	for_all	may_be_replaced_by	
	for_some		save
			sequence
		nonfirst	set
		nonlast	sqr
	goal	not_in	start
			strict_subset_of
const			subset_of
		odd	succ

div		pending	update
		pred	
		proof	var
element	last		where

Also in this category are all identifiers which start with the characters "fld_" or "upf_".

M.3 Extensions to annotations

Other annotations besides those in the core language (see Section 2.11) may be accepted by the SPARK Examiner in order to support related SPARK language tools. Similarly, extensions to the form of the core language annotations may be supported by the Examiner.

III Collected Syntax of SPARK

Rules marked with an asterisk (*) are variants of rules of standard Ada and those marked with a plus (+) are additional rules.

2.1

character ::= graphic_character | format_effector | other_control_function
graphic_character ::= identifier_letter | digit | space_character | special_character

2.3

identifier ::= identifier_letter { [underline] letter_or_digit }
letter_or_digit ::= identifier_letter | digit

2.4

numeric_literal ::= decimal_literal | based_literal

2.4.1

decimal_literal ::= numeral [.numeral] [exponent]
numeral ::= digit { [underline] digit }
exponent ::= **E** [+] numeral | **E** - numeral

2.4.2

* based_literal ::= base # based_numeral # [exponent]
base ::= numeral
based_numeral ::= extended_digit { [underline] extended_digit }
extended_digit ::= digit | A | B | C | D | E | F

2.5

character_literal ::= 'graphic_character'

2.6

string_literal ::= "{ string_element }"
string_element ::= "" | *non_quotation_mark*_graphic_character

2.7

comment ::= --{ *non_end_of_line* _character }

2.8

pragma ::=
pragma identifier [(pragma_argument_association
{ , pragma_argument_association })] ;
pragma_argument_association ::=
[*pragma_argument_*identifier **=>**] name
| [*pragma_argument_*identifier **=>**] expression

3.1

* basic_declaration ::=
type_declaration | subtype_declaration
| object_declaration | number_declaration
defining_identifier ::= identifier

3.2.1

* type_declaration ::=
full_type_declaration |
private_type_declaration |
private_extension_declaration
* full_type_declaration ::= **type** *defining_*identifier **is** type_definition ;

* type_definition ::=
enumeration_type_definition | integer_type_definition
| real_type_definition | array_type_definition
| record_type_definition | modular_type_definition
| record_type_extension

\+ record_type_extension ::= **new** type_mark **with** record_definition ;

3.2.2

subtype_declaration ::=
subtype defining_identifier **is** subtype_indication ;
subtype_indication ::= subtype_mark [constraint]
subtype_mark ::= *subtype_*name
constraint ::= scalar_constraint | composite_constraint
* scalar_constraint ::= range_constraint
* composite_constraint ::= index_constraint

3.3.1

* object_declaration ::=
defining_identifier_list : [**constant**] subtype_mark [:= expression] ;
defining_identifier_list ::= defining_identifier { , defining_identifier }

3.3.2

number_declaration ::= defining_identifier_list : **constant** := *static_*expression ;

3.4

*

3.5

* range_constraint ::= **range** *static*_range
range ::= range_attribute_reference
| simple_expression .. simple_expression

3.5.1

enumeration_type_definition ::=
(enumeration_literal_specification { , enumeration_literal_specification })
* enumeration_literal_specification ::= defining_identifier

3.5.4

* integer_type_definition ::= signed_integer_type_definition
signed_integer_type_definition ::=
range *static*_simple_expression .. *static*_simple_expression
modular_type_definition ::= **mod** *static*_simple_expression

3.5.6

real_type_definition ::=
floating_point_definition | fixed_point_definition

3.5.7

* floating_point_definition ::=
digits *static*_simple_expression [real_range_specification]
real_range_specification::=
range *static*_simple_expression .. *static*_simple_expression

3.5.9

* fixed_point_definition ::= ordinary_fixed_point_definition
* ordinary_fixed_point_definition ::=
delta *static*_simple_expression real_range_specification
*

3.6

array_type_definition ::=
unconstrained_array_definition | constrained_array_definition
unconstrained_array_definition ::=
array (index_subtype_definition { , index_subtype_definition }) **of**
component_definition
index_subtype_definition ::= subtype_mark **range** <>
constrained_array_definition ::=
array (discrete_subtype_definition { , discrete_subtype_definition }) **of**
component_definition
* discrete_subtype_definition ::= *discrete*_subtype_mark
* component_definition ::= subtype_mark

