SPARK Examiner
User Manual – Release 7.3 [NT]

EXM/UM
Issue: 7.3
Status: Definitive
12 January 2006

Originator

SPARK Team

Approver

SPARK Team Line Manager

Contents

1 Overview

SPARK is an annotated sublanguage of Ada, intended for high-integrity programming. The language exists in variants based on both Ada 83 and Ada 95. The SPARK language is described in a separate reports - *"SPARK - the SPADE Ada Kernel"* and *"SPARK 95 - the SPADE Ada 95 Kernel"* - which form part of the Examiner manual set; these reports will be referred to here as the *SPARK Definition.* The Examiner is operated in the same way on both SPARK 83 and SPARK 95 source code, although the rules which it implements differ according to the language variant selected.

As well as imposing language restrictions, to eliminate ambiguities and insecurities which exist in the full Ada language, SPARK incorporates *annotations*, or formal comments. These are of two kinds:

Core annotations whose use is imposed by certain language rules of SPARK (The SPARK Definition contains a list of these), and

Proof contexts which may be employed to include formal specifications in a SPARK text, and to guide its machine-assisted proof.

The SPARK Examiner is a free-standing software tool to support the use of this language. The Examiner always:

- checks conformance of a text with the rules of SPARK.
- checks consistency of executable code with its core annotations.

and optionally can be used to:

- obtain *verification conditions (VCs)* for SPARK programs to which the SPADE Automatic Simplifier (SAS) may be applied. The SAS reduces a substantial proportion of the provable conclusions in verification conditions to "true". Any VC conclusions remaining to be verified, after application of the SAS, can be read by the SPADE Proof Checker, which can then be used to prove them interactively. The use of the SPADE Automatic Simplifier and Proof Checker is described separately, in other Praxis High Integrity Systems documents. This manual explains how to use the Examiner to generate verification conditions for SPARK programs. The process of proof is described in the manual *"Generation of Verification Conditions for SPARK Programs"*.
- facilitate demonstration that a SPARK program is free from run-time errors. This process is described in the manual *"Generation of Run-time Checks for SPARK Programs"*.
- produce an obsolescent and less-rigorous form of analysis called *path functions*. Its use for this purpose is described in the manual *"Generation of Path Functions for SPARK Programs"*.

Advice on the contents of this document may be obtained:

by phone:	+44 (0)1225 823829 (direct line) or +44 (0)1225 466991 (exchange)
by FAX:	+44 (0)1225 469006
by Email:	sparkinfo@praxis-his.com

2 General description

The Examiner supports several levels of analysis. The Examiner supports:

- checking of SPARK language syntactic and static semantic rules
- data flow analysis
- data and information flow analysis
- formal program verification via generation of verification conditions
- proof of absence of run-time errors

There is also an option to make the Examiner perform syntax checks only. Using this option on a source file does not require access to any other units on which the file depends, so files can be syntax checked on an individual basis. This allows any syntax errors to be corrected before the file is included in a complex examination. *This option must only be used as a pre-processor: the absence of syntax errors does NOT indicate that the source text is a legal SPARK program.*

2.1 Examiner Operation

The Examiner will analyse the contents of all the source files named in the Examiner command line. Each file may contain any number of SPARK compilation units. Within a compilation unit, the body of any constituent subprogram or package may be fully implemented, or “hidden” (see SPARK Definition) or replaced by a stub, as appropriate.

For the Examiner to analyse a particular compilation unit, it may require access to some other compilation units. Specifically:

1 to analyse a package body, the Examiner requires access to its specification;

2 to analyse a package specification or main program which inherits other packages, the Examiner requires access to the specifications of all those inherited packages (and to the specifications of all packages which they themselves inherit, and so on);

3 to analyse a sub-unit, the Examiner requires access to its parent.

The two ways in which access to these units can be provided are described below.

Firstly, it is possible to specify all the files to be employed in an analysis in an Examiner command line (either directly, or indirectly by using meta files (see section 4.4)). In this case, the Examiner will analyse all the contents of those files, as it reads them, in the order in which their names occur in the command line. This order (and the order of units within each of the files) must be such that the Examiner only requires access to units which it has already analysed.

Alternatively, it is also possible to provide an index file (see section 4.2), which associates SPARK compilation units with the files which contain them. If an index file is supplied, the Examiner can use it to find compilation units it requires, but whose locations (file-names) have not been specified in the command line.

The index file is a text file that can be generated either directly by the user or by a user-supplied program. It could, for example, be produced by the “build” mechanism of a configuration-management tool. Facilities are provided for “inheriting” other index information (through a super-index), and for subdividing index information into “separate” files (using “auxiliary indexes”).

The Examiner checks conformance of the contents of the specified files to the rules of SPARK, performing all the tests that the visibility of the text allows. This analysis is performed in several passes. First lexical and syntax analysis of a compilation unit is carried out, allowing the Examiner to identify what other units are required to carry out the complete analysis of this compilation unit. The Examiner then reads all other required units, if they have not already been read. When all the required units of a compilation unit have been read the Examiner performs static-semantic analysis, and if this is successful, it goes on to perform data- and information-flow analyses.

2.1.1 Output files

The Examiner produces a listing file for each source file specified on the command line. This listing file has the same name as the source file but with the default file type .lst. It contains messages indicating the success of the analysis of each subprogram, or if an analysis is unsuccessful, indications of the errors found.

A single analysis report file is also produced. This lists the source files read, the compilation units contained within those source files, and all errors found during the analysis.

The report file and each listing file are terminated with a line reading

```
--End of file-----------------------------------------------------
```

A similar line terminates the screen echo.

The Examiner can optionally produce report and listing files in HTML allowing them to be “browsed” using any Web Browser - see Section 4.8.

2.2 Verification Conditions and Run-time Checks

If any of the Verification Condition (VC) options have been selected by the user, the Examiner will, after performing the analyses described in the previous section, produce VCs for each well-formed subprogram in the files specified for analysis.

It is recommended that VC generation is not attempted until the source text is free of SPARK semantic errors; this is because their generation takes additional time and the contents of the output files produced are unlikely to be valid unless the source is legal SPARK. It is *not* necessary to eliminate all flow errors from code before generating VCs (such errors should, of course, be considered for significance and acceptability).

2.2.1 Output files

When VC generation is selected, for each subprogram in the files specified on the Examiner command line the following additional output files are generated. They contain the FDL type declarations needed if the SAS or Proof Checker are to be used, the results of the selected analysis and associated proof rules. The file extensions of these output files are as follows:

File	Extension
FDL Declarations	.fdl
Analysis	.vcg
Rules	.rls

The naming of these output files is covered in more detail in section 4.7.

2.2.2 Relationship with the SPADE tools

The verification condition (VC) files produced by the Examiner can be manipulated by the SPADE tools. VCs can be simplified by the SPADE Automatic Simplifier and the proof of VCs too complex for automatic proof by the Simplifier can be attempted using the SPADE Proof Checker. The POGS tool provides a summary of VCs indicating their source and current proof status. Fig. 1 overleaf gives an overview of the relationship between these four tools and the input and output files used to connect them.

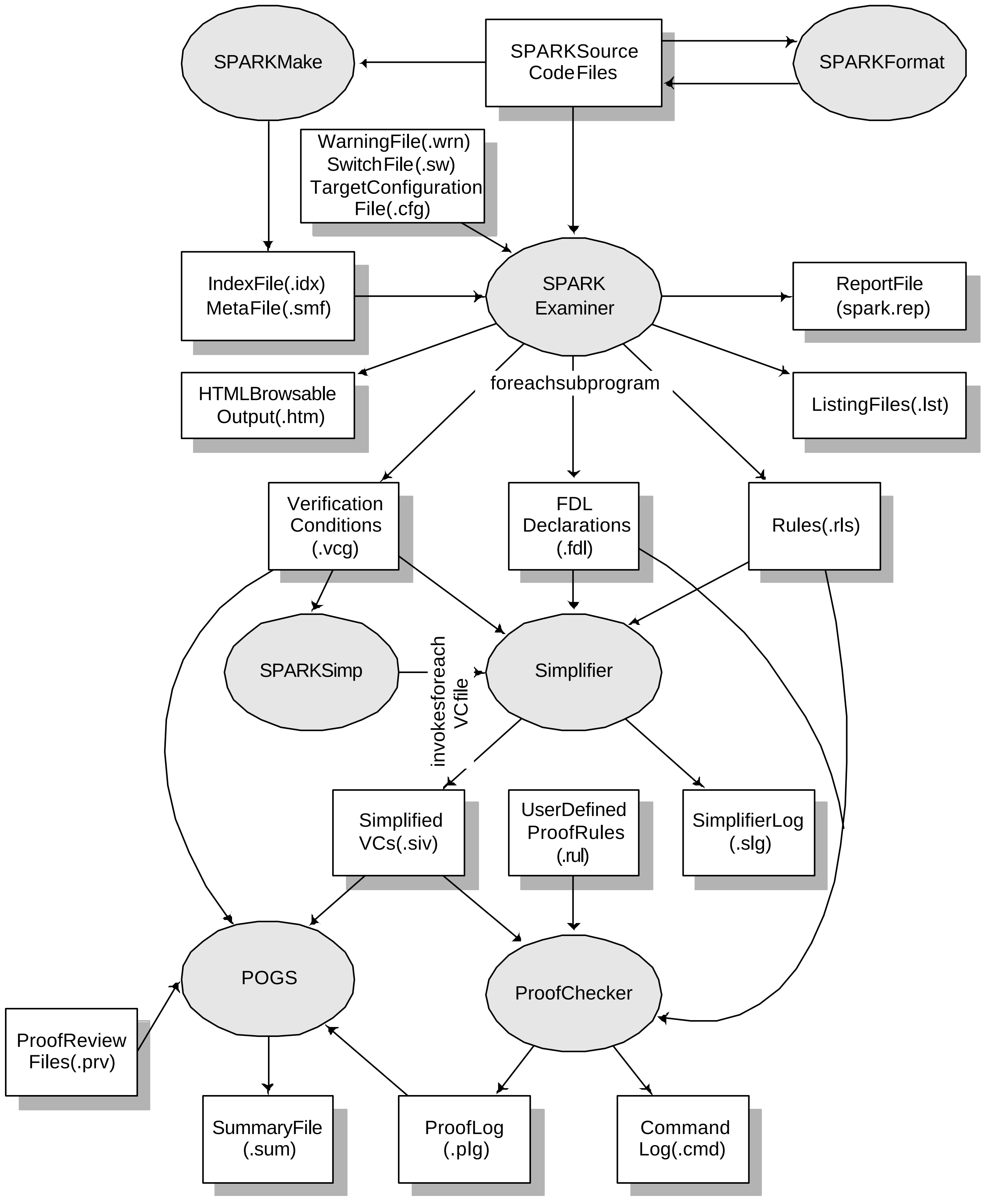


Figure 1 Relationship of SPARK and SPADE Tools

3 Operating the Examiner

The Examiner is a Windows NT console application which is command line driven. The command line syntax is given below.

```
Command-line = "SPARK" { Command-option } Argument-list
Argument-list = Argument { , Argument }
Argument = ( File-spec [ Argument-option ] ) | Meta-file-spec
Argument-option = ( " /listing_file=" file-spec | " /nolisting_file" )
Meta-file-spec = "@"File-spec
```

Argument-option gives the name of the listing file for that argument. If an extension is not included in the supplied file-spec, the default listing extension is used. The option /nolisting_file" suppresses the generation of a listing file for the source file to which the option is applied (global suppression of listing files can be achieved by specifying the /nolistings command option see section 3.1.2). The results are always summarised in the report file. The default listing-file name is the name of the source file, with the source-file extension replaced by the listing-file extension.

The file-specs of the argument-list specify those files whose contents are to be analysed. All units found in files specified on the command line will be analysed. If an extension is not included in a given file-spec then the default source extension will be used (see below). A Meta-file-spec is the name of a file containing a list of files to be analysed: this option is described further in section 4.4.

A file-spec may contain space characters; however, where spaces are used the entire file-spec must be enclosed in double quotes.

The Examiner may also be invoked without command line arguments in which case it displays help information giving the options available.

The Command-options are given in the following table. Note that the option names may be abbreviated: the minimum abbreviation is that given in the column specifying the command syntax.

Note on Rational Apex source file naming convention. The Rational Apex compiler uses a file naming convention that requires a little extra care when used with the SPARK Examiner. Source file names take the form `p.1.ada` for package specifications and `p.2.ada` for package bodies. The following steps should be taken to handle these filenames:

(1) specify the complete name of each source file including the `.ada` extension. (The Examiner cannot distinguish between `p.1` where the intended file extension is `1` or `p.1` where the intention is to use the default file extension giving a source name `p.1.ada`); and

(2) use the `listing_file` argument- to ensure that a suitable listing file names are generated (otherwise the Examiner will use generate the rather unintuitive listing file names `p.1` and `p.2`)

The use of command line metafiles described in Section 4.4 may be convenient for dealing with these cases.

3.1 Command Line Switch Summary

3.1.1 Input File Options

Option	Syntax	Default	Description
source_extension	/so=*file-type*	*.ada*	Allows the user to specify a default extension for source files
index_file	/i=*file-spec*		Identifies the index file to be used initially. The index file mechanism is described in section 4.2
noindex_file	/noi	v	Suppresses the index file mechanism
warning_file	/w=*file-spec*		This identifies the warning control file to be used. A warning control file is used to determine how certain warning messages generated by the Examiner are displayed. The format and content of a warning control file is described in section 4.3.
nowarning_file	/now	v	Provides full reporting of all warnings
target_compiler_data	/ta=*file_spec*		Specifies the name of the target data file: this file is described in section 4.3
notarget_compiler_data	/not	v	Suppresses the use of the target data file
config_file	/conf=*file_spec*		Specifies the name of the target configuration file: this file is described in section 4.6
noconfig_file	/noc	v	Suppresses the use of the target configuration file

3.1.2 Output File Options

Option	Syntax	Default	Description
listing_extension	/l=*file-type*	.lst	Allows the user to specify a default extension for listing files. The "_" character may be used as a "wild card" in the listing extension specified by this switch. This character has the effect of preserving a character from the original source file extension. For example, given source files `p.ads` and `p.adb` with `/listing_extension=ls_` we would generate listing files `p.lss` and `p.lsb`
report_file	/rep=*file-spec*	*spark.rep*	This specifies the report file name. The format of the report file is described in section 4.1. The default extension *.rep* is applied if no extension is given.
noreport_file	/nor		Suppresses the production of the report file
html	/ht[=dirname]		Enables generation of HTML output files in addition to plain text output. "HTML" is the default directory name if none is given. This option cannot be used with the /noreport_file option.
tree	/tr	NT and Sun	Causes VC files to be produced in the directory tree structure described in section 4.7.1.
concatenate	/conc	VAX	Causes VC files to be produced in the flat file structure described in section 4.7.2.
plain_output	/pl		Causes report and listing files to be produced without line numbers, error numbers, cross-references in output, or dates. This can be useful when using "diff" utilities to compare analysis results because it leaves only the significant changes in the comparison output.
brief	/b		Causes on-screen errors and warnings to be issued in a "brief" format comprising <filename>:<line>:<column>: <message> This format is designed to allow integration with other development environments and tools.
nolistings	/nol		Suppresses the generation of all listing files (to suppress individual listing files see section 3). The results are always summarised in the

			report file.

3.1.3 Analysis Options

Option	Syntax	Default	Description
syntax_check	/sy		Given this switch, the Examiner will only check that the source files are in the SPARK syntax. *Note that this does NOT check whether a text is legal SPARK, and must only be used as a pre-processor*
ada83	/ad		Forces the Examiner to operate according to the rules of SPARK 83 instead of the default SPARK 95.
flow_analysis	/fl=*type*	information	The types are *'information'* and *'data'* and may be abbreviated to 'i' and 'd' respectively. Section Error! Reference source not found. describes the different types of flow analysis. Note that the data flow analysis option is not enabled by default when the Examiner is being used in SPARK 83 mode. Please contact Praxis High Integrity Systems if you wish to make use of this feature on SPARK 83 programs.
vcs	/vc		This option is no longer supported for further assistance please contact Praxis High Integrity Systems.
rtc	/rt		Causes the Examiner to produce VCs to show that each well-formed subprogram meets its postcondition and VCs associated with the Ada checks *index check*, *range check* and *division check.*
realrtcs	/rea		Causes the Examiner to produce run-time checks for real numbers (float and fixed). Note that this option is not enabled by default. Please contact Praxis High Integrity Systems if you wish to make use of this feature.
rules	/ru=*policy*	none	The policies are *'none'*, *'lazy'*, *'keen'* and *'all'* and may be abbreviated to *'n', 'l', 'k'* and *'a'* respectively. This option defines the policy applied by the Examiner to replacement rule generation for composite constants.
noduration	/nodu		Causes the Examiner to not know anything about the predefined type Standard.Duration. This allow programs to use "duration" as an identifier.

exp_checks	/e		Causes the Examiner to produce all the VCs produced by the /rtc switch and VCs associated with the Ada *overflow check*.
pfs	/pf		Causes the Examiner to generate an obsolescent form of semantic analysis output called "path functions". Path function generation is described in the manual "Generation of path functions for SPARK programs"
profile	/pr= *type*	sequential	Selects a particular "profile" of the SPARK language which will be used during analysis. Currently the only permitted profile value is the default "sequential". The profile value "ravenscar", which is only valid for SPARK95, selects the RavenSPARK analysis profile which allows use of concurrent language features. Note that this option is not enabled by default. Please contact Praxis High Integrity Systems if you wish to make use of this feature.

3.1.4 Other Options

Option	Syntax	Default	Description
noecho	/noe		Suppresses the Examiner's screen output.
annotation_character	/an=*character*		Allows selection of a character other than the default "#" to indicate the start of a SPARK annotation. e.g. `/anno=$` would cause annotation to begin "`--$`" rather than "`--#`".
statistics	/st		Causes the Examiner to insert statistics concerning the consumption of internal tables at the end of the report file. These statistics are described in section 4.1.
nostatistics	/nos	v	Suppresses the production of statistics on the consumption of internal tables
fdl_identifiers	/fd	v	This switch causes the Examiner to recognise (proof-related) FDL identifiers (see section Error! Reference source not found.) as reserved words: this has the effect of making source containing these identifiers syntactically unacceptable.

nofdl_identifiers	/nofd		This switch suppresses the recognition of (proof-related) FDL identifiers (see section Error! Reference source not found.) as reserved words. This action is provided to give backward compatibility with the, no longer supplied, "Standard Examiner" which by default did not recognise FDL identifiers as reserved words.
version	/ve		If this switch appears anywhere on an Examiner command-line, then the Examiner simply prints its banner information, the date, and static limits to the console. All further processing of command-line switches and analysis is terminated. This option is useful for quickly checking the version number, licence information, and static limits of an Examiner.
help	/he		Prints a summary of Examiner command line options to the screen.
original_flow_errors	/o		This switch affects the formatting of the flow errors, printing one line for every error (see section 8.3.4).
error_explanations	/er=*option*	off	Option is one of: *off*, *first_occurrence*, *every_occurrence* (which can be abbreviated to *o, f* and e respectively). Selects whether explanatatory notes are appended to Examiner error messages. When *first_occurrence* is selected, the explanation appears only for the first occurrence of each different error mesage in each of the screen echo, listing files and report file. The use of the html switch overrides error_explanations and turns them off *for the report file only*; this is because explanations are already only one click away when viewing the HTML report file.

3.2 The Default Switch File

The Examiner can read a default switch file as an alternative to placing the above switches on the Examiner's command line each time it is invoked. When run, the Examiner checks for the presence of a file called "spark.sw" *in the current working directory*. If a file of this name is found its contents are interpreted exactly as if they had been typed at the start of the command line. Switches in the default switch file may be entered in free format and the file may include Ada-style comments. For example:

```
    -------------------------------------------------
    -- Default switch file for backward compatibility
    -- with a “Standard” SPARK 83 Examiner
    -------------------------------------------------
    /ada83     -- select SPARK 83
    /nofdl     -- ignore use of FDL identifiers

    -- end of file-----------------------------------
```

When the Examiner finds and uses a default switch file it reports this in both its screen echo messages and in the report file.

For convenience when working in a mixed-platform environment, the Solaris switch character, ‘-‘, may be used in default switch files for any platform.

The Examiner will not allow you to specify duplicate or contradictory options, e.g. both /statistics and /nostatistics. This holds true whether the options are both specified on the command line, or the switch file, or a combination of the two. The one exception to this rule is the combination of /warn and /nowarn which can be used to override each other. The Examiner will process whichever one is specified last.

4 Input and output files

4.1 The report file

The report file summarises the activity and findings of the Examiner. A report file has the following parts:

Banner The banner gives the tool name and release, and names the licensee of the tool. The date and time of analysis is also given.

Options An indication of the analysis options used by the Examiner, including default values.

Selected files A list of the filenames directly selected for analysis.

Index file list A list of the index files used.

Meta file list A list of the meta files used and the filenames read from them.

Warning selection This lists the causes of warning messages which will be provided in a summarised form rather than reported fully.

Target compiler data A list of the values read from the target compiler data file.

Target configuration file A listing of the target configuration file, including any syntactic or static-semantic errors detected by the Examiner.

Source file list A list of the full file-specs of all the source files used during analysis, both those supplied on the command line (including those specified in meta files) and those read using the index mechanism.

Units required but not found A list of the units which were required during the analysis but could not be located, because

- there was no index file, or
- the given index file did not contain a reference to the required unit, or
- the index file associated the unit with a file which did not exist, or
- the unit was not found in the file associated with it by the index file.