3.6.1

* index_constraint ::= (*discrete*_subtype_mark { , *discrete*_subtype_mark })
* discrete_range ::= *discrete*_subtype_indication | *static*_range

3.7
*

3.7.1
*

3.8
* record_type_definition ::= [**tagged**] record_definition
* record_definition ::=
record
component_list
end record | **null record**
* component_list ::= component_item { component_item } | **null**
* component_item ::= component_declaration
* component_declaration ::=
defining_identifier_list : component_definition ;

3.8.1
*
discrete_choice_list ::= discrete_choice { | discrete_choice }
* discrete_choice ::= *static*_simple_expression | discrete_range

3.9.1
*

3.10
*

3.10.1
*

3.11
* declarative_part ::=
{ renaming_declaration }
{ declarative_item | embedded_package_declaration
| external_subprogram_declaration }
* declarative_item ::= basic_declarative_item | body | generic_function_instatiation
* basic_declarative_item ::= basic_declaration | representation_clause
\+ embedded_package_declaration ::=
package_declaration
{ renaming_declaration | use_type_clause }
\+ external_subprogram_declaration ::=
subprogram_declaration
pragma Import (pragma_argument_association, pragma_argument_association
{ , pragma_argument_association });
body ::= proper_body | body_stub
* proper_body ::= subprogram_body | package_body

4.1

* name ::= direct_name
| indexed_component | selected_component
| attribute_reference | function_call
* direct_name ::= identifier
* prefix ::= name
*

4.1.1

indexed_component ::= prefix (expression { , expression })

4.1.2

*

4.1.3

selected_component ::= prefix . selector_name
* selector_name ::= identifier

4.1.4

attribute_reference ::= prefix'attribute_designator
* attribute_designator ::= identifier [(expression [, expression])] | **Delta** | **Digits**
range_attribute_reference ::= prefix'range_attribute_designator
range_attribute_designator ::= **Range** [(*static*_expression)]

4.3

4.3.1

* record_ aggregate ::= positional_record_aggregate | named_record_aggregate
\+ positional_record_aggregate ::= (expression { , expression })
\+ named_record_aggregate ::=
(record_component_association { , record_component_association })
* record_component_association ::= *component*_selector_name **=>** expression
*

4.3.2

* extension_aggregate ::= (ancestor_part **with** record_component_association_list) |
(ancestor_part **with null record**)
* ancestor_part ::= expression
\+ record_component_association_list ::=
named_record_component_association |
positional_record_component_association
\+ positional_record_component_association ::= expression { , expression }
\+ named_record_component_association ::=
record_component_association { , record_component_association }

4.3.3

array_aggregate ::= positional_array_aggregate | named_array_aggregate
* positional_array_aggregate ::=
(aggregate_item , aggregate_item { , aggregate_item })
| (aggregate_item { , aggregate_item } , **others =>** aggregate_item)
* named_array_aggregate ::=
(array_component_association { , array_component_association }
[, **others =>** aggregate_item])
| (**others =>** aggregate_item)
* array_component_association ::= discrete_choice_list **=>** aggregate_item
\+ aggregate_item ::= expression | array_aggregate

4.4

expression ::=
relation { **and** relation } | relation { **and then** relation }
| relation { **or** relation } | relation { **or else** relation }
| relation { **xor** relation }
relation ::=
simple_expression [relational_operator simple_expression]
| simple_expression [**not**] **in** range
| simple_expression [**not**] **in** subtype_mark
simple_expression ::=
[unary_adding_operator] term { binary_adding_operator term }
term ::= factor { multiplying_operator factor }
factor ::= primary [** primary] | **abs** primary | **not** primary
* primary ::=
numeric_literal | character_literal | string_literal
| name | type_conversion
| qualified_expression | (expression)

4.5

*
relation_operator ::= **=** | **/=** | **<** | **<=** | **>** | **>=**
binary_adding_operator ::= **+** | **-** | **&**
unary_adding_operator ::= **+** | **-**
multiplying_operator ::= ***** | **/** | **mod** | **rem**
*

4.6

* type_conversion ::= subtype_mark (expression)

4.7

qualified_expression ::=
subtype_mark'(expression) | subtype_mark'aggregate

4.8
*

5.1

sequence_of_statements ::= statement { statement }
* statement ::=
simple_statement | compound_statement
* simple_statement ::= null_statement
| assignment_statement | procedure_call_statement
| exit_statement | return_statement

* compound_statement ::=
if_statement | case_statement
| loop_statement
null_statement ::= **null**;
*
statement_identifier ::= direct_name

5.2

assignment_statement ::=
*variable*_name := expression;

5.3

if_statement ::=
if condition **then**
sequence_of_statements
{ **elsif** condition **then**
sequence_of_statements }
[**else**
sequence_of_statements]
end if;
condition ::= *boolean*_expression

5.4
* case_statement ::=
case expression **is**
case_statement_alternative
{ case_statement_alternative }
[**when others =>** sequence_of_statements]
end case;
case_statement_alternative ::=
when discrete_choice_list **=>** sequence_of_statements

5.5

loop_statement ::=
[*loop*_statement_identifier :]
[iteration_scheme]
loop
sequence_of_statements
end loop [*loop*_identifier] ;
iteration_scheme ::= **while** condition | **for** loop_parameter_specification
* loop_parameter_specification ::=
defining_identifier **in** [**reverse**] *discrete*_subtype_mark [**range** range]

5.6
*

5.7
* exit_statement ::= **exit** [*simple*_name] [**when** condition] ;

5.8
*

6.1
* subprogram_declaration ::=
procedure_specification ; procedure_annotation
| function_specification ; function_annotation
*
\+ procedure_specification ::=
procedure defining_identifier parameter_profile
\+ function_specification ::=
function defining_designator parameter_and_result_profile
* designator ::= identifier
* defining_designator ::= defining_identifier
defining_program_unit_name ::= [parent_unit_name .] defining_identifier
operator_symbol ::= string_literal
defining_operator_symbol ::= operator_symbol
parameter_profile ::= [formal_part]
parameter_and_result_profile::= [formal_part] **return** subtype_mark
formal_part ::=
(parameter_specification { ; parameter_specification })
* parameter_specification ::=
defining_identifier_list : mode subtype_mark
mode ::= [**in**] | **in out** | **out**

6.1.1

```
+   procedure_annotation ::=
        [ global_definition ]
        [ dependency_relation ]

+   function_annotation ::=
        [ global_definition ]
```

6.1.2

```
+   global_definition ::=
        --# global global_mode global_variable_list ; { global_mode global_variable_list ; }
+   global_mode ::= in | in out | out
+   global_variable_list ::= global_variable { , global_variable }
+   global_variable ::= entire_variable
+   entire_variable ::= [ package_name . ]  direct_name
+   dependency_relation ::=
      --# derives [dependency_clause {  & dependency_clause } [& null_dependency_clause]] ;
    |  --# derives null_dependency_clause ;
+   dependency_clause ::=
        exported_variable_list from [ imported_variable_list ]
+   exported_variable_list ::= exported_variable { , exported_variable }
+   exported_variable ::= entire_variable
+   imported_variable_list ::= *  |  [ * , ] imported_variable { , imported_variable }
+   imported_variable ::= entire_variable
+   null_dependency_clause ::= null from imported_variable { , imported_variable }
```

6.3

```
*   subprogram_body ::=
          procedure_specification
        [ procedure_annotation ]
          is
          subprogram_implementation
      |   function_specification
        [ function_annotation ]
          is
          subprogram_implementation
+   subprogram_implementation ::=
            declarative_part
          begin
            sequence_of_statements
          end designator ;
      |   begin
            code_insertion
          end designator ;
+   code_insertion ::= code_statement { code_statement }
```

6.4

* procedure_call_statement ::=
*procedure*_name [actual_parameter_part] ;
* function_call ::=
*function*_name [actual_parameter_part]
* actual_parameter_part ::= (parameter_association_list)
*
\+ parameter_association_list ::=
named_parameter_association_list | positional_parameter_association_list
\+ named_parameter_association_list ::=
*formal_parameter*_selector_name **=>** explicit_actual_parameter
{ , *formal_parameter*_selector_name **=>** explicit_actual_parameter }
\+ positional_parameter_association_list ::=
explicit_actual_parameter { , explicit_actual_parameter }
explicit_actual_parameter ::= expression | *variable*_name

6.5

* return_statement ::= **return** expression ;