Source file entries A list of entries, one for each source file used. The entry contains the name of the source file, the name of the listing file (if any), a list of the units found or expected in that source file and finally a list of errors found in that file (if any). If a source file could not be opened this is indicated. For each entry in the list of units the Examiner reports whether

- the unit was expected but could not be found - for example if the indication of its presence in a particular file, by the index, was erroneous;
- only lexical and syntax analysis was attempted on the unit (which occurs for example if a required unit is preceded in its file by a unit which is not required);
- a complete analysis, including static-semantic and flow analysis has been attempted.
- a warning summary, summarising the suppressed warnings (if any) and noting (with an asterisk) those which have the potential to invalidate the analysis.

Resource statistics A list summarising the size and usage of the Examiner's internal tables. If a run of the Examiner exhausts any one of these tables, it will produce an error message and stop: at which point it is necessary to switch to using a version of the Examiner with larger internal tables. Monitoring this statistics list may help to predict when this is about to occur; information from this list may be requested by Praxis High Integrity Systems if we need to tailor an Examiner with especially large tables for a particular customer.

This information is only produced if the statistics option is selected.

The tables are:

- the *relation table* which is used to construct the flow relations for each subprogram
- the *string table* which is used to store the identifiers and strings found in the source text
- the *symbol table* which stores the information about each object declared by the program
- the *syntax tree* which stores a symbolic representation of each unit which is examined
- the *VCG heap* which is used in the generation of Verification Conditions.
- the *Record components* table which is used to perform flow analysis of record fields.
- the *Record errors* table which is used in reporting and merging flow errors associated with record fields.

References Where the messages produced by the Examiner refer to external documentation, this section gives a list of the abbreviations used.

4.2 The index file

The index file associates compilation units with the files that contain them. The index file is a text file, whose default extension is “*.idx*”.

Index-file = [Super-index] { Index-entry }
Super-index = “superindex” “is” “in” file-spec

```
Index-entry = File-entry | Component-entry
File-entry = Unit-name Entry-Type “is” “in” file-spec
Entry-type = “auxindex” | “main_program” | “specification” | “body” | “subunit”
Unit-name = Ada-unit-name
Component-entry = Unit-name “components” “are” Unit-name { “,” Unit-name }
```

Tokens, i.e. Ada-unit-names, file-specs, the words “superindex”, “auxindex”, “main_program”, “specification”, “body”, “subunit”, “is” and “in”, may not contain spaces or line breaks. Tokens may be separated by any number of spaces, tabs and/or line breaks. This means that an index entry may be broken over any number of lines and blank lines are ignored.

Comments begin with “–” and are terminated by a line break. A comment is treated as though it were a line break. Hence comments may be placed anywhere in the index file that a line break is allowed, except that there must be at least one space or line-break separating the end of a file-spec from the start of a comment.

To include external index information in an index file, a super-index can be used, which must be the first entry in the file, and to subdivide index information an auxiliary index can be employed. An auxiliary index may only contain references to a single unit and its descendants: this includes both the specification and body of the unit and any units descended from it.

The mechanism which the Examiner uses to search an index file in order to associate a file location with a compilation unit is as follows:

1 The Examiner scans the successive entries in the index file, until it reaches an entry to which either of the following conditions apply.

a) The unit name and type in the index entry match those of the sought compilation unit. In this case the file location used is that given in the index entry.

b) The entry is for an auxiliary index, associated with the unit itself, or its parent or a further-removed ancestor. (In other words, the unit name in the entry is either the name of the sought unit or is a prefix of it.) In this case Step 1 is applied again, to the auxiliary index in place of the original one.

   This process is applied until either an entry meeting condition (a) is found, or it terminates unsuccessfully. (It may be applied several times, if auxiliary indexes themselves contain auxiliary indexes.)

2 If the search of an index (and possibly auxiliary indexes) in the above manner is unsuccessful, Step 1 is then applied to the super-index of the original index, if it has one. Should this be unsuccessful, Step 1 is then applied to the super-index of the super-index, and so on.

Note that the ordering of auxiliary indexes is important: an auxiliary index for a compilation unit A must precede all auxiliary indexes for ancestors of A (e.g. an auxiliary index for a unit X.Y must precede an auxiliary index for X).

4.2.1 SPARK 95 child components

The description above covers aspects of the index file mechanism applicable to both SPARK 83 and SPARK 95. The introduction of child packages in SPARK 95 requires certain extensions to the contents of index files.

Unit names that represent children are expressed in the normal way, for example

```
Autopilot.Altitude specification is in altitude_
```

states that the specification of package Altitude, a child of package Autopilot, is in the file named `altitude_.ada`.

The model of private child packages in SPARK is that they behave rather like embedded packages and so their "own variables" must appear as refinement constituents in the parent body. Private children are thus regarded as *components* of their parent. In the general case, this rule also extends to the public descendants of private children. Therefore, before any package body is examined, the specifications of any private children of the package (and any public descendants of those children) must first be examined.

So that the Examiner can locate these other packages without the user providing them explicitly on the command line, a new form of index file entry has been introduced, which states the names of the components of a given package. This new form of entry is described as a "Component-entry" in the earlier grammar. For example,

```
Autopilot components are Autopilot.Altitude,
                         Autopilot.Heading
```

states that the private children of package Autopilot are called Autopilot.Altitude and Autopilot.Heading. Other index entries would then be used in the normal way (as in the previous example above) to locate the relevant source files.

4.2.2 Example index files

queue.*idx*
```
SuperIndex                    is in library
QueueOperations specification is in queue_
QueueOperations body          is in queue
Stacks          auxindex      is in stacks
```

stacks.*idx*
```
Stacks          specification is in stacks_
Stacks          body          is in stacks
Stacks.Pop      subunit       is in stackpop
Stacks.Push     subunit       is in stackpush
```

In the above example `queue_` and `queue` refer to files queue_.ada and queue.ada which contain Ada source code (note that file names are case sensitive on Unix systems). The name `stacks` refers to file stacks.idx which is another (auxiliary) index file. Finally, `library` refers to a further (super) index file library.idx in which the search will recommence if a required unit cannot be found in the index file supplied and the auxiliary indexes specified within it.

4.3 The warning control file

The Examiner generates semantic warnings when certain program constructs are found. The warnings are raised because the Examiner identifies properties of the SPARK source which, although not illegal, may be of interest to the user or because certain language features could be used in a manner that changes the meaning of a program in ways that the Examiner could not detect. For example, representation clauses could be used to cause two variables to overlap so that changing one also changed the other. The warnings generated are listed in sections Error! Reference source not found. and.Error! Reference source not found..

The warning control file, which is a text file whose default extension is "*.wrn*", provides a flexible means of selecting how these warnings will be displayed. By default they are indicated in exactly the same way as errors with a message and location pointer appearing in both report and listing file. Entries in the warning control file allow individual categories of warnings, including pragmas selected by name, to be reported instead in a summary form where just the number of each found is appended to the report and listing file. The mechanism allows, for example, innocuous pragmas such as Page to be ignored while continuing to flag more serious ones.

Comments can be included in the warning control file using the normal Ada syntax.

4.3.1 File format

Warning-control-file = { Warning-entry }
Warning-entry = Keyword | Pragma-selection
Pragma-selection = "pragma all" | "pragma" pragma-identifier

The keywords to be used to suppress the individual warnings are given with the corresponding warning messages in sections Error! Reference source not found. and Error! Reference source not found..

Shorter unique representations of these options are recognised. Each language feature placed in the warning control file selects that feature for summary rather than full reporting. "Pragma all" selects all pragmas (other than those such as pragma Import which are recognised by the Examiner) for summary reporting; alternatively a number of pragmas can be selected individually by name.

4.3.2 Example of a warning control file

A warning control file with the following contents, selected using the warning_file option would cause warning messages connected with direct updating of package own variables, and with the pragmas Page and List, to be summarised; all other warnings would be reported in full.

```
direct
pragma list
pragma page
```

4.4 The meta file

A meta file is a list of filenames: this allows the user to specify that this list of files is to be Examined, simply by naming the meta file (preceded by the @ character) on the command line. The files specified in this way behave exactly as if they had been specified on the command line, so a listing file is produced for each (unless specifically suppressed). This method allows the repeated Examination of groups of files without the need to specify them on the command line. The default extension for a meta file name is *.smf*. Meta files can include Ada-style comments.

4.4.1 File format

```
Meta-file = Argument-list
Argument-list = Argument { Argument }
Argument = ( File-spec [ Argument-option ] ) | Meta-file-spec
Argument-option = ( "/listing_file=" file-spec | "/nolisting_file" )
Meta-file-spec = "@"file-spec
```

Note that the format for the arguments is exactly as on the command line, except that here they may be separated by line breaks. Note also that meta files can be nested. For convenience in multi-platform environments, argument options may optionally be introduced with the Solaris switch character **' - '** on all platforms.

4.4.2 Example of using a meta file

A meta file could be used as an alternative to an index file to specify common sets of files to be analysed. The following hypothetical example would cause the Examiner to process the files specified in common.smf followed by those in myunit.smf and then main_.ada and main.ada (again, note that file names are case sensitive on Unix systems). It would not produce listing files for any of the files specified in common.smf but would produce listings with the extension .lst for all the other files.

```
spark @common.smf @myunit.smf main_.ada main.ada
```

common.smf

```
constants_.ada /nolisting
io_.ada /nolisting
library_.ada /nolisting
```

myunit.smf

```
stack_.ada
queue_.ada
```

4.5 The target compiler data file

This file allows various implementation dependent values to be supplied to the Examiner. The availability of these makes significant improvements to the generation and discharge of VCs associated with Run-time Checks; there is also a positive impact on some wellformation checks. The use of the target compiler data file is mutually exclusive with the use of the target configuration file.

It is only possible to supply values for `Integer'first`, `Integer'last` (and their `Long_Integer` equivalents): the Examiner will also deduce the values of `Positive'last` and `Natural'last` from `Integer'last`. The use of the target compiler data file is now deprecated, and it is recommended that new users use the target configuration file detailed at 4.6.

The format of each line of the file is:

```
line = typemark'attribute_name "=" integer | based_literal
typemark = "integer" | "long_integer"
attribute_name = "first" | "last"
```

Note that the lines in the file do NOT end with semicolons.

4.5.1 Example

```
integer'first = -32768
integer'last  =  32767
```

4.6 The target configuration file

This file serves a similar purpose to the target data file, in that it allows implementation dependent values to be supplied to the Examiner. However, it is a significantly more general mechanism, and has a greater positive impact on the generation and simplification of VCs, as well as static semantic checking. The use of the target configuration file is mutually exclusive with the use of the target data file. The use of the target configuration is recommended.

NB. If the target configuration file is specified on the command line, but cannot be found by the Examiner, then analysis of the source files will not be carried out, and a warning will be emitted.

4.6.1 Syntax

The format of the file resembles a number of SPARK package specifications, concatenated in one file. The grammar of the file is as follows:

```
config_file = config_defn { , config_defn }
config_defn = "package" package_name "is" [seq_defn] "end" package_name ";"
seq_defn = defn { , defn }
defn = fp_type_defn | int_type_defn | int_subtype_defn | fp_const_defn | int_const_defn |
             private_defn
fp_type_defn = "type" fp_type_name "is" "digits" int_literal "range" fp_literal .. fp_literal ";"
int_type_defn = "type" int_type_name "is" "range" int_expr ".." int_expr ";"
private_defn = "type" private_type_name "is" "private" ";"
int_expr = un_exp_part [ add_sub int_literal ]
add_sub = "+" | "-"
un_exp_part = [ "-" ] exp_part
```

exp_part = int_literal "**" int_literal | int_literal
int_subtype_defn = "subtype" int_subtype_name "is" simple_name "range" int_expr ".." int_expr ";"
fp_const_defn = fp_const_name ":" "constant" ":=" fp_literal ";"
int_const_defn = int_const_name ":" "constant" ":=" int_expr ";"
int_literal = <a valid SPARK integer literal>
fp_literal = <a valid SPARK floating point literal>

Although the format of the target configuration file resembles a SPARK source file, it is important to note that it is not, and that only items from the above grammar are allowed: in particular, the range of acceptable expressions is much more limited. Standard Ada comments may also be used.

4.6.2 Example

The following is an example target configuration file for GNAT Pro 3.14a1 under Win32.

```
package Standard is

   type Integer is range -2**31 .. 2**31-1;
   type Short_Short_Integer is range -2**7 .. 2**7-1;
   type Short_Integer is range -2**15 .. 2**15-1;
   type Long_Integer is range -2**31 .. 2**31-1;
   type Long_Long_Integer is range -2**63 .. 2**63-1;

   type Short_Float is digits 6 range -3.40282E+38 ..  3.40282E+38;
   type Float is digits 6 range -3.40282E+38 ..  3.40282E+38;
   type Long_Float is digits 15
     range -1.79769313486232E+308 ..  1.79769313486232E+308;
   type Long_Long_Float is digits 18
     range -1.18973149535723177E+4932 ..  1.18973149535723177E+4932;

end Standard;

package System is
   type Address is private;

   Storage_Unit : constant := 8;
   Word_Size : constant := 32;
   Max_Int : constant := 2**63-1;
   Min_Int : constant := -2**63;
   Max_Binary_Modulus : constant := 2**64;

   Max_Base_Digits : constant := 18;
   Max_Digits : constant := 18;

   Fine_Delta : constant := 1.0842E-19;
   Max_Mantissa : constant := 63;

   subtype Any_Priority is Integer range 0 .. 31;
   subtype Priority is Any_Priority range 0 .. 30;
   subtype Interrupt_Priority is Any_Priority range 31 .. 31;

end System;
```

4.6.3 Legality rules

The Examiner supports ‘package’ specifications in the target configuration file. In SPARK95 mode, both package Standard and package System may be specified. In SPARK83 mode, only package Standard is allowed. The packages may appear in any order.

In package Standard, the following types may be declared:

- Integer;
- Float;
- Any signed integer type which has ‘_Integer’ as a suffix; and
- Any floating point type which has ‘_Float’ as a suffix.

In package System, the type Address may be declared, as may the following subtypes:

- Any_Priority;
- Priority; and
- Integer_Priority.

Additionally, the following named numbers may be defined:

- Storage_Unit;
- Word_Size;
- Max_Binary_Modulus;
- Min_Int;
- Max_Int;
- Max_Digits;
- Max_Base_Digits;
- Max_Mantissa; and
- Fine_Delta.

4.6.4 Static semantics

The ‘package’ specifications are expected to accord to the normal SPARK rule that the package name be specified in the ‘end’ clause. All declarations within the packages are optional, subject to any additional rules below.

In package Standard, type Integer and all ‘_Integer’ types are expected to be signed integer types. Similarly, type Float and all ‘_Float’ types are expected to be constrained floating point types.

In package System, which may only be included in SPARK95 mode, the following well-formedness checks apply:

- Type Address must be private. The declaration of type Address implicitly declares a deferred constant System.Null_Address of type Address;
- Storage_Unit, Word_Size, Max_Binary_Modulus, Min_Int, Max_Int, Max_Digits, Max_Base_Digits and Max_Mantissa are all expected to be named integers.
- Fine_Delta is expected to be a named real.
- Subtype Any_Priority is expected to have Integer as its parent type; additionally, if Any_Priority is specified, both Priority and Interrupt_Priority must also be specified.
- Subtypes Priority and Interrupt_Priority are expected to have Any_Priority as their parent types. The range of Priority must include at least 30 values. The declaration of subtype Priority implicitly defines a constant Default_Priority of type Priority.
- The following relations must hold between Any_Priority, Priority and Interrupt_Priority:
  — Any_Priority’First = Priority’First;
  — Any_Priority’Last = Interrupt_Priority’Last;
  — Priority’Last + 1 = Interrupt_Priority’First.
- Max_Binary_Modulus must be a positive power of 2.

In addition, standard SPARK rules on redeclaration of existing identifiers, empty ranges and legality of subtype ranges apply.

4.6.5 Configuration file generator

An Ada source file is distributed with the Examiner which will automatically generate a valid target configuration file when compiled and run; it is named *confgen.adb* and will be located in the same directory as the Examiner binary. Please note that the resultant configuration file will only be valid for SPARK95 usage, since it includes package System. The packages specifications are generated on the standard output, and can be redirected to a file in the normal fashion.

The configuration file generator will probably require some minor modification (depending on whether the target compiler supports Long_Integer, for example), so it is suggested that the user inspect the source code before use.

NOTE: In an environment where both host and target cross compilers are being used, it is very important that the configuration file is valid for the *final target computer* (i.e. using the cross compiler), not the host compiler.

4.7 VC output file structure

The output files from the VC generator are either:

- produced in a directory tree structure as described in section 4.7.1 (this is the default for NT and Sun); or
- produced in a single directory as described in section 4.7.2 (this is the default for VAX).

There are important considerations for VAX users before they adopt the tree output form. VAX VMS, prior to version 6.0, imposes a limit of 8 on directory nesting depth. If the creation of subdirectories for semantic output files causes this limit to be exceeded, *the output files will not be written*; this will be indicated by the error message:

```
!!! Unable to create output files for semantic analysis
```

being displayed for those subprograms affected. This VMS limitation has been removed with version 6.0. Use of the directory tree structure option is not recommended for users with earlier versions of VMS.

4.7.1 Tree structure

A tree of subdirectories is created beneath the current directory with one level for each level of embedding of scopes present in the code from which VCs or PFs are being generated. Within these directories the output files take their name from the name of the subprogram from which they were generated. Thus VCs for subprogram P within package K will appear in .\k\p.vcg , while those for subprogram Q embedded directly within P will appear in .\k\p\q.vcg .

The following short example illustrates the process.

```
package body P
is
  procedure Double(X : in out Integer)
  --# derives X from X;
  --# post X = X~ * 2;
  is
    function Sum(A, B : Integer) return Integer
    --# return A + B;
    is
    begin
      return A + B;
    end Sum;

  begin --Double
    X := Sum(X, X);
  end Double;
end P;
```

This would produce the following directories and files if VCs were generated:

- A directory called p containing a directory called double and files double*.fdl*, double*.rls* and double*.vcg*
- Files sum*.fdl*, sum*.rls* and sum*.vcg* within directory p\double

VCs or PFs generated from subprograms in child packages appear in a similar directory sub-tree rooted at a directory with the name of the parent package followed by a single underbar. For example, generation of VCs from:

```
package Parent.Child
is
  procedure Inc(X : in out Integer)
  --# derives X from X;
end Parent.Child;
```

would result in a directory called parent_ containing a directory called child containing files inc*.fdl*, inc*.rls* and inc*.vcg*.

If the length of any file name exceeds the maximum length permitted by the operating system it is truncated. Note that this could lead to ambiguous file names in some cases, causing the output from more than one subprogram to be written to files of the same name. A warning message is displayed if file name truncation occurs, see section Error! Reference source not found..

4.7.2 Flat structure

The name of each file generated is created by concatenating the name of the subprogram being analysed with name of each package or subprogram enclosing it, separating each element with underbar characters. Thus VCs for subprogram P in package K will appear in k_ _ _p.*vcg*, while those for subprogram Q embedded directly within P will appear in k_ _ _ p_ _ *q.vcg*.

The example given in section 4.7.1 would produce the following files if VCs are generated:

Analysis of subprogram:

Sum in p_ _ _double_ _sum.*fdl*, p_ _ _ double_ _sum.*rls*, p_ _ _ double _ _sum.*vcg* and Double in p_ _ _double.*fdl*, p_ _ _double.*rls* and p_ _ _double.*vcg*.

For the child package examples in section 4.7.1 the analysis of procedure *Inc* would appear in the files parent_ _child_ _ _inc.*fdl,* parent_ _child_ _ _inc.*rls* and parent_ _child_ _ _inc.*vcg*.

The use of 3 underbars in file names associated with library packages and two for those associated with child package is to ensure that embedded and child packages never generate the same file name.

If the length of the concatenated file name exceeds the maximum length permitted by the operating system, the file name will be truncated. Note that this could lead to ambiguous file names in some cases causing the output from more than one subprogram to be written to files of the same name. A warning message is displayed if file name truncation occurs, see section Error! Reference source not found..

4.7.3 Temporary files

The Examiner makes use of temporary files for the merging of error messages into report and listing files. By default, these temporary files are created in one of the following locations: the directory indicated by the %TMP% environment variable; or, if this is not set, C:\TEMP; or, if this does not exist, the current working directory. Users should ensure that they have write privileges for this location and that there is sufficient space for the temporary files to be created. If desired, the location where temporary files are created can be changed by changing the value of environment variable %TMP%.

4.8 HTML Output

The Examiner can produce browsable HTML output files that make inspection of the Examination results much easier. The output is compliant with the transitional HTML 4.0 specification and can be viewed in any HTML 4.0 browser. HTML generation is invoked using the /html switch described in the table at section 3.1.2. HTML generation is not an option on the VAX

4.8.1 Files Generated

In addition to the usual output, the Examiner produces the following files during HTML generation:

spark.htm	This is the starting point for viewing the HTML output.

spark_rep.htm The HTML version of the Examiner's report file. The file's name is based on the name of the report file and will change if a different report file name is used.

<file>_lst.htm For each file specified on the command line or in a meta-file for Examination, a listing file is produced. The Examiner produces an HTML version of the listing file also. The name of the HTML listing file is based on the name of the plain-text listing file; therefore using the /listing_extension option will change the names of the files. Suppression of listing file generation with the /nolisting option will suppress generation of the HTML listing file also.