7.1

* package_declaration ::= [inherit_clause] package_specification ;
\+ private_package_declaration ::=
[inherit_clause] **private** package_specification ;
* package_specification ::=
package defining_program_unit_name
package_annotation
is
visible_part
[**private**
private_part]
end [parent_unit_name .] identifier
\+ visible_part ::=
{ renaming_declaration }
{ package_declarative_item }
\+ private_part ::=
{ renaming_declaration }
{ package_declarative_item }
\+ package_declarative_item ::=
basic_declarative_item | subprogram_declaration
| external_subprogram_declaration

7.1.1

\+ inherit_clause ::=
--# inherit *package*_name { , *package*_name } ;

7.1.2

\+ package_annotation ::=
[own_variable_clause [initialization_specification]]

7.1.3

\+ own_variable_clause ::= **--# own** own_variable_list ;
\+ own_variable_list ::= mode own_variable { , mode own_variable }
\+ own_variable ::= direct_name

7.1.4

\+ initialization_specification ::=
--# initializes own_variable_list ;

7.2

* package_body ::=
package body defining_program_unit_name
[refinement_definition]
is
package_implementation
end [parent_unit_name .] identifier ;
\+ package_implementation ::=
declarative_part
[**begin**
package_initialization]

\+ package_initialization ::=
sequence_of_statements

7.2.1

\+ refinement_definition ::=
--# own refinement_clause { **&** refinement_clause } ;
\+ refinement_clause ::=
subject **is** constituent_list
\+ subject ::= direct_name
\+ constituent_list ::= mode constituent { , mode constituent }
\+ constituent ::= [*package*_name .] direct_name

7.3

* private_type_declaration ::=
type defining_identifier **is [tagged] [limited] private**;
* private_extension_declaration ::=
type defining_identifier **is new** *ancestor*_subtype_indication **with private** ;

8.4
*

use_type_clause ::= **use type** subtype_mark { , subtype_mark } ;

8.5
* renaming_declaration ::=
package_renaming_declaration
| subprogram_renaming_declaration

8.5.1 - 8.5.2
*

8.5.3
* package_renaming_declaration ::=
package defining_program_unit_name
renames parent_unit_name . *package_*direct_name ;

8.5.4
* subprogram_renaming_declaration ::=
function defining_operator_symbol formal_part **return** subtype_mark
renames *package_* name . operator_symbol ;
| function_specification
renames *package_* name . *function_*direct_name ;
| procedure_specification
renames *package_* name . *procedure_*direct_name ;

8.5.5
*

9.1
*

10.1.1
compilation ::= { compilation_unit }
compilation_unit ::=
context_clause library_item | context_clause subunit
* library_item ::= library_unit_declaration | library_unit_body
* library_unit_declaration ::=
package_declaration | private_package_declaration | main_program
*
* library_unit_body ::= package_body
parent_unit_name ::= name
+ main_program ::=
[inherit_clause]
main_program_annotation
subprogram_body
+ main_program_annotation ::=
--# main_program ;

10.1.2

context_clause ::= { context_item }
* context_item ::= with_clause | use_type_clause
* with_clause ::= **with** *library_package_*name { , *library_package_*name } ;

10.1.3
* body_stub ::= subprogram_body_stub | package_body_stub
* subprogram_body_stub ::=
procedure_specification [procedure_annotation] **is separate**;
| function_specification [function_annotation] **is separate** ;
* package_body_stub ::=
package body defining_identifier **is separate**;
subunit ::= **separate** (parent_unit_name) proper_body

11.1 - 12.2
*

12.3
* generic_instantiation ::= **function** defining_designator **is**
new *generic_function_*name [generic_actual_part]
generic_actual_part ::= (generic_association {, generic_association })
generic_association ::= [*generic_formal_parameter_*selector_name =>]
explicit_generic_actual_parameter
explicit_generic_actual_parameter ::= subtype_mark

12.4 - 7
*

13.1
* representation_clause ::=
attribute_definition_clause
| enumeration_representation_clause
| record_representation_clause
| at_clause
local_name ::= direct_name
| direct_name'attribute_designator
| *library_unit_*name

13.3
* attribute_definition_clause ::= **for** local_name'attribute_designator **use** simple_expression;

13.4
enumeration_representation_clause ::=
for *first_subtype_*local_name **use** enumeration_aggregate ;
enumeration_aggregate ::= array_aggregate ;

13.5

at_clause ::= **for** *simple*_name **use at** *simple*_expression ;

13.5.1

* record_representation_clause ::=
for *first_subtype*_local_name **use**
record [mod_clause]
{ component_clause }
end record;
component_clause ::=
*component*_local_name **at** position **range** first_bit .. last_bit ;
* position ::= *static*_simple_expression
first_bit ::= *static*_simple_expression
last_bit ::= *static*_simple_expression
mod_clause ::= **at mod** *simple*_expression ;

13.8

code_statement ::= qualified_expression ;

REFERENCES

[BARNES, 2003] BARNES, J. (2003) High Integrity Software, The SPARK Approach to Safety and Security, Addison-Wesley.