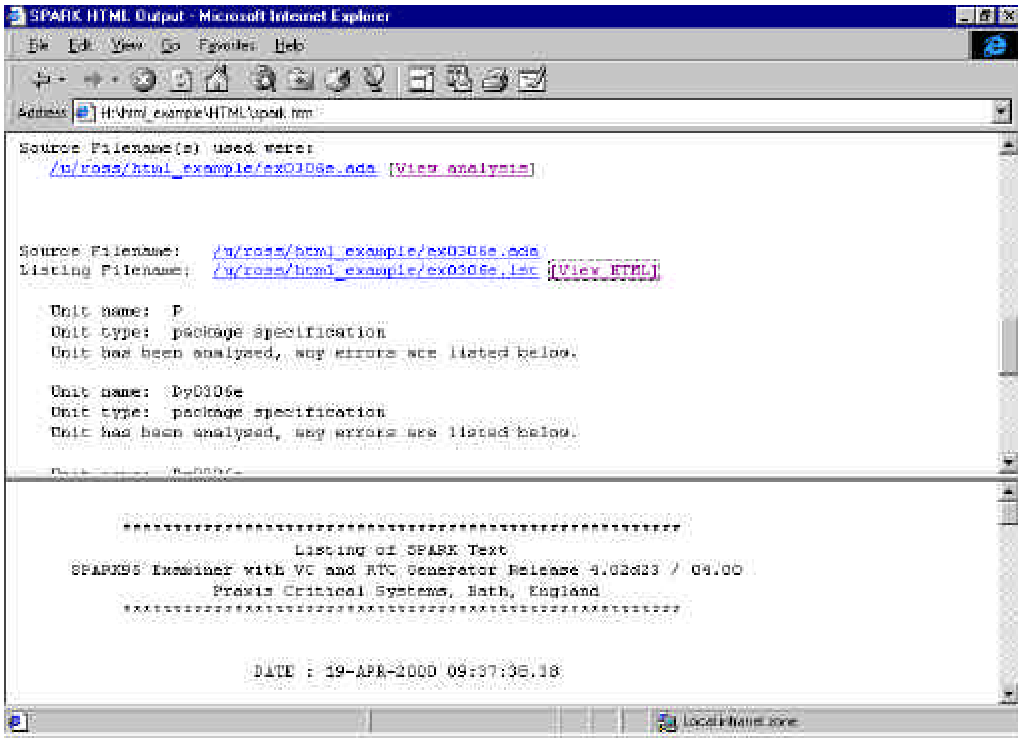
errors.htm This file contains explanations of error messages and is referenced by the HTML versions of the report file and listing files.

blank.htm This file ensures compatibility with all browsers. This is the default file for the bottom frame, as displayed when spark.htm is opened.

4.8.2 Browsing the Report File

The Examiner generates HTML output in a subdirectory of the [current working/default] directory. The name of this directory can be specified on the command-line as a parameter to the html switch. If no directory name is given, then “HTML” is used. If this directory does not exist, it is created by the Examiner. The HTML output can be viewed by opening the file "spark.htm" from the HTML subdirectory in an HTML browser.

This file splits the browser window into two frames. The top frame contains the report file, which is an HTML version of the report file described in section 4.1 and acts as a navigation tool. Following links in the report file opens files in the bottom frame for viewing. Figure 1 shows the report file in the top frame and a listing file being viewed in the bottom frame.


SPARK HTML Output - Microsoft Internet Explorer
File Edit View Go Favorites Help
Address H:\html_example\HTML\spark.htm
Local intranet zone

```
Source Filename(s) used were:
   /u/ross/html_example/ex0306e.ada [View analysis]


Source Filename:   /u/ross/html_example/ex0306e.ada
Listing Filename:  /u/ross/html_example/ex0306e.lsc [View HTML]

   Unit name:  P
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

   Unit name:  Dy0306e
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.
```

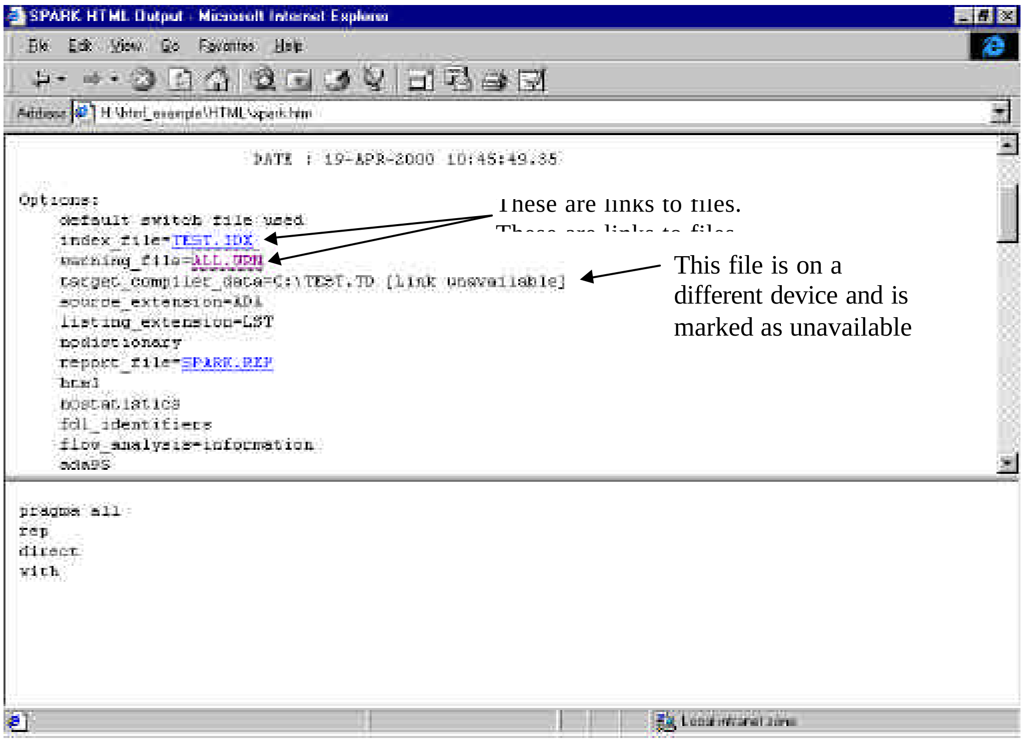
```
           **********************************************************
                          Listing of SPARK Text
      SPARK95 Examiner with VC and RTC Generator Release 4.02d23 / 04.00
                   Praxis Critical Systems, Bath, England
           **********************************************************


                        DATE : 19-APR-2000 09:37:35.38
```

Figure 2

4.8.2.1 Option links

The report file contains links to any files specified on the command-line. Figure 2 shows the links that are available in this section of the report file. Figure 2 also shows that links across physical devices are marked as unavailable. This is because the links can not be guaranteed to be available when browsing.

In figure 2, the warning control file link has been selected and is displayed in the bottom frame.


SPARK HTML Output - Microsoft Internet Explorer
File Edit View Go Favorites Help
Address H:\html_example\HTML\spark.htm
These are links to files.
This file is on a different device and is marked as unavailable
Local intranet zone

```
                    DATE : 19-APR-2000 10:45:49.35

Options:
    default switch file used
    index_file=TEST.IDX
    warning_file=ALL.WRN
    target_compiler_data=C:\TEST.TD (Link unavailable)
    source_extension=ADA
    listing_extension=LST
    nodictionary
    report_file=SPARK.REP
    html
    nostatistics
    fdl_identifiers
    flow_analysis=information
    ada95
```

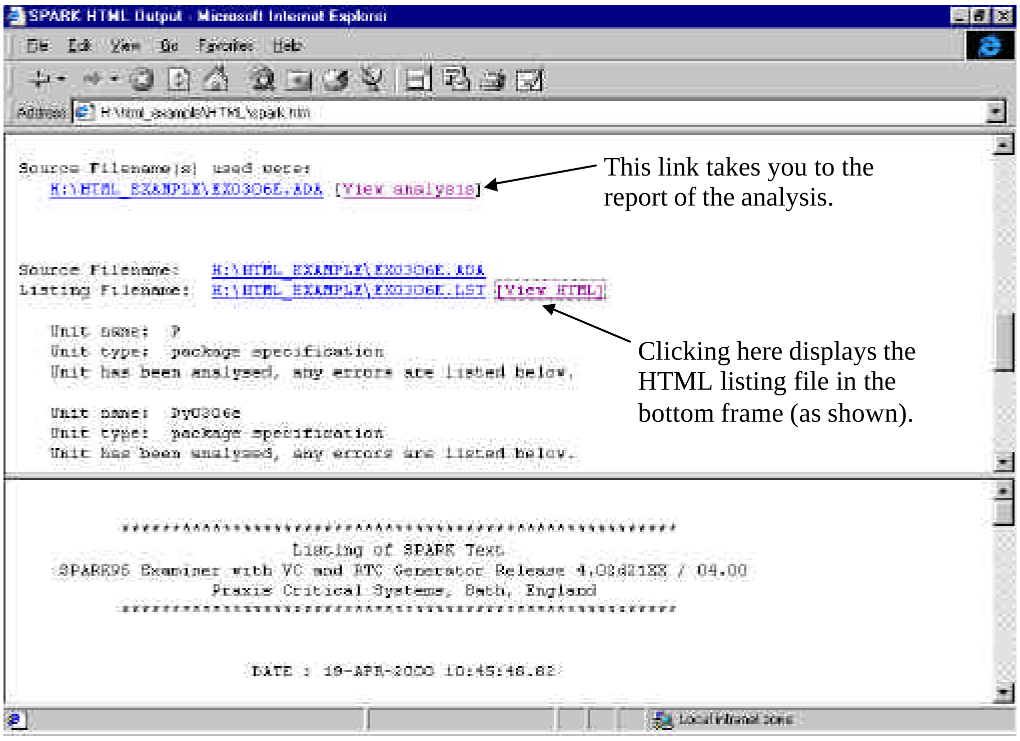
```
pragma all
rep
direct
with
```

Figure 3

4.8.2.2 File links

The "Selected Files", "Index Filename(s)" and "Meta Files" sections of the report file contain links to the files that are referenced.

The "Source Filename(s)" section also contains links to the source files that are referenced. Alongside there are links to the analysis sections of the report file. Clicking on "[View analysis]" will take you to the report of the analysis of that file (see figure 3).


SPARK HTML Output - Microsoft Internet Explorer
File Edit View Go Favorites Help
Address H:\html_example\HTML\spark.htm
This link takes you to the report of the analysis.
Clicking here displays the HTML listing file in the bottom frame (as shown).
Local intranet zone

```
Source Filename(s) used were:
   H:\HTML_EXAMPLE\EX0306E.ADA [View analysis]


Source Filename:    H:\HTML_EXAMPLE\EX0306E.ADA
Listing Filename:   H:\HTML_EXAMPLE\EX0306E.LST [View HTML]

   Unit name:  P
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.

   Unit name:  Py0306e
   Unit type:  package specification
   Unit has been analysed, any errors are listed below.
```

```
          *******************************************************
                          Listing of SPARK Text
    SPARK95 Examiner with VC and RTC Generator Release 4.0382138 / 04.00
                   Praxis Critical Systems, Bath, England
          *******************************************************


                       DATE : 19-APR-2000 10:45:48.82
```

Figure 4

4.8.2.3 Analysis section links

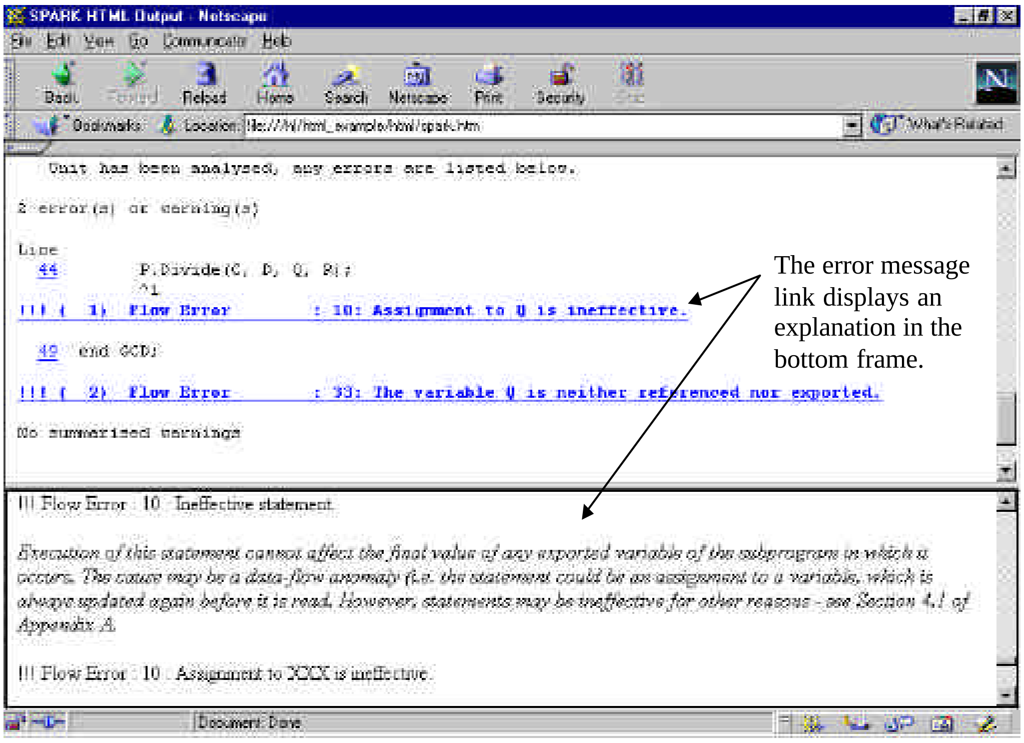
The analysis section links are also shown in figure 3. The source filename and listing filename are referenced and there is also a link to the HTML version of the listing file.

4.8.2.4 Error links

If the analysis section of the report file shows that errors were found, the error reports also have useful links.

The error message itself is linked to an explanation of the error, displayed in the bottom frame (see figure 4).

Clicking on the line number of the error will display that line in the listing file in the bottom frame (see figure 5). Note that the line number links will not work if no listing file was generated.


SPARK HTML Output - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///H/html_example/html/spark.htm
What's Related
Document: Done

```
   Unit has been analysed, any errors are listed below.

2 error(s) or warning(s)

Line
  44         P.Divide(C, D, Q, R);
             ^1
!!! (  1)  Flow Error         : 10: Assignment to Q is ineffective.

  49  end GCD;

!!! (  2)  Flow Error         : 33: The variable Q is neither referenced nor exported.

No summarised warnings
```

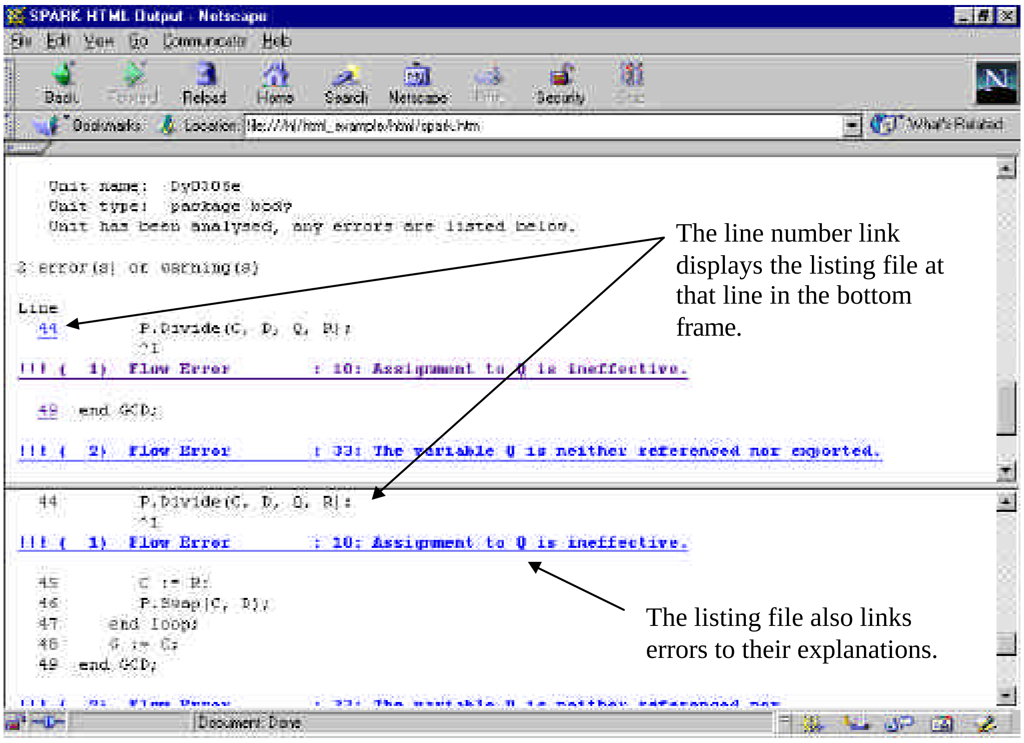
!!! Flow Error : 10 : Ineffective statement.

Execution of this statement cannot affect the final value of any exported variable of the subprogram in which it occurs. The cause may be a data-flow anomaly (i.e. the statement could be an assignment to a variable, which is always updated again before it is read. However, statements may be ineffective for other reasons - see Section 4.1 of Appendix A.

!!! Flow Error : 10 : Assignment to XXX is ineffective.

The error message link displays an explanation in the bottom frame.

Figure 5


SPARK HTML Output - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: file:///H/html_example/html/spark.htm
What's Related
Document: Done

```
   Unit name:  Dy0306e
   Unit type:  package body
   Unit has been analysed, any errors are listed below.

2 error(s) or warning(s)

Line
  44         P.Divide(C, D, Q, R);
             ^1
!!! (  1)  Flow Error         : 10: Assignment to Q is ineffective.

  49  end GCD;

!!! (  2)  Flow Error         : 33: The variable Q is neither referenced nor exported.
```

```
  44         P.Divide(C, D, Q, R);
             ^1
!!! (  1)  Flow Error         : 10: Assignment to Q is ineffective.

  45         C := R;
  46         P.Swap(C, D);
  47      end loop;
  48      G := C;
  49  end GCD;

!!! (  2)  Flow Error         : 33: The variable Q is neither referenced nor
```

The line number link displays the listing file at that line in the bottom frame.

The listing file also links errors to their explanations.

Figure 6

4.8.2.5 Listing File Links

The listing file also contains links from error messages to their explanations, as shown in figure 5.

5 Lexical and syntactic analysis

5.1 General description

The Examiner employs an LALR parsing mechanism. It is similar to a conventional LALR parser, except that it employs explicit shift and reduce tables. The tables have been generated explicitly to facilitate the correctness proof of the parser itself.

5.2 Error messages

The Examiner employs a uniform method of error reporting, in terms of the syntactic entities (terminal and non-terminal symbols) of the SPARK grammar. Reserved words, operators and punctuation marks in syntax error messages are enclosed in quotes.

In almost all respects the grammar employed by the Examiner is the same as that given in the SPARK Definition; however, some minor transformations have been applied, to reduce the original grammar to LALR form. The messages given are nevertheless mostly quite comprehensible. Occasionally misleading messages can be produced because of the form of the SPARK grammar; common cases are listed below, with a brief

*** Syntax Error : “;” expected.
If this is reported at the end of the input file it may well be caused by the misspelling of an identifier in a hide directive. The parser then skips all the following text looking for the misspelled identifier but finds the end of file first where it reports a syntax error.

*** Syntax Error : No APRAGMA can be start with reserved word “IS”.
This can occur when a stub for an embedded subprogram is wrongly terminated by a semicolon.

*** Syntax Error : No complete PROCEDURE_SPECIFICATION can be followed by ANNOTATION_START here.
This can occur when the reserved word body has been omitted from the declaration of a package body. This error will occur at the annotation placed between the specification and the reserved word is of the first subprogram.

*** Syntax Error : No complete PROCEDURE_SPECIFICATION can be followed by reserved word “IS” here.
This can occur when the reserved word body has been omitted from the declaration of a package body. This error will occur at the reserved word is which introduces the body of the first subprogram.

*** Syntax Error : reserved word “INHERIT” expected.
This occurs where the annotation on a subprogram body is placed after the reserved word is instead of before it.

*** Syntax Error : No complete SIMPLE_EXPRESSION can be followed by “)” here.
This can occur in an aggregate expression when there is a mixure of named and positional association being used.

*** Syntax Error : No complete SIMPLE_EXPRESSION can be followed by “,” here.

This can occur in an aggregate expression when there is a mixure of named and positional association being used.

5.3 Reserved words

In addition to the reserved words of Ada, certain additional words listed below are reserved words of SPARK. These additional reserved words are associated with SPARK's core annotations and proof contexts.

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

In addition to these reserved words users may not use certain other identifiers which are reserved for use in FDL (Functional Description Language) which is the underlying logic in which theorems about SPARK programs are expressed. Use of these identifiers as Ada identifiers would lead to ambiguous and badly typed VCs. The predefined FDL identifiers involved are listed below. A complete list of the reserved words of SPARK and FDL is also given in the SPARK Definition. Note that this restriction can be avoided by use of the nofdl_identifier command line switch (see Section 3.1) although doing so will prevent use of the Examiner's VC, PFS and RTC generation capabilities.

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

For backward compatibility with earlier definitions of the SPARK language the reserved word "some" is controlled by the command line option "fdl_identifiers". If "nofdl_identifiers" is selected then "some" may be used as a normal identifier.

"abstract" is a reserved word of SPARK 95. For SPARK 83 it is predefined FDL identifier which can be controlled by the "fdl_identifiers" command line option.

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

6 Static semantic analysis

6.1 General description

This analysis involves checking that a text obeys the static-semantic rules of SPARK (other than rules relating to control-, data- and information-flow, which are discussed in chapters 7 and 8).

Where error messages contain references to other documents, such as the SPARK report, the full title of the references used will appear in the report file.

6.2 Error messages

The following section explains error messages, specifically relating to SPARK restrictions, the incorrect use of annotations, or inconsistencies between annotations and executable code. A numeric error code is given in the message, the error messages are presented in numerical order. An explanation for each error message is given except where the message text is self-explanatory

*** Semantic Error :1: The identifier YYY.XXX is either undeclared or not visible at this point.

*** Semantic Error :2: XXX does not denote a formal parameter for YYY.

*** Semantic Error :3: Incorrect number of actual parameters for call of subprogram XXX.

*** Semantic Error :4: More than one parameter association is given for formal parameter XXX.

*** Semantic Error :5: Illegal use of identifier XXX.
Usually associated with the use of an identifier other than a package name as a prefix in a selected component.

*** Semantic Error :6: Identifier XXX is not the name of a variable.

*** Semantic Error :7: Identifier XXX is not the name of a procedure.

*** Semantic Error :8: There is no field named XXX in this entity.
Issued when the selector in a selected component of a record references a non-existent field.

*** Semantic Error :9: Selected components are not allowed for XXX.
Occurs if the prefix to a selected component representing a procedure in a procedure call statement or a type mark is not a package. Also occurs if a selector is applied in an expression to an object which is not a record variable.

*** Semantic Error :10: Illegal redeclaration of identifier XXX.

*** Semantic Error :11: There is no package declaration for XXX.
Issued if a package body is encountered for which there is no package specification.

*** Semantic Error :13: A body for subprogram XXX has already been declared.

*** Semantic Error :14: Illegal parent unit name.
Issued if the name is a "separate" clause of a subunit does not correctly identify a compilation unit. A common cause of this error is a syntax error in the parent unit.

*** Semantic Error :15: No stub exists for XXX.

*** Semantic Error :16: A body for package XXX has already been declared.

*** Semantic Error :17: A body stub for package XXX has already been declared.

*** Semantic Error :18: Identifier XXX is not the name of a package.

*** Semantic Error :19: Identifier XXX is not the name of a procedure.

*** Semantic Error :20: Illegal operator symbol.
Issued if a renaming declaration contains a non-existent operator.

*** Semantic Error :21: This entity is not an array.
Issued if an attempt is made to index into a name which does not represent an array.

*** Semantic Error :22: The type in this declaration is not consistent with the previous declaration of XXX.
Occurs when the type given in the Ada declaration of an own variable differs from that "announced" in the package's own variable clause.

*** Semantic Error :23: No parameter association is given for formal parameter XXX.

*** Semantic Error :24: The identifier XXX (exported by called subprogram) is not visible at this point.

When a procedure is called any global variables exported by that procedure must be visible at the point of call. This error message indicates that the global variable concerned is not visible. It may be that it needs to be added to the global annotation of the procedure containing the call (or some further enclosing subprogram) or it may be that an inherit clause is missing from the package containing the call.

*** Semantic Error :25: The identifier XXX (imported by called subprogram) is not visible at this point.
When a procedure is called any global variables imported by that procedure must be visible at the point of call. This error message indicates that the global variable concerned is not visible. It may be that it needs to be added to the global annotation of the subprogram containing the call (or some further enclosing subprogram) or it may be that an inherit clause is missing from the package containing the call.

*** Semantic Error :26: The deferred constant XXX does not have an associated full definition.
Issued at the end of a package specification if no full declaration has been supplied for a deferred constant declared in the package specification.

*** Semantic Error :27: The private type XXX does not have an associated full definition.
Issued at the end of a package specification if no full declaration has been supplied for a private type declared in the package specification.

*** Semantic Error :28: The own variable XXX does not have a definition.
Issued at the end of a package body if an own variable announced in the package specification has neither been given an Ada declaration nor refined.

*** Semantic Error :29: The subprogram XXX, declared in the package specification, does not have an associated body.

*** Semantic Error :30: Attribute XXX is not yet implemented in the Examiner.
The attribute is identified in Annex K of the SPARK 95 report as a valid SPARK 95 attribute but the Examiner does not currently support it. Please contact Praxis High Integrity Systems if use of this attribute is important to your project. It is possible to work round the omission by putting the use of the attribute inside a suitable function which is hidden from the Examiner.

*** Semantic Error :31: The prefix of this attribute is not an object or type.

*** Semantic Error :32: Illegal type conversion.
Likely causes are type conversions involving record types or non-convertible arrays.

*** Semantic Error :33: Illegal aggregate.
Issued if the prefix of an aggregate is not a composite type.

*** Semantic Error :34: Illegal procedure call.
Issued if a call is made to a user-defined subprogram in a package initialization part.

*** Semantic Error :35: Operator is not declared for these types.
Indicates use of an undeclared operator; this message means that the type on each side of the operator cannot appear with the operator used. e.g. attempting to add an integer to an enumeration literal.

*** Semantic Error :36: Expression is not static.

*** Semantic Error :37: Expression is not constant.

*** Semantic Error :38: Expression is not of the expected type.

*** Semantic Error :39: Illegal use of unconstrained type.

An unconstrained array type or variable of such a type is illegally used. Use of unconstrained arrays in SPARK is limited to passing them as parameters, indexing into them and taking attributes of them. This message also arises if a string literal is used as an actual parameter where the formal parameter is a string subtype. In this case, the error can be removed by qualifying the string literal with the subtype name.

*** Semantic Error :40: Numeric or Time_Span type required.
This operator is only defined for numeric types and, if the Ravenscar Profile is selected, for type Ada.Real_Time.Time_Span.

*** Semantic Error :41: Array type required.
Issued if a subtype declaration taking the form of a constrained subtype of an unconstrained array type is encountered but with a type mark which does not represent an array.

*** Semantic Error :42: Incompatible types.
Issued when a name represents an object which is not of the required type.

*** Semantic Error :43: Range is not constant.

*** Semantic Error :44: Scalar type required.
The bounds of an explicit range must be scalar types.

*** Semantic Error :45: Range is not static.

*** Semantic Error :46: Discrete type required.

*** Semantic Error :47: Definition is not static.
Issued if an array type definition is encountered where one or more of the index types used in the definition is not static.

*** Semantic Error :48: Subtypes of private types are not permitted.
Issued if an attempt is made to declare a subtype of a private type in a location where the full view of the type is not visible.

*** Semantic Error :49: Attribute XXX takes only one argument.
Only SPARK 95 attributes 'Min and 'Max require two arguments.

*** Semantic Error :51: Arrays may not be ordered.
Issued if an ordering operator such as "&lt;" is encountered between objects of an array type other than string or a constrained subtype of string.

*** Semantic Error :52: Only Scalar, String and Time types may be ordered.
Ordering operators are only defined for scalar types and type String plus, if the Ravenscar Profile is selected, types Time and Time_Span in package Ada.Real_Time.

*** Semantic Error :53: Illegal others clause.
In SPARK record aggregates may not contain an others clause.

*** Semantic Error :54: Illegal attribute: XXX.
Issued when an attribute not supported by SPARK is used.

*** Semantic Error :55: Attribute XXX takes no argument.

*** Semantic Error :56: Argument expected.

*** Semantic Error :57: Fixed type definition must have associated range constraint.

*** Semantic Error :58: XXX expected, to repeat initial identifier.
Occurs at the end of a package, subprogram, protected type, task type or loop if the terminal identifier does not match the name or label originally given.

*** Semantic Error :59: Composite subtype definition may not have associated range constraint.
A subtype of the form applicable to a subrange of a scalar type has been encountered but the type provided is not a scalar type.

*** Semantic Error :60: Illegal choice in record aggregate.
In SPARK record aggregates may not contain multiple choices, each field must be assigned a value individually.

*** Semantic Error :61: Illegal occurrence of body stub - a body stub may only occur in a compilation unit.

*** Semantic Error :62: A body for the embedded package XXX is required.
Issued if an embedded package declares subprograms or own variables and no body is provided.

*** Semantic Error :63: XXX is not a type mark.

*** Semantic Error :64: Parameters of function subprograms must be of mode in.

*** Semantic Error :65: Formal parameters of renamed operators may not be renamed.
The names of the parameters used in renaming declarations may not be altered from Left, Right for binary operators and Right for unary operators. These are the names given for the parameters in the ARM and the SPARK Definition requires that parameter names are not changed.

*** Semantic Error :66: Unexpected package initialization - no own variables of package XXX require initialization.
Either the package does not have an initializes annotation or all the own variables requiring initialization were given values at the point of declaration.

*** Semantic Error :67: Illegal machine code insertion. Machine code functions are not permitted in SPARK 83.
This is an Ada 83 rule. Machine code can only be used in procedures.

*** Semantic Error :68: Illegal operator renaming - operators are defined on types not subtypes.
Issued if an attempt is made to rename an operator using a subtype of the type for which it was originally implicitly declared.

*** Semantic Error :69: pragma XXX has two parameters.

*** Semantic Error :70: pragma Import expected.

*** Semantic Error :70: pragma Interface expected.

*** Semantic Error :71: This expression does not represent the expected subprogram or variable name XXX.
Issued if the name supplied in a pragma interface, import or attach_handler does not match the name of the associated subprogram or variable.

*** Semantic Error :72: Unexpected pragma Import.
Pragma import may only occur in a body stub, or immediately after a subprogram declaration in the visible part of a package, or immediately after a variable declaration.

*** Semantic Error :72: Unexpected pragma Interface.
Pragma interface may only occur in a body stub or immediately after a subprogram declaration in the visible part of a package.

*** Semantic Error :73: XXX has already been declared or refined.

Issued if an Ada declaration is given for an own variable which has been refined, or in a refinement clause if an own variable is refined more than once.

*** Semantic Error :74: XXX does not occur in the package own variable list.
A subject of a refinement definition of a package must be an own variable of that package.

*** Semantic Error :75: Illegal use of inherited package.
Issued if an attempt is made to refine an own variable onto an own variable of a non-embedded package.

*** Semantic Error :76: Identifier XXX is already declared and cannot be the name of an embedded package.
Issued when a refinement clause in a package body attempts to name an embedded package own variable as a refinement constituent and the name given for the embedded package is already in use.

*** Semantic Error :77: Variable XXX should occur in this own variable clause.
Occurs in the own variable clause of a package embedded in another package if an own variable which is a refinement constituent of an own variable of the enclosing package is omitted.

*** Semantic Error :78: Initialization of own variable XXX is ineffective.
Issued if an own variable occurs in the initialization clause of an embedded package and the own variable concerned is a refinement constituent of another own variable which is not listed in the initialization specification of its package.

*** Semantic Error :79: Variable XXX should occur in this initialization specification.
Occurs in the initialization clause of a package embedded in another package if an own variable which is a refinement constituent of an initialized own variable of the enclosing package is omitted.

*** Semantic Error :80: Unexpected own variable clause - no variable in this clause is a refinement constituent.

*** Semantic Error :81: Own variable clause expected - own variables of this package occur as refinement constituents.

*** Semantic Error :82: Unexpected initialization specification - no own variables of this package require initialization.
An own variable initialization clause and that of its refinement constituents must be consistent.

*** Semantic Error :83: Initialization specification expected - own variables of this package require initialization.
Issued if an own variable does not occur in the initialization clause of an embedded package and the own variable concerned is a refinement constituent of another own variable which is listed in the initialization clause of its package.

*** Semantic Error :84: The refinement constituent XXX does not have a declaration.
Issued at the end of a package if a refinement constituent of a refined own variable has not been given an Ada declaration or further refined.

*** Semantic Error :85: XXX is not a constituent of any abstract own variable appearing in the earlier global definition for this subprogram.
A variable XXX which has occurred in a refined global annotation is neither a variable that occurred in the earlier global definition nor a refinement constituent of any such variable.

*** Semantic Error :86: At least one constituent of XXX was expected in this refined global definition.

If the global annotation of a procedure specification contains an own variable and that own variable is later refined then at least one refinement constituent of the own variable shall appear in the second global annotation supplied for the procedure body.

*** Semantic Error :87: Refined global definition expected for subprogram XXX.
A global definition containing abstract own variables was given in the definition for subprogram XXX, in a package specification. A refined global definition is required in the package body.

*** Semantic Error :88: Variable XXX is not a refinement constituent.

*** Semantic Error :89: XXX is not a private type declared in this package.

*** Semantic Error :90: This operator may not be applied to ranges.

*** Semantic Error :91: Ranges may not be assigned.

*** Semantic Error :92: Named association may not be used here.

*** Semantic Error :93: Number of index expressions differs from number of dimensions of array XXX.

*** Semantic Error :94: Condition is not boolean.
Issued anywhere a boolean expression is required (e.g. in if, exit and while statements) and the expression provided is not of type boolean.

*** Semantic Error :95: Type mark expected.

*** Semantic Error :96: Attribute XXX is not valid with this prefix.

*** Semantic Error :97: Attribute BASE may only appear as a prefix.
'BASE may only be used as a prefix to another attribute.

*** Semantic Error :98: This expression is not a range.

*** Semantic Error :99: Unconstrained array expected.
Occurs if a subtype is declared of an array which is already constrained.

*** Semantic Error :100: Floating point type mark expected.

*** Semantic Error :101: Fixed point type mark expected.

*** Semantic Error :102: This is not the name of a field of record XXX.

*** Semantic Error :103: A value has already been supplied for field XXX.

*** Semantic Error :104: No value has been supplied for field XXX.

*** Semantic Error :105: More values have been supplied than number of fields in record XXX.

*** Semantic Error :106: Range is not of the expected type.

*** Semantic Error :107: Expression is not of the expected type. Actual type is XXX. Expected type is YYY.

*** Semantic Error :108: Expression is not of the expected type. Expected any Integer type.

*** Semantic Error :109: Expression is not of the expected type. Expected any Real type.

*** Semantic Error :110: Use type clauses following an embedded package are not currently supported by the Examiner.

*** Semantic Error :111: Package renaming is not currently supported by the Examiner.

*** Semantic Error :112: A use type clause may not appear here. They are only permitted as part of a context clause or directly following an embedded package specification.

*** Semantic Error :113: Private subprogram declarations are not permitted in SPARK 83.
Private subprograms would not be callable in SPARK 83 and are therefore not permitted; they may be declared and called in SPARK 95.

*** Semantic Error :114: Subtype mark or Range may not be used in an expression in this context.
A subtype mark or an explicit Range attribute may not be used in a context where a simple expression is expected.

*** Semantic Error :120: Pragma import not allowed here because variable XXX is already initialized. See ALRM B.1(24).

*** Semantic Error :148: The abstract proof type XXX may not be used to define an own variable in another package.
Own variables may be "type announced" as being of an abstract proof type only where that type is declared later in the same package. Thus –# own State : T; is legal if --# type T is abstract; appears later in the package; however, –# own State : P.T; is illegal if T is an abstract proof type declared in remote package P.

*** Semantic Error :149: More than one own variable has been announced as being of type XXX which may not therefore be declared as an abstract proof type.
Occurs when an own variable clause announces more than one own variable as being of a type XXX and XXX is later declared as being of an abstract proof type. Each abstract own variable must be of a unique type.

*** Semantic Error :150: Entire variable expected. The names of constants never appear in mandatory annotations.
Issued when a the name of a constant is found in a mandatory annotation such as a global or derives annotation. Constants should not appear in such annotations.

*** Semantic Error :151: The variable XXX does not occur either in the package own variable list or as a refinement constituent.
A variable declared in a package must have been previously announced as either an own variable or as a concrete refinement constituent of an own variable.

*** Semantic Error :152: The number of formal parameters is not consistent with the previous declaration of XXX.

*** Semantic Error :153: The declaration of formal parameter XXX is not consistent with the subprogram's previous declaration.
Issued if the name, type or parameter mode of a parameter is different in the subprogram body declaration from that declared originally.

*** Semantic Error :154: The subprogram or task body XXX does not have an annotation.
A subprogram or task body must have a global annotation if it references global variables; a procedure or task body must have a dependency relation to perform information flow analysis.

*** Semantic Error :155: Unexpected annotation - all annotations required for procedure or task body XXX have already occurred.

*** Semantic Error :156: Entire variable expected.
Issued when an identifier which SPARK requires to be an entire variable represents something other than this. Most commonly this message occurs when a component of a structured variable appears in a core annotation.

*** Semantic Error :157: The name XXX already appears in the global variable list.

*** Semantic Error :158: XXX is a formal parameter of this subprogram.
Issued in a global annotation if it names a formal parameter of the subprogram.

*** Semantic Error :159: The name XXX has already appeared as an exported variable.

*** Semantic Error :160: The name XXX already appears in the list of imported variables.

*** Semantic Error :161: Exportation of XXX is incompatible with its parameter mode.
Issued if a parameter appears as an export to a procedure when it is of parameter mode in.

*** Semantic Error :162: Importation of XXX is incompatible with its parameter mode.
Issued if a parameter appears as an import to a procedure when it is of parameter mode out.

*** Semantic Error :163: Subprogram XXX cannot be called from here.
SPARK contains rules to prevent construction of programs containing recursive subprogram calls; this error message occurs if a procedure or function is called before its body has been declared. Re-ordering of subprogram bodies in the package concerned will be required.

*** Semantic Error :164: Actual parameter of mode in out or out must be the name of a variable.

*** Semantic Error :165: This parameter is overlapped by another one, which is exported.
Violation of the anti-aliasing rule.

*** Semantic Error :166: This parameter is overlapped by an exported global variable.
Violation of the anti-aliasing rule.

*** Semantic Error :167: Imported variable XXX is not named in the initialization specification of its package.
Issued when an own variable which is imported into the main program procedure (or a task when the Ravenscar profile is enabled) has not been declared as being initialized by its package. At the main program level the only imports that are permitted are initialized own variables of inherited packages. There are two possible cases to consider: (1) the main program should be importing the variable in which case it should be annotated in its package with –# initializes (and, of course, actually initialized in some way) or be an external variable or protected variable which is implicitly initialized; or (2) the own variable concerned is not initialized at elaboration, should not therefore be considered an import to the main program and should be removed from the main program's import list.

*** Semantic Error :168: XXX is a loop parameter, whose updating is not allowed.

*** Semantic Error :169: Global variables of function subprograms must be of mode in.
It is an important property of SPARK that functions cannot have side-effects, therefore only the reading of global variable is permitted. It is usually convenient to omit modes from function global annotations but use of mode 'in' is permitted.

*** Semantic Error :170: XXX is a formal parameter of mode in, whose updating is not allowed.

*** Semantic Error :171: XXX is a formal parameter of mode out, whose value cannot be read.

*** Semantic Error :172: The actual parameter associated with an exported formal parameter must be an entire variable.
Issued if an actual parameter which is an array element is associated with an exported formal parameter in a procedure call. Exported parameters must be either entire variables or a record field.

*** Semantic Error :173: This exported parameter is named in the global definition of the procedure.

Violation of the anti-aliasing rule.

*** Semantic Error :174: XXX is not an own variable.
Occurs in initialization specifications if something other than a variable is listed as being initialized.

*** Semantic Error :175: Information flow from XXX to YYY violates the selected information flow policy.

*** Semantic Error :180: Entire composite constant expected.
Issued when an identifier which SPARK requires to be an entire composite constant represents something other than this.

*** Semantic Error :181: Invalid policy for constant proof rule generation.

*** Semantic Error :182: Rule Policy for YYY.XXX already declared in current scope.
Issued when a rule policy has already been declared for this constant within this declarative region. This rule policy will be ineffective.

*** Semantic Error :190: The name XXX already appears in the inherit clause.

*** Semantic Error :191: The name XXX already appears in the with clause.

*** Semantic Error :200: The parameter XXX is neither imported nor exported.
Each formal parameter of a subprogram shall be imported or exported or both.

*** Semantic Error :201: The global variable XXX is neither imported nor exported.
Every variable in a global definition shall be either imported or exported or both.

*** Semantic Error :250: The 'Size value for type XXX has already been set.

*** Semantic Error :251: The attribute value for XXX'Size must be of an integer type.

*** Semantic Error :252: The attribute value for XXX'Size must be a static simple expression.
The value of 'Size must be static and must be of an integer type.

*** Semantic Error :253: The attribute value for XXX'Size must not be negative.
The value of 'Size must be a positive integer or zero.

*** Semantic Error :254: The Size attribute can only be specified for a first subtype.
Setting 'Size for a user-defined non-first subtype is not permitted. See Ada95 LRM 13.3(48).

*** Semantic Error :255: The Address attribute can only be specified for a variable, a constant, or a program unit.
Ada95 LRM Annex N.31 defines a program unit to be either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal.

*** Semantic Error :273: Own variable XXX may not be refined because it was declared with a type mark which has not subsequently been declared as an abstract proof type.
Where a type mark is included in an own variable declaration it indicates that the own variable will either be of a concrete type of that name (which may be either already declared or be declared later in the package) or of an abstract proof type declared in the package specification. In the former case the refinement is illegal because own variables of concrete Ada types may not be refined. In the latter case it is legal; however, no suitable proof type declaration has been found in this case.

*** Semantic Error :300: Renaming declarations are not allowed here.
A renaming declaration must be the first declarative item of a package body or main program or it must be placed immediately after the declaration of an embedded package.

*** Semantic Error :301: Renaming or use type declarations here can only rename subprograms in package XXX.
A renaming declaration may be placed immediately after the declaration of an embedded package; in this case it may only rename subprograms declared in that package.

*** Semantic Error :302: The subprogram specification in this renaming declaration is not consistent with the declaration of subprogram XXX.
Issued in a subprogram renaming declaration if it contains parameter names, numbers or types which differ from those originally declared.

*** Semantic Error :303: An operator can only be renamed by the same operator.
Issued if a renaming declaration has a different operator on each side of the reserved word RENAMES.

*** Semantic Error :304: A renaming declaration for operator XXX is not allowed.

*** Semantic Error :305: The specification in this renaming declaration is not consistent with the implicit declaration of operator XXX.
Issued in an operator renaming declaration if it contains types which differ from those applicable to the operator being renamed.

*** Semantic Error :306: Operator XXX is already visible.
Occurs in an operator renaming declaration if an attempt is made to rename an operator which is already visible. (The message will also appear as a secondary consequence of trying to rename an operator between undeclared types.).

*** Semantic Error :307: The implicit declaration of this operator does not occur in package XXX.

*** Semantic Error :308: Type is limited.
Issued if an attempt is made to assign a variable of a type which is limited or which contains a limited type.