[BERGERETTI and CARRÉ, 1985] BERGERETTI J.-F. and CARRÉ B.A. (1985). "Information-flow and data-flow analysis of while-programs". ACM Trans. on Prog. Lang. and Syst. 7, 37-61.

[BJORNER and OEST, 1980] BJORNER D. and OEST O.N. (1980). Towards a Formal Description of Ada. LNCS-98, Springer-Verlag.

[BOOCH, 1983] BOOCH, G. (1983) Software Engineering with Ada, Benjamin Cummings.

[BUNDGAARD and SCHULTZ, 1980] BUNDGAARD J. and SCHULTZ L. (1980). "A denotational (static) semantics method for defining Ada context conditions." In BJORNER D. and OEST O.N. (Editors), Towards a Formal Description of Ada. LNCS-98, Springer-Verlag, 21-212.

[CARRÉ and DEBNEY, 1985] CARRÉ B.A. and DEBNEY C.W. (1985). SPADE-Pascal Manual. Program Validation Limited.

[CRAIGEN, 1987] CRAIGEN D. (1987). A Description of m-Verdi. I.P.Sharp Technical Report TR-87-5420-02.

[CULLYER and GOODENOUGH, 1987] CULLYER W.J. and GOODENOUGH S.J. (1987). The choice of computer languages for use in safety-critical systems. RSRE Memorandum 3946.

[CURRIE, 1984] CURRIE I.F. (1984). Orwellian programming in safety-critical systems. RSRE Memorandum 3924.

[De MILLO et al, 1979] De MILLO R.A., LIPTON R.J. and PERLIS A.J. (1979). "Social processes and proofs of theorems and programs." Comm. ACM 22, 271-280.

[GERMAN, 1978] GERMAN S.M. (1978). "Automating proofs of the absence of common runtime errors." Proc. 5th ACM Conference on Principles of Programming Languages, ACM, New York.

[GOODENOUGH, 1987] GOODENOUGH S.J. (1987). "Technical comparisons, Ada and Pascal." Annex to CULLYER W.J. and GOODENOUGH S.J.: The choice of computer languages for use in safety-critical systems, RSRE Memorandum 3946.

[HAYES, 1987] HAYES I. (1987). Specification Case Studies. Prentice-Hall International.

[HOARE, 1981] HOARE C.A.R. (1981). "The emperor's old clothes." Comm. ACM 24, 75-83.

[HOLZAPFEL and WINTERSTEIN, 1987] HOLZAPFEL R. and WINTERSTEIN G. (1987). Safe Ada (Version 1.1). SYSTEAM KG.

[LEDGARD and SINGER, 1982] LEDGARD F.L. and SINGER A. (1982). "Scalin g down Ada (or towards a standard Ada subset)." Comm. ACM 25, 121-125.

[LUCKHAM et al, 1987] LUCKHAM D.C., von HENKE F.W., KRIEG-BRUECKNER B. and OWE O. (1987). ANNA - A language for annotating Ada programs. LNCS-260, Springer-Verlag.

[McGETTRICK, 1982] McGETTRICK A. (1982). Program verification using Ada. Cambridge University Press.

[MENDAL, 1988] MENDAL G.0. (1988). "Three reasons to avoid the use clause". Ada Letters.

[PEDERSON, 1980] PEDERSON J.S. (1980). "A formal semantics definition of sequential Ada." In BJORNER D. and OEST O.N. (Editors), Towards a Formal Description of Ada. LNCS-98, Springer-Verlag, 213-308.

[PROGRAM VALIDATION LTD, 1994] PROGRAM VALIDATION LTD (1994). "Formal Semantics of SPARK"

[SAALTINK, 1987] SAALTINK, M. (1987). The mathematics of m-Verdi. I.P.Sharp Technical Report TR-87-5420-03.

END OF DOCUMENT