*** Semantic Error :309: Operator not visible for these types.
This message means that the operator exists between the types on each side of it but that it is not visible. The most likely cause is that the types concerned are defined in another package and that renaming is required to make the operator visible.

*** Semantic Error :310: The % operator may only appear in an assert or check statement in a for loop.
The % operator is used to indicate the value of a variable on entry to a for loop. This is because the variable may be used in the exit expression of the loop and may also be modified in the body of the loop. Since the semantics of Ada require the exit expression to be fixed after evaluation we require a way of reasoning about the original value of a variable prior to any alteration in the loop body. No other situation requires this value so % may not be used anywhere else.

*** Semantic Error :311: Announced own variable types may not be implemented as unconstrained arrays.
Where an an own variable is announced as being of some type, SPARK requires that type to be declared; the declaration cannot be in the form of an unconstrained array because SPARK prohibits unconstrained variables.

*** Semantic Error :312: A subprogram can only be renamed to the same name with the package prefix removed.

*** Semantic Error :313: Only one main program is permitted.

*** Semantic Error :314: Own variable XXX has been refined and may not appear here.

Issued if an attempt is made to use, in a second annotation, an own variable which has been refined. Second annotations should use the appropriate refinement constituents of the own variable.

*** Semantic Error :315: Unsupported proof context.
Certain proof contexts have been included in the syntax of SPARK but are not yet supported; this error message results if one is found.

*** Semantic Error :317: Tilde, in a function return annotation, may only be applied to an external variable of mode IN.
The tilde decoration indicates the initial value of a variable or parameter which is both imported and exported. A function may not have an explicit side effect on a program variable and so cannot be regarded as exporting suhc a variable. For modelling purposes a read of an external (stream) variable is regarded as having a side effect (outside the SPARK boundary). Since it may be necessary to refer to the initial value of the external variable, before this implicit side effect occurs, the use of tilde is allowed only for external variables of mode IN which are globally referenced by function.

*** Semantic Error :318: Tilde or Percent may only be applied to variables.
The tilde decoration indicates the initial value of a variable or parameter which is both imported and exported. Percent indicates the value of a variable on entry to a for loop; this message occurs if either operator is applied to any other object.

*** Semantic Error :319: Tilde may only be applied to a variable which is both imported and exported.
The tilde decoration indicates the initial value of a variable or parameter which is both imported and exported; this message occurs if the variable concerned is either exported only or imported only in which case no distinction between its initial and final value is required.

*** Semantic Error :320: Tilde or Percent may only be applied to an entire variable.
Tilde (and %) may not be applied to an element of an array or field of a record. e.g. to indicate the initial value of the Ith element of array V use V~(I) not V(I)~.

*** Semantic Error :321: Tilde may not appear in pre-conditions.
Since it does not make sense to refer to anything other than the initial value of a variable in a pre-condition there is no need to use tilde to distinguish initial from final values.

*** Semantic Error :322: Only imports may be referenced in pre-conditions or return expressions.
Pre-conditions are concerned with the initial values of information carried into a subprogram. Since only imports can do this only imports can appear in pre-condition expressions.

*** Semantic Error :323: Updates may only be applied to records or arrays.
The extended SPARK update syntax is only used to express changes to components of a structured variable.

*** Semantic Error :324: Only one field name may appear here.
When using the extended SPARK update syntax for a record, you can not update more than one element in each clause of the update. For example, you cannot use [x,y =&gt; z], you must instead use [x =&gt; z; y =&gt; z].

*** Semantic Error :325: Type XXX has not been declared.
Occurs if a type is "announced" as part of an own variable clause and the end of the package is reached without an Ada declaration for a type of this name being found.

*** Semantic Error :326: Predicate is not boolean.
Occurs anywhere where a proof context is found not to be a boolean expression.

*** Semantic Error :327: XXX is a global variable which may not be updated in a function subprogram.

*** Semantic Error :328: The identifier XXX (exported by called subprogram) may not be updated in a function subprogram.
Occurs if a function calls a procedure which exports a global variable; this would create an illegal side-effect of the function.

*** Semantic Error :329: Illegal function call.
Issued if a call is made to a user-defined subprogram in a package initialization part.

*** Semantic Error :330: Illegal use of an own variable not of this package.
Issued if an attempt is made, in a package initialization part, to update an own variable of a non-enclosing package.

*** Semantic Error :331: Private types may not be unconstrained arrays.

*** Semantic Error :332: This private type was not declared as limited.
Issued where the type contains a component which is a limited private type, but where the declaration of this type in the visible part of the package does not specify that the type is limited.

*** Semantic Error :333: Initialization of XXX is not announced in the initialization clause of this package.
Issued when an own variable is initialized either by assignment or by having a pragma Import attached to it when initialization of the variable is not announced in its package's own variable initialization specification.

*** Semantic Error :334: Identifier XXX is not the name of a function.

*** Semantic Error :335: This annotation should be placed with the declaration of function XXX.
Issued if a function is declared in a package specification without an annotation but one is then supplied on the function body.

*** Semantic Error :336: Unexpected annotation - all annotations required for function XXX have already occurred.

*** Semantic Error :337: Package XXX may not be used as a prefix here.
Selected component notation may not be used in places where an item is directly visible.

*** Semantic Error :338: Scalar parameter XXX is of mode in out and must appear as an import.
Parameters passed as mode in out must be listed as imports in the subprogram's dependency relation if they are of scalar types. The rule also applies to a parameter of a private type if its full declaration is scalar.

*** Semantic Error :339: Subprogram XXX was not declared in package YYY.

*** Semantic Error :340: Only operators may be renamed in package specifications.
User-declared subprograms may not be renamed in package specifications although the implicitly declared function subprograms associated with operators may be.

*** Semantic Error :341: A range may not appear here.
Issued if a range is found where a single value is expected, for example, if an array slice is constructed.

*** Semantic Error :342: This proof annotation should be placed with the declaration of subprogram XXX.

Like global and derives annotations, proof annotations should be placed on the first appearance of a subprogram. There may also be a requirement for a second proof annotation on a subprogram body where it references an abstract own variable.

*** Semantic Error :343: Unexpected proof annotation - all annotations required for subprogram XXX have already occurred.
Issued if a second proof annotation for a subprogram is found but the subprogram does not reference any abstract own variables. A second annotation is only required where it is necessary to express both an abstract (external) and a refined (internal) view of an operation.

*** Semantic Error :399: Range error in annotation expression.
Issued if a proof annotation contains an expression that would cause a constraint error if it were in an executable Ada statement. For example: "--# post X = T'Succ(T'Last);" VCs generated from such malformed predicates would always be unprovable.

*** Semantic Error :400: Expression contains division by zero.
Issued when a static expression, evaluated using perfect arithmetic, is found to contain a division by zero.

*** Semantic Error :401: Illegal numeric literal.
Issued when a numeric literal is illegal because it contains, for example, digits not compatible with its number base.

*** Semantic Error :402: Constraint_Error will be raised here.
Issued whenever a static expression would cause a constraint error. e.g. assigning a value to a constant outside the constant's type range. In SPARK a static expression may not yield a value which violates a range constraint.

*** Semantic Error :403: Argument value is inconsistent with the number of dimensions of array type XXX.
Issued when an array attribute containing an argument is found and the value of the argument is inconsistent with the number of dimensions of the array type to which it is being applied.

*** Semantic Error :406: Only scalar or non-tagged record subtypes may be declared without a constraint.
Issued if a subtype declaration of the form subtype S is T is used where T is not a scalar or non-tagged record type. Secondly, T must not be private at the point of this declaration.

*** Semantic Error :407: This choice overlaps a previous one.
Choices in case statements and array aggregates may not overlap.

*** Semantic Error :408: Case statement is incomplete.
A case statement must either explicitly supply choices to cover the whole range of the (sub)type of the controlling expression, or it must supply an others choice.

*** Semantic Error :409: Empty range specified.
In SPARK, no static range is permitted to be null.

*** Semantic Error :410: Choice out of range.
The choices in case statements and array aggregates must be within the constraints of the appropriate (sub)type.

*** Semantic Error :411: Others clause required.
Issued where an others clause is required to satisfy the Ada language rules.

*** Semantic Error :412: Explicit boolean range not permitted.

*** Semantic Error :413: Invalid range constraint.

Issued where a range constraint is outside the range of the (sub)type to which the constraint applies.

*** Semantic Error :414: Array aggregate is incomplete.
An array aggregate must either explicitly supply values for all array elements or provide an others clause.

*** Semantic Error :415: Too many entries in array aggregate.
Issued where an array aggregate using positional association contains more entries than required by the array index type.

*** Semantic Error :416: Type may not have an empty range.

*** Semantic Error :417: String subtypes must have a lower index bound of 1.

*** Semantic Error :418: Index upper and/or lower bounds do not match those expected.
Issued where assignment, association or type conversion is attempted between two different constrained subtypes of the same unconstrained array type, and where the index bounds do not match.

*** Semantic Error :420: Array index(es) not convertible.
Issued when an attempt is made to convert between two arrays whose indexes are neither of the same type nor numeric.

*** Semantic Error :421: Array components are not of the expected type.
Issued when a type conversion attempts to convert between two array types whose components are of different types.

*** Semantic Error :422: Array component constraints do not match those expected.
Issued when a type conversion attempts to convert between two array types whose components are of the same type but do not have constraints which can be statically determined to be identical.

*** Semantic Error :423: Array has different number of dimensions from that expected.
Issued when attempting to convert between two array types which have different numbers of dimensions.

*** Semantic Error :425: String literals may not be converted.
Issued if the argument of a type conversion is a string literal. A common cause is an attempt to type qualify a string and accidentally omitting the tick character.

*** Semantic Error :500: Mode expected.
Issued when performing data flow analysis only where a subprogram has no dependency clause and its global variables have not been given modes in the global annotation.

*** Semantic Error :501: Dependency relation expected.
A dependency relation is required for each procedure if information flow analysis is to be performed.

*** Semantic Error :502: Exportation of XXX is incompatible with its global mode.
Issued when a procedure has both a global annotation with modes and a dependency relation, and a global of mode in is listed as an export in the dependency relation.

*** Semantic Error :503: Importation of XXX is incompatible with its global mode.
Issued when a procedure has both a global annotation with modes and a dependency relation, and a global of mode out is listed as an import in the dependency relation.

*** Semantic Error :504: Parameter XXX is of mode in out and must appear as an import.

*** Semantic Error :505: Global variable XXX is of mode in out and must appear as an import.

Issued where a procedure has both a global annotation with modes and a dependency relation, and a global variable of mode in out is not listed as an import in the dependency relation.

*** Semantic Error :506: Parameter XXX is of mode in out and must appear as an export.

*** Semantic Error :507: Global variable XXX is of mode in out and must appear as an export.
Issued where a procedure has both a global annotation with modes and a dependency relation, and a global variable of mode in out is not listed as an export in the dependency relation.

*** Semantic Error :508: This global variable is a parameter of mode in and can only have the global mode in.

*** Semantic Error :550: use type clauses may only be used in SPARK95: clause ignored.

*** Semantic Error :551: All operators for type XXX are already visible.

*** Semantic Error :552: The type XXX already appears in the use type clause.

*** Semantic Error :554: XXX is a limited private type for which no operators can be made visible.

*** Semantic Error :555: XXX is not mentioned in an earlier with clause of this compilation unit.

*** Semantic Error :600: pragma Import has a minimum of 2 and a maximum of 4 parameters.

*** Semantic Error :601: Convention, Entity, External_Name or Link_Name expected.

*** Semantic Error :602: An association for XXX has already been given.

*** Semantic Error :603: No association for XXX was given.

*** Semantic Error :604: This package may not have a body - consider use of pragma Elaborate_Body.
In Ada 95, a package body is illegal unless it is required for the purpose of providing a subprogram body, or unless this pragma is used. This error is issued where a package body is found for a package whose specification does not require a body.

*** Semantic Error :605: pragma Elaborate_Body has one parameter.

*** Semantic Error :606: This expression does not represent the expected package name XXX.
Issued when the parameter to a pragma Elaborate_Body is invalid.

*** Semantic Error :607: This package requires a body and must therefore include either pragma Elaborate_Body or a subprogram declaration.
Issued where a package specification contains no subprogram declarations, but whose own variables (as specified in the package annotation) are not all declared (and initialized where appropriate) in the package specification. This is because such a package is not allowed a body in Ada 95 unless either the pragma is given or a subprogram declared.

*** Semantic Error :608: Reduced accuracy subtypes of real numbers are considered obsolescent and are not supported by SPARK.

*** Semantic Error :609: This entity cannot be assigned to.

*** Semantic Error :610: Child packages may not be used in SPARK83.

*** Semantic Error :611: Illegal use of deferred constant prior to its full declaration.

*** Semantic Error :613: Illegal name for body stub.

Issued if a dotted name appears in a body stub as in "package body P.Q is separate". No legal stub could ever have such a name.

*** Semantic Error :614: Child packages may be declared only at library level.
Issued if an attempt is made to declare a child package which is embedded in a package or subprogram.

*** Semantic Error :615: Name does not match name of package.
Issued if the closing identifier of a package has a different number of identifiers from the name originally given for the package. For example "package P.Q is … end P.Q.R;".

*** Semantic Error :616: The private package XXX is not visible at this point.
Issued if an attempt is made to with or inherit a private package from the visible part of a public package.

*** Semantic Error :617: Public sibling XXX is not visible at this point.
Arises from attempting to inherit a public sibling child package from a private child package.

*** Semantic Error :618: The owner of the current package does not inherit the package XXX.
A private child package can only inherit a remote package if its parent also inherits it; this is a analogous to the behaviour of embedded packages which may also only inherit a remote package if their enclosing package also does so.

*** Semantic Error :619: The package XXX is not owned by the current package.
This message indicates an attempt to claim that own variables of a package other than a private child package of the current package are refinement constituents of an abstract own variable of the current package.

*** Semantic Error :620: Own variables here must be refinement constituents in package owner XXX.
Own variables of private child packages must appear as refinement constituents of the package which owns the child. If the Examiner has seen the owner package body before processing the child and has not found the required refinement constituent then this message results on processing the child.

*** Semantic Error :621: Own variable XXX expected as a refinement constituent in this package.
Own variables of private child packages must appear as refinement constituents of the package which owns the child. If the Examiner has seen a child package which declares an own variable before examining its owner&#146;s body then this message is issued if the owner lacks the required refinement constituent declaration.

*** Semantic Error :622: Own variable XXX did not occur in an initialization specification.
Issued if an own variable appears in an initialization clause and is also a refinement constituent of an own variable which is not marked as initialized.

*** Semantic Error :623: Own variable XXX occurred in an initialization specification.
Issued if an own variable does not appear in an initialization clause and is also a refinement constituent of an own variable that is marked as initialized.

*** Semantic Error :624: All operators from ancestor package XXX are already visible.
A package must appear in a with clause before types declared in it can be specified in a use type clause.

*** Semantic Error :630: XXX is not the name of generic subprogram.
Only generic subprogram can be instantiated.

*** Semantic Error :631: Generic function found where a generic procedure was expected.
Subprogram kind of generic and its instantiation must match.

*** Semantic Error :632: Generic procedure found where a generic function was expected.
Subprogram kind of generic and its instantiation must match.

*** Semantic Error :633: Generic actual part expected, generic unit XXX has generic formal parameters.
The number of generic formal and actual parameters must match exactly.

*** Semantic Error :634: Unexpected generic actual part, generic unit XXX does not have any generic formal parameters.
The number of generic formal and actual parameters must match exactly.

*** Semantic Error :635: Incorrect number of generic actual parameters for instantiation of genric unit XXX.
The number of generic formal and actual parameters must match exactly.

*** Semantic Error :636: Type XXX is not compatible with generic formal parameter YYY.
See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :637: User-defined generic units are not permitted in SPARK 83.
There are weaknesses in the generic type model of Ada 83 that prevent the implementation of a safe subset of generics in SPARK 83. These deficiences are overcome in Ada 95. SPARK 83 users may employ the predefined unit Unchecked_Conversion only.

*** Semantic Error :638: Unexpected global annotation. A generic subprogram may not reference or update global variables.
A standalone generic subprogram may not have a global annotation. Note that a subprogram in a generic package may have a global annotation as long as it only refers to own variables that are local to the package.

*** Semantic Error :639: A generic formal object may only have default mode or mode in.
SPARK restricts formal objects to being constants in order to avoid concealed information flows.

*** Semantic Error :640: A generic formal object may only be instantiated with a constant expression.
SPARK restricts formal objects to being constants in order to avoid concealed information flows.

*** Semantic Error :641: There is no generic subprogram declaration named XXX so a generic body of that name cannot be declared here.
A generic body must be preceded by a generic declaration of the same name.

*** Semantic Error :642: XXX is not the name of a generic subprogram so a generic body cannot be declared here.
A generic body must be preceded by a generic declaration of the same name.

*** Semantic Error :643: XXX is a generic function so a generic procedure body cannot be declared here.
A generic body must be preceded by a generic declaration of the same name and kind.

*** Semantic Error :644: XXX is a generic procedure so a generic function body cannot be declared here.
A generic body must be preceded by a generic declaration of the same name and kind.

*** Semantic Error :645: Actual array element XXX is not compatible with the element type YYY of the generic formal parameter.

See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :646: Actual array index XXX is not compatible with the index type YYY of the generic formal parameter.
See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :647: Actual array XXX has more dimensions than formal arrayYYY.
See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :648: Actual array XXX has fewer dimensions than formal array YYY.
See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :649: Actual array XXX is constrained but the associated formal YYY is unconstrained.
See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :650: Actual array XXX is unconstrained but the associated formal YYY is constrained.
See ALRM 12.5. Each generic formal type parameter must be supplied with an actual type which is of a compatible class. Note that SPARK does not have default values for such associations.

*** Semantic Error :700: Mode 'in out' may not be applied to own variables or their refinement constituents.
Own variables may be given a mode to indicate that they are system level inputs or outputs (i.e. they obtain values from or pass values to the external environment). Since effective SPARK design strictly separates inputs from outputs the mode 'in out' is not permitted.

*** Semantic Error :701: The mode of this refinement constituent is not consistent with its subject: XXX.
If an abstract own variable is given a mode then its refinement constituents must all be of the same mode.

*** Semantic Error :702: Own variable XXX must be given the mode 'in' to match its earlier announcement .
Issued if an own variable of an embedded package is not given the same mode as the earlier refinement constituent that announced it would exist.

*** Semantic Error :703: Own variable XXX must be given the mode 'out' to match its earlier announcement .
Issued if an own variable of an embedded package is not given the same mode as the earlier refinement constituent that announced it would exist.

*** Semantic Error :704: Own variable XXX may not have a mode because one was not present in its earlier announcement .
Issued if an own variable of an embedded package is given a mode when the earlier refinement constituent that announced it would exist did not have one.

*** Semantic Error :705: Refinement constituent XXX must be given the mode 'in' to match the child package own variable with which it is being associated.
If a refinement constituent is an own variable of a private package then the constituent must have the same mode as the own variable to which it refers.

*** Semantic Error :706: Refinement constituent XXX must be given the mode 'out' to match the child package own variable with which it is being associated.
If a refinement constituent is an own variable of a private package then the constituent must have the same mode as the own variable to which it refers.

*** Semantic Error :707: Refinement constituent XXX may not have a mode because one was not present on the child package own variable with which it is being associated.
If a refinement constituent is an own variable of a private package then the constituent can only be given a mode if the own variable to which it refers has one.

*** Semantic Error :708: Own variable XXX has a mode and may not appear in an initializes clause.
Mode own variables (stream variables) are implicitly initialized by the environment to which they are connected and may not appear in initializes clauses since this would require their explicit initialization.

*** Semantic Error :709: Own variable or constituent XXX has mode 'out' and may not be referenced by a function.
Functions are permitted to reference own variables that are either unmoded or of mode 'in'. Since mode 'out' own variables represent outputs to the environment, reading them in a function does not make sense and is not allowed.

*** Semantic Error :710: The own variable or constituent XXX is of mode 'in' and can only have global mode 'in'.
Global modes, if given, must be consistent with the modes of own variables that appear in the global list.

*** Semantic Error :711: The own variable or constituent XXX is of mode 'out' and can only have global mode 'out'.
Global modes, if given, must be consistent with the modes of own variables that appear in the global list.

*** Semantic Error :712: The own variable or constituent XXX is of either mode 'in' or mode 'out' and may not have global mode 'in out'.
Global modes, if given, must be consistent with the modes of own variables that appear in the global list.

*** Semantic Error :713: The own variable or constituent XXX is of mode 'in' and may not appear in a dependency clause as an export.
Own variables with mode 'in' denote system-level inputs; their exportation is not allowed.

*** Semantic Error :714: The own variable or constituent XXX is of mode 'out' and may not appear in a dependency clause as an import.
Own variables with mode 'out' denote system-level outputs; their importation is not allowed.

*** Semantic Error :715: Function XXX references external (stream) variables and may only appear directly in an assignment or return statement.
To avoid ordering effects, functions which globally access own variables which have modes (indicating that they are connected to the external environment) may only appear directly in assignment or return statements. They may not appear as actual parameters or in any other form of expression.

*** Semantic Error :716: External (stream) variable XXX may only appear directly in an assignment or return statement; or as an actual parameter to an unchecked conversion.
To avoid ordering effects, own variables which have modes (indicating that they are connected to the external environment) may only appear directly in assignment or return statements. They may not appear as actual parameters (other than to instantiations of Unchecked_Conversion) or in any other form of expression.

*** Semantic Error :717: External (stream) variable XXX is of mode 'in' and may not be assigned to.
Own variables with mode 'in' represent inputs to the system from the external environment. As such, assigning to them does not make sense and is not permitted.

*** Semantic Error :718: External (stream) variable XXX is of mode 'out' and may not be referenced.
Own variables with mode 'out' represent outputs to the external environment from the system. As such, referencing them does not make sense and is not permitted.

*** Semantic Error :719: External (stream) variables may not be referenced or updated during package elaboration.
Own variables with modes represent inputs and outputs between the external environment and the system. Referencing or updating them during package elaboration would introduce ordering effects and is not permitted.

*** Semantic Error :720: Variable XXX is an external (stream) variable and may not be initialized at declaration.
Own variables with modes represent inputs and outputs between the external environment and the system. Referencing or updating them during package elaboration would introduce ordering effects and is not permitted.

*** Semantic Error :721: This refined function global annotation may not reference XXX because it is an external (stream) variable whose abstract subject YYY does not have a mode.
Functions may be used to reference external (stream) variables and the Examiner generates the appropriate information flow to show that the value returned by the function is 'volatile'. If the abstract view of the same function shows it referencing an own variable which is not an external stream then the volatility of the function is concealed. The error can be removed either by making the abstract own variable a mode 'in' stream or by using a procedure instead of a function to read the refined stream variable.

*** Semantic Error :722: The mode on abstract global variable YYY must be made 'in out' to make it consistent with the referencing of mode 'in' external (stream) constituent XXX in the refined global annotation.
Where a procedure references an external (stream) variable of mode 'in' the Examiner constructs appropriate information flow to show that the input stream is 'volatile'. If the abstract view shows that the procedure obtains its result by simply reading an own variable which is not an external stream then the volatility is concealed. The error can be removed either by making the global mode of XXX 'in out' or making XXX an external (stream) variable of mode 'in'.

*** Semantic Error :723: Variable XXX must appear in this refined global annotation.
Issued when a global variable which is present in the first (abstract) global annotation is omitted from the second (refined) one.

*** Semantic Error :724: Exit label must match the label of the most closely enclosing loop statement.
If an exit statement names a loop label, then the most closely enclosing loop statement must have a matching label.

*** Semantic Error :725: Protected function or variable XXX may only appear directly in an assignment or return statement.
To avoid ordering effects, protected functions may only appear directly in assignment or return statements. They may not appear as actual parameters or in any other form of expression. Ordering effects occur because the global state referenced by the protected function may be updated by another process during expression evaluation.

*** Semantic Error :730: A loop with no iteration scheme or exit statements may only appear as the last statement in the outermost scope of the main subprogram (or a task body when using the Ravenscar profile).
If a loop has neither an iteration scheme nor any exit statements then it will run forever. Any statements following it will be unreachable. SPARK only allows one such loop which must be the last statement of the main program.

*** Semantic Error :750: The identifier YYY.XXX is either undeclared or not visible at this point. An array type may not be used as its own index type.
The type mark used for the index of an array type declaration must not be the same as the name of the array type being declared.

*** Semantic Error :751: The identifier YYY.XXX is either undeclared or not visible at this point. A record type may not include fields of its own type.
The type mark given for a field in a record type declaration must not be the same as the name of the record type being declared.

*** Semantic Error :752: The identifier YYY.XXX is either undeclared or not visible at this point. This identifier must appear in a preceding legal global annotation or formal parameter list.
For an identifier to appear legally as an import in a derives annotation, it must be a formal parameter or must appear legally in a preceding global annotation and must be of mode 'in' or mode 'in out'.

*** Semantic Error :753: The identifier YYY.XXX is either undeclared or not visible at this point. This identifier must appear in a preceding legal global annotation or formal parameter list.
For an identifier to appear legally as an export in a derives annotation, it must be a formal parameter or must appear legally in a preceding global annotation and must be of mode 'out' or mode 'in out'.

*** Semantic Error :754: The identifier YYY.XXX is either undeclared or not visible at this point. This package must be both inherited and withed to be visible here.
For a package name to be visible in Ada context, it must appear in both the inherit clause and the with clause of the enclosing package.

*** Semantic Error :755: The identifier YYY.XXX is either undeclared or not visible at this point. A parent of a child package must be inherited to be visible here.
A parent of a child package must be inherited (but not withed) to be visible in that child.

*** Semantic Error :770: If Any_Priority is defined, Priority and Interrupt_Priority must also be defined.
If the type Any_Priority is defined in package System, then the subtypes Priority and Interrupt_Priority must also be defined; if support for tasking is not required, then the definition of Any_Priority may be removed.

*** Semantic Error :771: The parent type of this subtype must be Any_Priority.
Ada 95 requires that both Priority and Interrupt_Priority be immediate subtypes of Any_Priority.

*** Semantic Error :772: The range of Priority must contain at least 30 values; LRM D.1(26).

Ada 95 requires that the range of the subtype Priority include at least 30 values; this requirement is stated in the Ada 95 Language Reference Manual at D.1(26).

*** Semantic Error :773: Priority'First must equal Any_Priority'First; LRM D.1(10).
Ada 95 requires that task priority types meet the following criteria, the second of which is relevant to this

- *subtype Any_Priority is Integer range implementation-defined;*
- *subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;*
- *subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last.*

*** Semantic Error :774: Interrupt_Priority'First must equal Priority'Last + 1; LRM D.1(10).
Ada 95 requires that task priority types meet the following criteria, the third of which is relevant to this

- *subtype Any_Priority is Integer range implementation-defined;*
- *subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;*
- *subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last.*

*** Semantic Error :775: Interrupt_Priority'Last must equal Any_Priority'Last; LRM D.1(10).
Ada 95 requires that task priority types meet the following criteria, the third of which is relevant to this

- *subtype Any_Priority is Integer range implementation-defined;*
- *subtype Priority is Any_Priority range Any_Priority'First .. implementation-defined;*
- *subtype Interrupt_Priority is Any_Priority range Priority'Last+1 .. Any_Priority'Last.*

*** Semantic Error :776: In SPARK95 mode, only packages Standard, System, Ada.Real_Time and Ada.Interrupts may be specified in the config file.
In SPARK95 mode, the packages that may be specified in the target configuration file are: Standard, System, Ada.Real_Time and Ada.Interrupts. The latter two are ignored unless the Ravenscar profile is selected.

*** Semantic Error :777: In package System, Priority must be an immediate subtype of Integer.
Ada 95, and hence SPARK95, defines Priority as being an immediate subtype of Integer.

*** Semantic Error :778: This identifier is not valid at this point in the target configuration file.
The specified identifier cannot be used here; it is most probably either not valid in the target configuration file at all, or might be valid in a different package, but not here.

*** Semantic Error :779: Definition of this package in the target configuration file is not allowed in SPARK83 mode.
In SPARK83 mode, only package Standard may be specified in the target configuration file.

*** Semantic Error :780: Type XXX must be private.
This type may only be declared as private in the target configuration file.

*** Semantic Error :781: The lower bound of a signed integer type declaration must be greater than or equal to System.Min_Int.
This error can only be generated in SPARK95 mode when the configuration file specifies a value for System.Min_Int.

*** Semantic Error :782: The upper bound of a signed integer type declaration must be less than or equal to System.Max_Int.
This error can only be generated in SPARK95 mode when the configuration file specifies a value for System.Max_Int.

*** Semantic Error :783: Modulus must be less than or equal to System.Max_Binary_Modulus.
This error can only be generated in SPARK95 mode when the configuration file specifies a value for System.Max_Binary_Modulus.

*** Semantic Error :784: System.Max_Binary_Modulus must be a positive power of 2.

*** Semantic Error :785: The number of digits specified exceeds the value defined for System.Max_Digits.
The maximum decimal precision for a floating point type, where a range specification has not been included, is defined by System.Max_Digits.

*** Semantic Error :786: The number of digits specified exceeds the value defined for System.Max_Base_Digits.
The maximum decimal precision for a floating point type, where a range specification has been included, is defined by System.Max_Base_Digits.

*** Semantic Error :787: Digits value must be positive.

*** Semantic Error :788: Delta value must be positive.

*** Semantic Error :789: The only currently supported type attribute in this context is 'Base.

*** Semantic Error :790: A base type assertion requires a type here.

*** Semantic Error :791: The base type in this assertion must be a predefined type.
Predefined types are those defined either by the language, or in package Standard, using the configuration file mechanism.

*** *Semantic Error :792: The types in this assertion must both be either floating point or signed integer.*

*** *Semantic Error :793: The base type in a base type assertion must have a defined range.*
The ranges for the predefined types will generally be specified in the target configuration file.

*** Semantic Error :794: The range of the base type in a base type assertion must be at least as large as the type which is the subject of the assertion.

*** Semantic Error :795: A base type assertion must be in the same declarative region as that of the full type definition.

*** Semantic Error :796: This type already has a base type defined: either it already has a base type assertion, or it is a predefined type.
If the type which is the subject of the assertion is not a derived type, i.e. it is its own base type, then this error is raised.

*** Semantic Error :797: The base type in a floating point base type assertion must have a defined accuracy.

*** Semantic Error :798: The accuracy of the base type in a base type assertion must be at least that of the type which is the subject of the assertion.

*** Semantic Error :800: Modulus must be a positive power of 2.
In SPARK, modular types must have a modulus which is a positive power of 2.

*** Semantic Error :801: Modular types may only be used in SPARK95.
Ada83 (and hence SPARK83) does not include modular types.

*** Semantic Error :803: Unary arithmetic operators are not permitted for modular types.

Unary arithmetic operators are of little value. The "abs" and "+" operators have no effect for modular types, and so are not required. The unary minus operator is a source of potential confusion, and so is not permitted in SPARK.

*** Semantic Error :804: Universal expression may not be implicitly converted to a modular type here. Left hand operand requires qualification to type XXX.
A universal expression cannot be used as the left hand operand of a binary operator if the right hand operand is of a modular type. Qualification of the left hand expression is required in this case.

*** Semantic Error :805: Universal expression may not be implicitly converted to a modular type here. Right hand operand requires qualification to type XXX.
A universal expression cannot be used as the right hand operand of a binary operator if the left hand operand is of a modular type. Qualification of the right hand expression is required in this case.

*** Semantic Error :811: Unnecessary others clause - case statement is already complete.

*** Semantic Error :820: Abstract types are not currently permitted in SPARK.
Only non-abstract tagged types are currently supported. It is hoped to lift this restriction in a future Examiner release.

*** Semantic Error :821: This type declaration must be a tagged record because it's private type is tagged.
If a type is declared as "tagged private" then its full declaration must be a tagged record.

*** Semantic Error :822: XXX is not a tagged type; only tagged types may be extended.
In SPARK, "new" can only be used to declare a type extension; other derived types are not permitted.

*** Semantic Error :823: This type may not be extended in the same package in which it is declared.
SPARK only permits types from another library package to be extended. This rule prevents overloading of inherited operations.

*** Semantic Error :824: This package already extends a type from package XXX. Only one type extension per package is permitted.
SPARK only permits one type extension per package. This rule prevents overloading of inherited operations.

*** Semantic Error :825: Type XXX expected in order to complete earlier private extension.
Since SPARK only permits one type extension per package it follows that the declaration "new XXX with private" in a package visible part must be paired with "new XXX with record..." in its private part. The ancestor type XXX must be the same in both declarations.

*** Semantic Error :826: Type extension is not permitted in SPARK 83.
Type extension is an Ada 95 feature not included in Ada or SPARK 83.

*** Semantic Error :827: The actual parameter associated with a tagged formal parameter in an inherited operation must be an object not an expression.
There are several reasons for this SPARK rule. Firstly, Ada requires tagged parameters to be passed by reference and so an object must exist at least implicitly. Secondly, in order to perform flow analysis of inherited subprogram calls, the Examiner needs identify what subset of the information available at the point of call is passed to and from the called subprogram. Since information can only flow through objects it follows that actual parameter must be an object.

*** Semantic Error :828: Tagged types and tagged type extensions may only be declared in library-level package specifications.

This SPARK rule facilitates the main uses of tagged types while greatly simplifying visibility rules.

*** Semantic Error :829: Illegal re-declaration: this subprogram shares the same name as the inheritable root operation XXX but does not override it.
To avoid overloading, SPARK prohibits more than one potentially visible subprogram having the same name.

*** Semantic Error :830: A private type may not be implemented as a tagged type or an extension of a tagged type.
This rule means that a private type can only be implemented as a tagged type if the private type itself is tagged.

*** Semantic Error :831: Extended tagged types may only be converted in the direction of their root type.
This is an Ada rule: type conversions simply omit unused fields of the extended type. It follows that conversions must be in the direction of the root type.

*** Semantic Error :832: Only tagged objects, not expressions, may be converted.
For flow analysis purposes the Examiner needs to know what subset of the information in the unconverted view is available in the converted view. Since information can only flow through objects it follows that only objects can be converted.

*** Semantic Error :833: Invalid record aggregate: type XXX has a private ancestor.
If an extended type has a private ancestor then an extension aggregate must be used rather than a normal aggregate.

*** Semantic Error :834: Null records are only permitted if they are tagged.
An empty record can have no use in a SPARK program others than as a root type from which other types can be derived and extended. For this reason, null records are only allowed if they are tagged.

*** Semantic Error :835: XXX is not an extended tagged record type.
An extension aggregate is only appropriate if the record type it is defining is an extended record. A normal aggregate should be used for other record (and array) types.

*** Semantic Error :836: This expression does not represent a valid ancestor type of the aggregate XXX.
The expression before the reserved word "with" must be of an ancestor type of the overall aggregate type. In SPARK, the ancestor expression may not be a subtype mark.

*** Semantic Error :837: Invalid record aggregate: there is a private ancestor between the type of this expression and the type of the aggregate XXX.
The ancestor type can be an tagged type with a private extension; however, there must be no private extensions between the ancestor type and the type of the aggregate.

*** Semantic Error :838: Incomplete aggregate: null record cannot be used here because fields in XXX require values.
The aggregate form "with null record" can only be used if the type of the aggregate is a null record extension of the ancestor type. If any fields are added between the ancestor type and the aggregate type then values need to be supplied for them so "null record" is inappropriate.

*** Semantic Error :839: This package already contains a root tagged type or tagged type extension. Only one such declaration per package is permitted.
SPARK permits one root tagged type or one tagged type extension per package, but not both. This rule prevents the declaration of illegal operations with more than one controlling parameter.

*** Semantic Error :840: A tagged or extended type may not appear here. SPARK does not permit the declaration of primitive functions with controlling results.
A primitive function controlled by its return result would be almost unusable in SPARK because a data flow error would occur wherever it was used.

*** Semantic Error :841: The return type in the declaration of this function contained an error. It is not possible to check the validity of this return type.
Issued when there is an error in the return type on a function's initial declaration. In this situation we cannot be sure what return type is expected in the function's body. It would be misleading to simply report a type mismatch since the types might match perfectly and both be wrong. Instead, the Examiner reports the above error and refuses to analyse the function body until its specification is corrected.

*** Semantic Error :842: Pragma Atomic_Components is not permitted in SPARK when the Ravenscar profile is selected.

*** Semantic Error :843: Pragma Volatile_Components is not permitted in SPARK when the Ravenscar profile is selected.

*** Semantic Error :850: This construct may only be used when the Ravenscar profile is selected.
Support for concurrent features of the SPARK language, including protected objects, tasking, etc. are only supported when the Ravenscar profile is selected.

*** Semantic Error :851: The parameter to pragma Atomic must be a simple_name.
The parameter to pragma Atomic must be a simple_name; and may not be passed using a named association.

*** Semantic Error :852: pragma Atomic may only appear in the same immediate scope as the type to which it applies.
This is an Ada rule (pragma Atomic takes a local name see LRM 13.1(1)). Note that this precludes the use of pragma Atomic on a predefined type.

*** Semantic Error :853: pragma Atomic may only apply to a scalar base type.
pragma Atomic may only be applied to base types that are scalar. i.e. enumeration types, integer types, real types, modular types. The type must not be private.

*** Semantic Error :854: pragma Atomic takes exactly one parameter.

*** Semantic Error :855: The type of own variable XXX is not consistent with its modifier.
An own variable with a task modifier must be of a task type. A task own variable must have the task modifier. An own variable with a protected modifier must be a protected object, suspension object or pragma atomic type. A protected or suspension object own variable must have the protected modifier.

*** Semantic Error :858: A variable that appears in a protects property list may not appear in a refinement clause.
A variable in a protects list is effectively protected and hence cannot be refined.

*** Semantic Error :859: A protected own variable may not appear in a refinement clause.
Protected state cannot be refined or be used as refinement constituents.

*** Semantic Error :860: Own variable XXX appears in a protects list and hence must appear in the initializes clause.
Protected state (including all refinement constituents) must be initialized.

*** Semantic Error :863: Own variable XXX is protected and may not appear in an initializes clause.

Protected own variables must always be initialized, and should not appear in initializes annotations.

*** Semantic Error :864: Unexpected initialization specification - all own variables of this package are either implicitly initialized, or do not require initialization.
An own variable initialization clause and that of its refinement constituents must be consistent.

*** Semantic Error :865: Field XXX is part of the ancestor part of this aggregate and does not require a value here.
An extension aggregate must supply values for all fields that are part of the overall aggregate type but not those which are part of the ancestor part.

*** Semantic Error :866: The expression in a delay_until statement must be of type Ada.Real_Time.Time.
When the Ravenscar Profile is selected, the delay until statement may be used. The argument of this statement must be of type Ada.Real_Time.Time.

*** Semantic Error :867: Subprogram XXX contains a delay statement but does not have a delay property.
Any subprogram that may call delay until must have a delay property in a declare annotation. Your subprogram is directly or indirectly making a call to delay until.

*** Semantic Error :868: Protected object XXX may only be declared immediately within a library package.
This error message is issued if a type mark representing a protected type appears anywhere other than in a library level variable declaration or library-level own variable type announcement.

*** Semantic Error :869: Protected type XXX already contains an Entry declaration; only one Entry is permitted.
The Ravenscar profile prohibits a protected type from declaring more than one entry.

*** Semantic Error :870: Protected type XXX does not have any operations, at least one operation must be declared.
A protected type which provides no operations can never be used so SPARK requires the declaration of at least one.

*** Semantic Error :872: Variable XXX is not protected; only protected items may be globally accessed by protected operations.
In order to avoid the possibility of shared data corruption, SPARK prohibits protected operations from accessing unprotected data items.

*** Semantic Error :873: This subprogram requires a global annotation which references the protected type name XXX.
In order to statically-detect certain bounded errors defined by the Ravenscar profile, SPARK requires every visible operation of protected type to globally reference the abstract state of the type.

*** Semantic Error :874: Protected state XXX must be initialized at declaration.
Because there is no guarantee that a concurrent thread that initializes a protected object will be executed before one that reads it, the only way we can be sure that a protected object is properly initialized is to do so at the point of declaration. You have either declared some protected state and not included an initialisation or you have tried to initialise some protected state in package body elaboration.

*** Semantic Error :875: Protected type expected; access discriminants may only refer to protected types in SPARK.

Access discriminants have been allowed in SPARK solely to allow devices made up of co-operating Ravenscar-compliant units to be constructed. For this reason only protected types may appear in access discriminants.

*** Semantic Error :876: This protected type or task declaration must include either a pragma Priority or pragma Interrupt_Priority.
To allow the static detection of certain bounded errors defined by the Ravenscar profile, SPARK requires an explicitly-set priority for each protected type, task type or object of those types. The System.Default_Priority may used explicitly provided package System has been defined in the configuration file.

*** Semantic Error :877: Priority values require an argument which is an expression of type integer.

*** Semantic Error :878: This protected type declaration contains a pragma Attach_Handler and must therefore also include a pragma Interrupt_Priority.
To allow the static detetion of certain bounded errors defined by the Ravenscar profile, SPARK requires an explicitly-set priority for each protected type or object. The System.Default_Priority may used explicitly provided package System has been defined in the configuration file.

*** Semantic Error :879: Unexpected pragma XXX: this pragma may not appear here.
pragma Interrupt_Priority must be the first item in a protected type declaration or task type declaration; pragma Priority must be the first item in a protected type declaration, task type declaration or the main program.

*** Semantic Error :880: Pragma Priority or Interrupt_Priority expected here.
Issued when a pragma other than Priority or Interrupt_Priority appears as the first item in a protected type or task type declaration.

*** Semantic Error :881: The priority of XXX must be in the range YYY.
See LRM D.1(17).

*** Semantic Error :882: Integrity property requires an argument which is an expression of type Natural.

*** Semantic Error :883: Pragma Interrupt_Handler may not be used; SPARK does not support the dynamic attachment of interrupt handlers [LRM C3.1(9)].
Interrupt_Handler is of no use unless dynamic attachment of interrupt handlers is to be used.

*** Semantic Error :884: Pragma Attach_Handler is only permitted immediately after the corresponding protected procedure declaration in a protected type declaration.
Pragma Attach_Handler may only be used within a protected type declaration. Furthermore, it must immediately follow a protected procedure declaration with the same name as the first argument to the pragma.

*** Semantic Error :885: Pragma Attach_Handler may only be applied to a procedure with no parameters.
See LRM C.3.1(5).

*** Semantic Error :887: A discriminant may only appear alone, not in an expression.
Issued when a task or protected type priority is set using an expression involving a discriminant. The use of such an expression greatly complicates the static evaluation of the priority of task or protected subtypes thus preventing the static elimination of certain Ravenscar bounded errors.

*** Semantic Error :888: Unexpected Delay, XXX already has a Delay property.
A procedure may only have a maximum of one delay annotation.

*** Semantic Error :889: The own variable XXX must have the suspendable property.

The type used to declare this object must be a protected type with and entry or a suspension object type.

*** Semantic Error :890: The name XXX already appears in the suspends list.
Items may not appear more than once in an a suspends list.

*** Semantic Error :891: Task type or protected type required.
Issued in a subtype declaration where the constraint is a discriminant constraint. Only task and protected types may take a discriminant constraint as part of a subtype declaration.

*** Semantic Error :892: Array type, task type or protected type required.
Issued in a subtype declaration where the constraint is a either a discriminant constraint or an index constraint (these two forms cannot always be distinguished syntactically). Only task and protected types may take a discriminant constraint and only array types may take an index constraint as part of a subtype declaration.

*** Semantic Error :893: Number of discriminant constraints differs from number of known discriminants of type XXX.
Issued in a subtype declaration if too many or two few discriminant constraints are supplied.

*** Semantic Error :894: Only variables of a protected type may be aliased.
SPARK supports the keyword aliased in variable declarations only so that protected and task types can support access discriminants. Since it has no other purpose it may not be used except in a protected object declaration.

*** Semantic Error :895: Attribute Access may only be applied to variables which are declared as aliased, variable XXX is not aliased.
This is a slightly annoying Ada issue. Marking a variable as aliased prevents it being placed in a register which would make pointing at it hazardous; however, SPARK only permits 'Access on protected types which are limited and therefore always passed by reference anyway and immune from register optimization. Requiring aliased on protected objects that will appear in discriminant constraints is therefore unwanted syntactic sugar only.

*** Semantic Error :896: The task type XXX does not have an associated body.
Issued at the end of a package body if a task type declared in its specification contains neither a body nor a body stub for it.

*** Semantic Error :897: The protected type XXX does not have an associated body.
Issued at the end of a package body if a protected type declared in its specification contains neither a body nor a body stub for it.

*** Semantic Error :898: XXX is not a protected or task type which requires a body.
Issued if a body or body stub for a task or protected type is encountered and there is no matching specification.

*** Semantic Error :899: A body for type XXX has already been declared.
Issued if a body or body stub for a task or protected type is encountered and an earlier body has already been encountered.

*** Semantic Error :901: Suspension object XXX may only be declared immediately within a library package specification or body.
Suspension objects must be declared at library level. They cannot be used in protected type state or as local variables in subprograms.

*** Semantic Error :903: Protected or suspension object types cannot be used to declare constants.
Protected and suspension objects are used to ensure integrity of shared objects. If it is necessary to share constant data then these constructs should not be used.

*** Semantic Error :904: Protected or suspension objects cannot be used as subprogram parameters.
SPARK does not currently support this feature.

*** Semantic Error :905: Protected or suspension objects cannot be returned from functions.
SPARK does not currently support this feature.

*** Semantic Error :906: Protected or suspension objects cannot be used in composite types.
Protected and suspension objects cannot be used in record or array structures.

*** Semantic Error :907: Delay until must be called from a task or unprotected procedure body.
You are calling delay until from an invalid construct. Any construct that calls delay until must have a delay property in the declare annotation. This construct must be one of a task or procedure body.

*** Semantic Error :908: Blocking properties are not allowed in protected scope.
Procedures in protected scope must not block and therefore blocking properties are prohibited.

*** Semantic Error :909: Object XXX cannot suspend.
You are either applying the suspendable property to an own variable that cannot suspend or you have declared a variable (whose own variable has the suspendable property) which cannot suspend. Or you have used an item in a suspends list that does not have the suspendable property. An object can only suspend if it is a suspension object or a protected type with an entry.

*** Semantic Error :910: Name XXX must appear in the suspends list property for the enclosing unit.
Protected entry calls and calls to Ada.Synchronous_Task_Control.Suspend_Until_True may block the currently executing task. SPARK requires you announce this fact by placing the actual callee name in the suspends list for the enclosing unit.

*** Semantic Error :911: The argument in pragma Priority for the main program must be an integer literal or a local constant of static integer value.
If the main program priority is not an integer literal then you should declare a constant that has the required value in the declarative part of the main program prior to the position of the pragma.

*** Semantic Error :912: This call contains a delay property that is not propagated to the enclosing unit.
The call being made has a declare annotation that contains a delay property. SPARK requires that this property is propagated up the call chain and hence must appear in a declare annotation for the enclosing unit.

*** Semantic Error :913: This call has a name in its suspends list which is not propagated to the enclosing unit.
The call being made has a declare annotation that contains a suspends list. SPARK requires that the entire list is propagated up the call chain and hence must appear in a declare annotation for the enclosing unit.

*** Semantic Error :914: The name XXX specified in the suspends list can never be called.
You have specified the name of a protected or suspension object in the suspends list that can never be called by this procedure or task.

*** Semantic Error :915: Procedure XXX has a delay property but cannot delay.
You have specified a delay property for this procedure but delay until can never be called from it.

*** Semantic Error :916: Protected object XXX has a circular dependency in subprogram YYY.
The type of the protected object mentions the protected object name in the derives list for the given subprogram.

*** Semantic Error :917: Procedure XXX cannot be called from a protected action.
The procedure being called may block and hence cannot be called from a protected action.

*** Semantic Error :918: The delay property is not allowed for XXX.
The delay property may only be applied to a procedure.

*** Semantic Error :919: The priority property is not allowed for XXX.
The priority property can only be applied to protected own variables which are type announced. If the type has been declared it must be a protected type.

*** Semantic Error :920: The suspends property is not allowed for XXX.
The suspends property may only be applied to task type specifications and procedures.

*** Semantic Error :921: The identifier XXX is not recognised as a component of a property list.
The property list can only specify the reserved word delay, suspends or priority.

*** Semantic Error :922: The own variable XXX must have the priority property.
In order to perform the ceiling priority checks the priority property must be given to all own variables of protected type.

*** Semantic Error :923: The procedure XXX cannot be called from a function as it has a blocking side effect.
Blocking is seen as a side effect and hence procedures that potentially block cannot be called from functions.

*** Semantic Error :924: The suspendable property is not allowed for XXX.
Objects that suspend must be declared as own protected variables.

*** Semantic Error :925: The own variable or task XXX must have a type announcement.
Own variables of protected type and own tasks must have a type announcement.

*** Semantic Error :926: Illegal declaration of task XXX. Task objects must be declared at library level.
Task objects must be declared in library level package specifications or bodies.

*** Semantic Error :927: The own task annotation for this task is missing the name XXX in its suspends list.
The task type declaration has name XXX in its list and this must appear in the own task annotation.

*** Semantic Error :928: Private elements are not allowed for protected type XXX.
Protected type XXX has been used to declare a protected, moded own variable. Protected, moded own variables are refined onto a set of virtual elements with the same mode. As such private elements are not allowed.

*** Semantic Error :929: Unexpected declare annotation. Procedure XXX should have the declare annotation on the specification.
Declare annotations cannot appear on the procedure body if it appears on the procedure specification.

*** Semantic Error :930: Task XXX does not appear in the own task annotation for this package.
A task has been declared that is not specified as an own task of the package.

*** Semantic Error :931: Task XXX does not have a definition.
A task name appears in the own task annotation for this package but is never declared.

*** Semantic Error :932: The priority for protected object XXX does not match that given in the own variable declaration.
The priority given in the priority property must match that given in the protected type.

*** Semantic Error :933: A pragma Priority is required for the main program when Ravenscar Profile is enabled.
When SPARK profile Ravenscar is selected, all tasks, protected objects and the main program must explicitly be assigned a priority.

*** Semantic Error :934: Priority ceiling check failure: the priority of YYY is less than that of XXX.
The active priority of a task is the higher of its base priority and the ceiling priorities of all protected objects that it is executing. The active priority at the point of a call to a protected operation must not exceed the ceiling priority of the callee.

*** Semantic Error :935: The own variable XXX must have the interrupt property.
An own variable has been declared using a protected type with a pragma attach handler. Such objects are used in interrupt processing and must have the interrupt property specified in their own variable declaration.

*** Semantic Error :936: The interrupt property is not allowed for XXX.
The interrupt property can only be applied to protected own variables that are type announced. If the type is declared then it must be a protected type that contains an attach handler.

*** Semantic Error :937: The protects property is not allowed for XXX.
The protects property can only be applied to protected own variables that are type announced. If the type is declared then it must be a protected type.

*** Semantic Error :938: The unprotected variable XXX is shared by YYY and ZZZ.
XXX is an unprotected variable that appears in the global list of the threads YYY and ZZZ. Unprotected variables cannot be shared between threads in SPARK. A thread is one of: the main program, a task, an interrupt handler.

*** Semantic Error :939: The suspendable item XXX is referenced by YYY and ZZZ.
XXX is an own variable with the suspends property that appears in the suspends list of the threads YYY and ZZZ. SPARK prohibits this to prevent more than one thread being suspended on the same item at any one time. A thread is one of: the main program, a task, an interrupt handler.

*** Semantic Error :940: XXX is a protected own variable. Protected variables may not be used in proof contexts.
The use of protected variables in pre and postconditions or other proof annotations is not (currently) supported. Protected variables are volatile because they can be changed at any time by another program thread and this may invalidate some common proof techniques. The prohibition of protected variables does not prevent proof of absence of run-time errors nor proof of protected operation bodies. See the manuals "Generation of VCs" and "Generation of RTCs" for more details.

*** Semantic Error :941: The type of own variable XXX must be local to this package.
The type used to an announce an own variable with a protects property must be declared in the same package.

*** Semantic Error :942: Only one instance of the type XXX is allowed.
Type XXX has a protects property. This means there can be only one object in the package that has this type or any subtype of this type.

*** Semantic Error :943: The name XXX cannot appear in a protects list.
All items in a protects list must be unprotected own variables owned by this package.

*** Semantic Error :944: The name XXX is already protected by YYY.
The name XXX appears in more than one protects list. The first time it appeared was for own variable YYY. XXX should appear in at most one protects list.

*** Semantic Error :945: The property XXX must be given a static expression for its value.
This property can only accept a static expression.

*** Semantic Error :946: The own variable XXX must only ever be accessed from operations in protected type YYY.
The own variable XXX is protected by the protected type YYY and hence must never be accessed from anywhere else.

*** Semantic Error :947: The own variable XXX appears in a protects list for type YYY but is not used in the body.
The protected type YYY claims to protect XXX via a protects property. However, the variable XXX is not used by any operation in YYY.

*** Semantic Error :948: The type of own variable or task XXX must be a base type.
Own tasks and protected own variables of a protected type must be announced using the base type. The subsequent variable declaration may be a subtype of the base type.

*** Semantic Error :949: Unexpected partition annotation: a global annotation may only appear here when the Ravenscar profile is selected.
When the sequential SPARK profile is selected, the global and derives annotation on the main program describes the entire program's behaviour. No additional, partition annotation is required or permitted. Note that an annotation must appear here if the Ravenscar profile is selected.

*** Semantic Error :950: Partition annotation expected: a global and, optionally, a derives annotation must appear after 'main_program' when the Ravenscar profile is selected.
When the Ravenscar profile is selected the global and derives annotation on the main program describes the behaviour of the environment task only, not the entire program. An additional annotation, called the partition annotation, is required to describe the entire program's behaviour; this annotation follows immediately after 'main_program;'.

*** Semantic Error :951: Inherited package XXX contains tasks and/or interrupt handlers and must therefore appear in the preceding WITH clause.
In order to ensure that a Ravenscar program is complete, SPARK requires that all 'active' packages inherited by the environment task also appear in a corresponding with clause. This check ensures that any program entities described in the partition annotation are also linked into the program itself.

*** Semantic Error :952: Subprogram XXX is an interrupt handler and cannot be called.
Interrupt handler operations cannot be called.

*** Semantic Error :953: Interrupt property error for own variable YYY. XXX is not an interrupt handler in type ZZZ.
The handler names in an interrupt property must match one in the protected type of the own variable.

*** Semantic Error :954: Interrupt property error for own variable XXX. Interrupt stream name YYY is illegal.
The stream name must be unprefixed and not already in use within the scope of the package.

*** Semantic Error :955: XXX can only appear in the partition wide flow annotation.

Interrupt stream variables are used only to enhance the partition wide flow annotation and must not be used elsewhere.

*** Semantic Error :956: XXX already appears in as an interrupt handler in the interrupt mappings.
An interrupt handler can be mapped onto exactly one interrupt stream variable. An interrupt stream variable may be mapped onto many interrupt handlers.

*** Semantic Error :957: Unconditional update of protected variable XXX not allowed here.
You may only wholly update a protected variable once within a sequence of unprotected statements. As the protected state may be updated by another thread between two successive unconditional updates it would be impossible to reason about information flow for the state.

*** Semantic Error :958: A task may not import the unprotected state XXX.
A task may not imoprt unprotected state unless it is mode in. This is because under the concurrent elaboration policy, the task cannot rely on the state being initialised before it is run.

*** Semantic Error :959: Unprotected state XXX is exported by a task and hence must not appear in an initializes clause.
Own variable XXX is being accessed by a task. The order in which the task is run and the own variable initialised is non-deterministic under a concurrent elaboration policy. In this case SPARK forces the task to perform the initialisation and as such the own variable must not appear in an initializes clause.

*** Semantic Error :960: The function Ada.Real_Time.Clock can only be used directly (1) in an assignment or return statement or (2) to initialise a library a level constant.

- *To avoid ordering effects, functions which globally access own variables which have modes (indicating that they are connected to the external environment) may only appear directly in assignment or return statements. They may not appear as actual parameters or in any other form of expression.*
- *SPARK relaxes the illegal use of function calls in elaboration code in the case of the function Ada.Real_Time.Clock. However the function can only be used to directly initialise a constant value.*

*** Semantic Error :961: This property value is of an incorrect format.
Please check the user manual for valid property value formats.

*** Semantic Error :986: A protected function may not call a locally-declared protected procedure.
See LRM 9.5.1 (2). A protected function has read access to the protected elements of the type whereas the called procedure has read-write access. There is no way in which an Ada compiler can determine whether the procedure will illegally update the protected state or not so the call is prohibited by the rules of Ada. (Of course, in SPARK, we know there is no function side effect but the rules of Ada must prevail nonetheless).

*** Semantic Error :987: Task types and protected types may only be declared in package specifications.
The Examiner performs certain important checks at the whole program level such as detection of illegal sharing of unprotected state and partition-level information flow analysis. These checks require visibility of task types and protected types (especially those containing interrupt handlers). SPARK therefore requires these types to be declare in package specifications. Subtypes and objects of task types, protected types and their subtypes may be declared in package bodies.

*** Semantic Error :988: Illegal re-use of identifier XXX; this identifier is used in a directly visible protected type.

SPARK does not allow the re-use of operation names which are already in use in a directly visible protected type. The restriction is necessary to avoid overload resolution issues in the protected body. For example, type PT in package P declares operation K. Package P also declares an operation K. From inside the body of PT, a call to K could refer to either of the two Ks since both are directly visible.

*** Semantic Error :989: The last statement of a task body must be a plain loop with no exits.
To prevent any possibility of a task terminating (which can lead to a bounded error), SPARK requires each task to end with a non-terminating loop. The environment task (or "main program") does not need to end in a plain loop provided the program closure includes at least one other task. If there are no other tasks, then the environment task must be made non-terminating with a plain loop.

*** Semantic Error :990: Unexpected annotation, a task body may have only global and derives annotations.
Issued if a pre, post or declare annotation is attached to a task body.

*** Semantic Error :991: Unexpected task body, XXX is not the name of a task declared in this package specification.
Issued if task body is encountered for which there is no preceding declaration.

*** Semantic Error :992: A body for task type XXX has already been declared.
Issued if a duplicate body or body stub is encountered for a task.

*** Semantic Error :993: There is no protected type declaration for XXX.
Issued if a body is found for a protected types for which there is no preceding declaration.

*** Semantic Error :994: Invalid guard, XXX is not a Boolean protected element of this protected type.
The SPARK Ravenscar rules require a simple Boolean guard which must be one of the protected elements of the type declaring the entry.

*** Semantic Error :995: Unexpected entry body, XXX is not the name of an entry declared in this protected type.
Local entries are not permitted so a protected body can declare at most one entry body and that must have declared in the protected type specification.

*** Semantic Error :996: The protected operation XXX, declared in this type, does not have an associated body.
Each exported protected operation must have a matching implementation in the associated protected body.

*** Semantic Error :997: A body for protected type XXX has already been declared.
Each protected type declaration must have exactly one matching protected body or body stub.

*** Semantic Error :998: There is no protected type declaration for XXX.
Issued if a protected body or body stub is found and there is no matching declaration for it.

*** Semantic Error :999: This feature of Generics is not yet implemented.

6.3 Warning messages

As well as the error messages described above the Examiner may produce warnings. The presence of warning messages does not mean that the program contains errors but does indicate areas which may require additional care by the user because the construction warned about may affect the meaning of a SPARK program in ways that cannot be detected by the Examiner. Section 4.3 describes a way of instructing the Examiner to summarise, rather than individually report, some or all of these warning messages; the keyword used in the warning control file to inhibit full reporting in each case is included in the explanations which follow. The possible warning messages are described below

Warning : No semantic checks carried out, text may not be legal SPARK.
Issued when the Examiner is used solely to check the syntax of a SPARK text: this does not check the semantics of a program (e.g. the correctness of the annotations) and therefore does not guarantee that a program is legal SPARK.

— Warning :1: The identifier XXX is either undeclared or not visible at this point.
This warning will appear against an identifier in a with clause if it is not also present in an inherit clause. Such an identifier cannot be used in any non-hidden part of a SPARK program. The use of with without inherit is permitted to allow reference in hidden parts of the text to imported packages which are not legal SPARK. For example, the body of SPARK_IO is hidden and implements the exported operations of the package by use of package TEXT_IO. For this reason TEXT_IO must appear in the with clause of SPARK_IO. (warning control file keyword: with_clauses).

— Warning :2: Representation clause - ignored by the SPARK Examiner.
The significance of representation clauses cannot be assessed by the Examiner because it depends on the specific memory architecture of the target system. Like pragmas, representation clauses can change the meaning of a SPARK program and the warning highlights the need to ensure their correctness by other means. (warning control file keyword: representation_clauses).

— Warning :3: Pragma - ignored by the SPARK Examiner.
All pragmas encountered by the Examiner generate this warning. While many pragmas (e.g. pragma page) are harmless others can change a program's meaning, for example by causing two variables to share a single memory location. (warning control file keyword: pragma pragma_identifier or pragma all).

— Warning :4: declare annotation - ignored by the SPARK Examiner.
The declare annotation is ignored by the Examiner if the profile is not Ravenscar. (warning control file keyword: declare_annotations).

— Warning :5: XXX contains interrupt handlers; it is important that an interrupt identifier is not used by more than one handler.
Interrupt identifiers are implementation defined and the Examiner cannot check that values are used only once. Duplication can occur by declaring more than object of a single (sub)type where that type defines handlers. It may also occur if interrupt identifiers are set via discriminants and two or more actual discriminants generate the same value. (warning control file keyword: interrupt_handlers).

— Warning :6: Machine code insertion. Code insertions are ignored by the Examiner.

Machine code is inherently implementation dependent and cannot be analysed by the SPARK Examiner. Users are responsible for ensuring that the behaviour of the inserted machine code matches the annotation of the subprogram containing it.

— Warning :9: The body of XXX has a hidden exception handler - analysis and verification of contracts for this handler have not been performed.
Issued when a –# hide XXX annotation is used to hide a user-defined exception handler. (warning control file keyword: handler_parts).

— Warning :10: XXX is hidden - hidden text is ignored by the SPARK Examiner.
Issued when a –# hide XXX annotation is used. (warning control file keyword: hidden_parts).

— Warning :11: Unnecessary others clause - case statement is already complete.
The others clause is non-executable because all case choices have already been covered explicitly. If the range of the case choice is altered later then the others clause may be executed with unexpected results. It is better to omit the others clause in which case any extension of the case range will result in a compilation error.

— Warning :12: Function XXX is an instantiation of Unchecked_Conversion.
See ALRM 13.9. The use of Unchecked_Conversion can result in implementation-defined values being returned. The function should be used with great care. The principal use of Unchecked_Conversion is SPARK programs is the for the reading of external ports prior to performing a validity check; here the suppression of contraint checking prior to validation is useful. The Examiner does not assume that the value returned by an unchecked conversion is valid and so unprovable run-time check VCs will result if a suitable validity check is not carried out before the value is used. (warning control file keyword: unchecked_conversion).

— Warning :13: Function XXX is an instantiation of Unchecked_Conversion returning a type for which run-time checks are not generated. Users must take steps to ensure the validity of the returned value.
See ALRM 13.9. The use of Unchecked_Conversion can result in invalid values being returned. The function should be used with great care especially, as in this case, where the type returned does not generate Ada run-time checks nor SPARK run-time verification conditions. For such types, this warning is the ONLY reminder the Examiner generates that the generated value may have an invalid representation. For this reason the warning is NOT supressed by the warning control file keyword unchecked_conversion. The principal use of Unchecked_Conversion is SPARK programs is the for the reading of external ports prior to performing a validity check; here the suppression of contraint checking prior to validation is useful.

— Warning :169: Direct update of own variable of a non-enclosing package.
With the publication of Edition 3.1 of the SPARK Definition the previous restriction prohibiting the direct updating of own variables of non-enclosing packages was removed; however, the preferred use of packages as abstract state machines is compromised by such action which is therefore discouraged. (warning control file keyword: direct_updates).

— Warning :200: This static expression cannot be evaluated by the Examiner.
Issued if a static expression exceeds the internal limits of the Examiner because its value is, for example, too large to be evaluated using infinite precision arithmetic. No value will be recorded for the expression and this may limit the Examiner's ability to detect certain sorts of errors such as numeric constraints. (warning control file keyword: static_expressions).

— Warning :201: This expression cannot be evaluated statically because its value may be implementation-defined.
Raised, for example, when evaluating 'Size of a type that does not have an explicit Size representation clause. Attributes of implementation-defined types, such as Integer'Last may also be unknown to be Examiner if they are not specified in the configuration fil (warning control file keyword: static_expressions).

- Warning :202: An arithmetic overflow has ocurred. Constraint checks have not been performed.
  Raised when comparing two real numbers. The examiner cannot deal with real numbers specified to such a high degree of precision. Consider reducing the precision of these numbers.
- Warning :300: VCs and PFs cannot be built for multi-dimensional array aggregates.
  Issued when an aggregate of a multi-dimensional array is found. Suppresses generation of VCs or PFs for that subprogram. Can be worked round by using arrays of arrays.
- Warning :301: Called subprogram exports abstract types for which RTCs are not possible.
- Warning :302: This expression may be re-ordered by a compiler. Left to right evaluation was assumed for RTC generation.
  Issued when a potentially re-orderable expression is encountered when run-time expression overflow checks are being generated. For example x := a + b + c; Whether intermediate sub-expression values overflow may depend on the order of evaluation which is compiler-dependent. The Examiner generates RTCs on the basis of left-to-right evaluation; however, code generating this warning should be parenthesised to remove the ambiguity. e.g. x := (a + b) + c;.
- Warning :303: Overlapping choices may not be detected.
  Issued where choices in an array aggregate or case statement are outside the range which can be detected because of limits on the size of a table internal to the Examiner.
- Warning :304: Case statement may be incomplete.
  Issued when the Examiner cannot determine the completeness of a case statement because the bounds of the type of the controlling expression exceed the size of the internal table used to perform the checks.
- Warning :305: Value too big for internal representation.
  Issued when the Examiner cannot determine the completeness of an array aggregate or case statement because the number used in a choice exceed the size allowed in the internal table used to perform the checks.
- Warning :306: Aggregate may be incomplete.
  Issued when the Examiner cannot determine the completeness of an array aggregate because its bounds exceed the size of the internal table used to perform the checks.
- Warning :307: Completeness checking incomplete: index type(s) undefined or not discrete.
  Issued where the array index (sub)type is inappropriate: this is probably because there is an error in its definition, which will have been indicated by a previous error message.
- Warning :308: Use of equality operator with floating point type.
  The use of this operator is discouraged in SPARK because of the difficulty in determining exactly what it means to say that two instances of a floating point number are equal.
- Warning :309: Unnecessary type conversion to own type.
  Issued where a type conversion is either converting from a (sub)type to the same (sub)type or is converting between two subtypes of the same type. In the former case the type conversion may be safely removed because no constraint check is required; in the latter case the type conversion may be safely replaced by a type qualification which preserves the constraint check.(warning control file keyword: type_conversions).
- Warning :310: Use of obsolescent Ada 83 language feature.
  Issued when a language feature defined by Ada 95 to be obsolescent is used. Use of such features is not recommended because compiler support for them cannot be guaranteed.
- Warning :311: Priority pragma for XXX is unavailable and has not been considered in the ceiling priority check.

- Warning :312: Replacement rules cannot be built for multi-dimensional array constant XXX.
  Issued when a VC or PF references a multi-dimensional array constant. Can be worked round by using arrays of arrays.
- Warning :313: The constant XXX has semantic errors in its initializing expression or has a hidden completion which prevent generation of a replacement rule.
  Issued when replacement rules are requested for a composite constant which had semantic errors in its initializing expression, or is a deferred constant whose completion is hidden from the Examiner. Semantic errors must be eliminated before replacement rules can be generated.
- Warning :314: The constant XXX has semantic errors in its type which prevent generation of rules.
  Issued when an attempt is made to generate type deduction rules for a constant which has semantic errors in its type. These semantic errors must be eliminated before type deduction rules can be generated.
- Warning :350: Unexpected pragma Import. Variable XXX is not identified as an external (stream) variable.
  The presence of a pragma Import makes it possible that the variable is connected to some external device. The behaviour of such variables is best captured by making them moded own variables (or "stream" variables). If variables connected to the external environment are treated as if they are normal program variables then misleading analysis results are inevitable. The use of pragma Import on local variables of subprograms is particularly deprecated. The warning may safely be disregarded if the variable is not associated with memory-mapped input/output or if the variable concerned is an own variable and the operations on it are suitably annotated to indicate volatile, stream-like behaviour. Where pragma Import is used, it is essential that the variable is properly initialized at the point from which it is imported. (warning control file keyword:imported_objects).
- Warning :351: Unexpected address clause. XXX is a constant.
  Great care is needed when attaching an address clause to a constant. The use of such a clause is safe if, and only if, the address supplied provides a valid value for the constant which does not vary during the execution life of the program, for example, mapping the constant to PROM data. If the address clause causes the constant to have a value which may alter, or worse, change dynamically under the influence of some device external to the program, then misleading or incorrect analysis is certain to result. If the intention is to create an input port of some kind, then a constant should not be used. Instead a moded own variable (or "stream" variables) should be used. (warning control file keyword: address_clauses).
- Warning :390: This generic subprogram has semantic errors in its declaration which prevent instantiations of it.
  Issued to inform the user that a generic subprogram instantiation cannot be completed because of earlier errors in the generic declaration.
- Warning :391: If the identifier XXX represents a package which contains a task or an interrupt handler then the partition-level analysis performed by the Examiner will be incomplete. Such packages must be inherited as well as withed.
- Warning :392: External variable XXX may have an invalid representation.
  Where values are read from external variables (i.e. variables connected to the external environment) there is no guarantee that the bit pattern read will be a valid representation for the type of the external variable. Unexpected behaviour may result if invalid values are used in expressions. For SPARK 95 the use of attribute 'Valid is strongly recommended (see ALRM 13.9.2). For SPARK 83 external variables should always be read into variables of a type for which any bit pattern would be a valid representation (e.g. a "word") and then range checked before conversion to the actual type desired. Note that when the Examiner is used to generate

run-time checks, it will not be possible to discharge those involving external variables unless these steps are taken. Boolean external variables require special care since the Examiner does not generate run-time checks for Boolean variables; use of 'Valid is essential when reading Boolean external variables. More information on interfacing can be found in the INFORMED manual. (warning control file keyword: external_assignment).

— Warning :393: External variable XXX may have an invalid representation and is of a type for which run-time checks are not generated. Users must take steps to ensure the validity of the assigned or returned value.
Where values are read from external variables (i.e. variables connected to the external environment) there is no guarantee that the bit pattern read will be a valid representation for the type of the external variable. Unexpected behaviour may result if invalid values are used in expressions. Where, as in this case, the type is one for which neither Ada run-time checks nor SPARK run-time verification conditions are generated, extra care is required. For such types, this warning is the ONLY reminder the Examiner generates that the external value may have an invalid representation. For this reason the warning is NOT supressed by the warning control file keyword external_assignment. For SPARK 95 the use of attribute 'Valid is strongly recommended (see ALRM 13.9.2). For SPARK 83 external variables should always be read into variables of a type for which any bit pattern would be a valid representation (e.g. a "word") and then range checked before conversion to the actual type desired. Note that when the Examiner is used to generate run-time checks, it will not be possible to discharge those involving external variables unless these steps are taken. Boolean external variables require special care since the Examiner does not generate run-time checks for Boolean variables; use of 'Valid is essential when reading Boolean external variables. More information on interfacing can be found in the INFORMED manual.

— Warning :394: Variables of type XXX cannot be initialized using the facilities of this package.
A variable of a private type can only be used (without generating a data flow error) if there is some way of giving it an initial value. For a limited private type only a procedure that has an export of that type and no imports of that type is suitable. For a private type either a procedure, function or (deferred) constant is required. The required facility may be placed in, or already available in, a public child package.

— Warning :395: Variable XXX is an external (stream) variable but does not have an address clause or a pragma import.
When own variables are given modes they are considered to be inputs from or outputs to the external environment. The Examiner regards them as being volatile (i.e. their values can change in ways not visible from an inspection of the source code). If a variable is declared in that way but it is actually an ordinary variable which is NOT connected to the environment then misleading analysis is inevitable. The Examiner expects to find an address clause or pragma import for variables of this kind to indicate that they are indeed memory-mapped input/output ports. This warning is issued if an address clause or pragma import is not found.

— Warning :396: Unexpected address clause. Variable XXX is not identified as an external (stream) variable.
The presence of an address clause makes it possible that the variable is connected to some external device. The behaviour of such variables is best captured by making them moded own variables (or "stream" variables). If variables connected to the external environment are treated as if they are normal program variables then misleading analysis results are inevitable. The use of address clauses on local variables of subprograms is particularly deprecated. The warning may safely be disregarded if the variable is not associated with memory-mapped input/output or if the variable concerned is an own variable and the operations on it are suitably annotated to indicate volatile, stream-like behaviour. (warning control file keyword: address_clauses).

— Warning :397: Variables of type XXX can never be initialized before use.
A variable of a private type can only be used (without generating a data flow error) if there is some way of giving it an initial value. For a limited private type only a procedure that has an export of that type and no imports of that type is suitable. For a private type either a procedure, function or (deferred) constant is required.

— Warning :398: The own variable XXX can never be initialized before use.
The own variable can only be used (without generating a data flow error) if there is some way of giving it an initial value. If it is initialized during package elaboration (or implicitly by the environment because it represents an input port) it should be placed in an "initializes" annotation. Otherwise there needs to be some way of assigning an initial value during program execution. Either the own variable needs to be declared in the visible part of the package so that a direct assignment can be made to it or, more usually, the package must declare at least one procedure for which the own variable is an export but not an import. Note that if the own variable is an abstract own variable with some constituents initialized during elaboration and some during program execution then it will never be possible correctly to initialize it; such abstract own variables must be divided into separate initialized and uninitialized components.

— Warning :399: The called subprogram has semantic errors in its interface (parameters and/or annotations) which prevent flow analysis of this call.
Issued to inform the user that flow analysis has been suppressed because of the error in the called subprogram's interface.

— Warning :400: Variable XXX is declared but not used.
Issued when a variable declared in a subprogram is neither referenced, nor updated. (warning control file keyword: unused_variables).

— Warning :401: RTCs have not been generated for statements including real numbers.
The SPARK Examiner does not, by default, generate run-time checks for real numbers, due to the imprecise nature of their machine representations. Facilities can be made available for run-time checking of real numbers: please contact Praxis High Integrity Systems if this is of interest.

— Warning :402: Default assertion planted to cut loop.
In order to prove properties of code containing loops, the loop must be "cut" with a suitable assertion statement. When generating run-time checks, the Examiner inserts a simple assertion to cut any loops which do not have one supplied by the user. The assertion is placed at the point where this warning appears in the listing file. The default assertion asserts that the subprogram's precondition (if any) is satisfied, that all imports to it are in their subtypes and that any for loop counter is in its subtype. In many cases this provides sufficient information to complete a proof of absence of run-time errors. If more information is required, then the user can supply an assertion and the Examiner will append the above information to it.

— Warning :403: XXX is declared as a variable but used as a constant.
XXX is a variable which was initialized at declaration but whose value is only ever read not updated; it could therefore have been declared as a constant. (warning control file keyword: constant_variables).

— Warning :404: Subprogram imports variables of abstract types for which run-time checks cannot be generated.

— Warning :405: RTCs for statements including real numbers are approximate.
Where the optional RealRTC switch is used, the Examiner generates checks associated with real numbers using perfect arithmetic rather than the machine approximations used on the target platform. It is possible that rounding errors might cause a constraint_error even if these run-time check proofs are completed satisfactorily. Please consult the Supplementary Release Note "The RealRTC Option" for more details.

- Warning :406: Subprogram imports or exports unconstrained array; existence of run-time errors depends on the calling environment, see RTC manual.
  Where a formal parameter of a subprogram is an unconstrained array the size of the array is determined by the size of the actual parameter in the calling environment; this causes two difficulties for the proof of absence of run-time errors. (1) Checks on the validity of index expressions which are made inside the body of the subprogram will be in terms of the unconstrained array index type rather than in terms of the actual constrained size of the array for a particular call. Therefore the successful proof of the index to be within the unconstrained array type is not sufficient to prove the absence of run-time errors. It is necessary to insert suitable annotations to move the proof obligation to the point of call, where the array index type is known. (2) Attributes such as 'Last of the array will not have a constant value for proof purposes since the value will vary from call to call depending on the actual parameter passed.
- Warning :407: This package requires a body. Care should be taken to provide one because an Ada compiler will not detect its omission.
  Issued where SPARK own variable and initialization annotations make it clear that a package requires a body but where no Ada requirement for a body exists.
- Warning :408: VCs could not be generated for this subprogram due to errors in its abstract specification. Unprovable (False) VC generated.
  Because there are errors in the abstract specification of the subprogram proofs of refinement consistency would be invalid and so VCs have not been generated. Correct the first annotations for this subprogram in its package specification.
- Warning :409: RTCs have not been generated for expressions of type Time or Time_Span.
  Types Time and Time_Span are private types which are implemented as an implementation-defined integer type (see ALRM 9.6 and D.8. Since there is no way of knowing the range of values that are valid for these types, the Examiner cannot generate checks to ensure that constraint is error is not raised in their use. This warning serves as a reminder that the validity of expressions involving Time and Time_Span must be checked by other means.
- Warning :410: Task or interrupt handler XXX is either unavailable (hidden) or has semantic errors in its specification which prevent partition-wide flow analysis being carried out.
  Partition-wide flow analysis is performed by checking all packages withed by the main program for tasks and interrupt handlers and constructing an overall flow relation that captures their cumulative effect. It is for this reason that SPARK requires task and protected types to be declared in package specifications. If a task or protected type which contains an interrupt handler, is hidden from the Examiner (in a hidden package privat part) or contains errors in it specification, the partition-wide flow analysis cannot be constructed correctly and is therefore supressed. Correct the specification of the affected tasks and (temporarily if desired) make them visible to the Examiner.
- Warning :411: Task type XXX is unavailable and has not been considered in the shared variable check.
  The Examiner checks that there is no potential sharing of unprotected data between tasks. If a task type is hidden from the Examiner in a hidden package private part, then it is not possible to check whether that task may share unprotected data.
- Warning :412: Task type XXX is unavailable and has not been considered in the max-one-in-a-queue check.
  The Examiner checks that no more than one task can suspend on a single object. If a task is hidden from the Examiner in a hidden package private part, then it is not possible to check whether that task may suspend on the same object as another task.
- Warning :413: Task or main program XXX has errors in its annotations. The shared variable and max-one-in-a-queue checks may be incomplete.

The Examiner checks that no more than one task can suspend on a single object and that there is no potential sharing of unprotected data between tasks. These checks depend on the accuracy of the annotations on the task types withed by the main program. If these annotations contain errors, then any reported violations of the shared variable and max-one-in-a-queue checks will be correct; however, the check may be incomplete. The errors in the task annotations should be corrected.

6.4 Notes

The Examiner may also produce the following messages, which draw the user's attention to points which are considered worthy of note, but which are considered to be less serious than the warnings described above.

Some of these messages may be suppressed by the use of the warning control file as described in section 4.3 where indicated below.

Note: Ada 83 language rules selected.
Issued when the Examiner is used in SPARK 83 mode.

Note: Information flow analysis not carried out.
This is issued as a reminder that information flow analysis has not been carried out in this run of the Examiner: information flow errors may be present undetected in the text analysed.

— Note :1: This dependency relation was not used for this analysis and has not been checked for accuracy.
Issued when information flow analysis is not performed and when modes were specified in the global annotation. It is a reminder that the dependencies specified in this annotation (including whether each variable is an import or an export) have not been checked against the code, and may therefore be incorrect. (warning control file keyword: notes).

— Note :2: This dependency relation has been used only to identify imports and exports, dependencies have been ignored.
Issued as a reminder when information flow analysis is not performed in SPARK 83. The dependencies specified in this annotation have not been checked against the code, and may therefore be incorrect. (warning control file keyword: notes).

— Note :3: The deferred constant Null_Address has been implicitly defined here.
Issued as a reminder that the declaration of the type Address within the target configuration file implicitly defines a deferred constant of type Null_Address. (warning control file keyword: notes).

— Note :4: The constant Default_Priority, of type Priority, has been implicitly defined here.
Issued as a reminder that the declaration of the subtype Priority within the target configuration file implicitly defines a constant Default_Priority, of type Priority, with the value (Priority'First + Priority'Last) / 2. (warning control file keyword: notes).

7 Control flow analysis

7.1 General description

Since Ada is already rich in control structures, it is possible to remove its goto statement without unreasonably hindering the programmer. Then, to ensure that control structures are "well-formed", for analysis purposes, it suffices to place minor restrictions on the placement of exit statements, and return statements. (Precise details are given in the SPARK Definition.) Checks that exit and return statements have been correctly placed are performed directly on the abstract syntax tree of a SPARK text.

7.2 Error messages

Exit statements that violate the rules given in the SPARK Definition are always indicated by the following message:

*** Illegal Structure :1: An exit statement may not occur here.
Exit statements must be of the form "exit when c;" where the closest enclosing statement is a loop or "if c then S; exit;" where the if statement has no else part and its closest enclosing statement is a loop. See the SPARK Definition for details.

Violations of the rules concerning return statements are indicated by messages of the following kinds:

*** Illegal Structure :2: A return statement may not occur here.
A return statement may only occur as the last statement of a function.

*** Illegal Structure :3: The last statement of this function is not a return statement.
SPARK requires that the last statement of a function be a return statement.

*** Illegal Structure :4: Return statements may not occur in procedure subprograms.

8 Data and information flow analysis

8.1 General description

The data-flow and information-flow relations on which this analysis is based have been defined recursively, in such a way that they can be computed directly, in the course of a traversal of the abstract syntax tree of a SPARK text. The principles of the method are described in the paper given in Appendix A (though the recurrence relations used to compute the flow relations of SPARK loop structures are rather more complex).

Data-flow and information-flow errors are detected by mechanical inspection of these relations.

8.2 Optional information flow analysis

8.2.1 Description

This feature can be used for analysis of SPARK 95 programs where global modes have been provided. The syntax for moded global annotations is described in the SPARK Definition. For example:

```
procedure P(X : in     T;
            Y :    out T);
--# global in     A;
--#        in out B, C;
--#           out D;
```

Provided global modes are present and the Examiner is operating in SPARK 95 mode, data flow analysis can be selected using the /flow_analysis=data command line option described in Section 3.1.3. When selected, the Examiner will ignore any derives annotations it encounters. A note to this effect is generated but it can be suppressed if desired by using the warning file mechanism described in Section 4.3. Instead of using the derives annotation, the Examiner determines whether each parameter and global is an import or an export by checking its mode. Data flow analysis is then conducted using this mode information. The validity of the data flow errors listed in Section 8.3.1 is completely unaffected. Other error messages generated also remain valid; however, some errors normally found by information flow analysis may no longer be detected. For example, if the Examiner (in data flow analysis mode) detects a stable loop (see Section 8.3.3) then the loop is stable; however, not all cases of loop stability detectable by information flow analysis will necessarily be indicated. The specific information flow error messages described in Section 8.3.4 will not appear at all.

It is possible to generate verification conditions, including run-time checks, in conjunction with the data flow analysis option; information flow analysis is not a prerequisite for these forms of analysis. In contrast, it is not possible to generate path functions unless information flow analysis has been carried out.

8.2.2 Recommended use

For new developments we recommend that global modes are supplied and that derives annotations are also supplied except where, high in the calling tree, their size and complexity is judged to make them of limited value. This approach allows full information flow analysis to be performed lower in the calling hierarchy while still permitting full protection from data flow errors and language violations at the top of the calling tree (e.g. in the system main loop or scheduler). Additionally, the use of moded global as a substitute for derives annotations may be appropriate for some, perhaps less critical, systems.

8.2.3 SPARK 83 data flow analysis

The data flow analysis option is not available when operating the Examiner in SPARK 83 mode; this is because Ada 83 language rules prevent the modes of a parameter being sufficient to determine whether it is an import or an export. Users who have a need for data flow analysis of SPARK 83 programs should contact Praxis High Integrity Systems for guidance.

8.3 Error messages

Data-flow and information-flow errors may be either unconditional or conditional. An error is unconditional if it applies to all paths through a program (and consequently to all its possible executions). In signalling flow errors of this kind, error messages always begin with three exclamation marks (!!!). A conditional flow error is one that applies to some but not all program paths. To determine whether such an error can manifest itself in practice, it is necessary to establish whether any of the paths to which the error applies are executable -which involves their semantic analysis. Error reports on conditional errors begin with three question marks (???).

8.3.1 Data-flow errors (References to undefined variables)

!!! Flow Error :20: Expression contains reference(s) to undefined variable XXX.
The expression may be that in an assignment or return statement, an actual parameter, or a condition occurring in an if or case statement, an iteration scheme or exit statement. NOTE: the presence of random and possibly invalid values introduced by data flow errors invalidates proof of exception freedom for the subprogram body which contains them. All unconditional data flow errors must be eliminated before attempting exception freedom proofs. See the manual "Generation of RTCs for SPARK Programs" for full details.

!!! Flow Error :23: Statement contains reference(s) to undefined variable XXX.
The statement here is a procedure call or an assignment to an array element, and the variable XXX may appear in an actual parameter, whose value is imported when the procedure is executed. If the variable XXX does not occur in the actual parameter list, it is an imported global variable of the procedure (named in its global definition). NOTE: the presence of random and possibly invalid values introduced by data flow errors invalidates proof of exception freedom for the subprogram body which contains them. All unconditional data flow errors must be

eliminated before attempting exception freedom proofs. See the manual "Generation of RTCs for SPARK Programs" for full details.

— Warning :501: Expression contains reference(s) to variable XXX, which may be undefined.
The expression may be that in an assignment or return statement, an actual parameter, or a condition occurring in an if or case statement, an iteration scheme or exit statement. The Examiner has identified at least one syntactic path to this point where the variable has NOT been given a value. Conditional data flow errors are extremely serious and must be carefully investigated. NOTE: the presence of random and possibly invalid values introduced by data flow errors invalidates proof of exception freedom for the subprogram body which contains them. All reports of data flow errors must be eliminated or shown to be associated with semantically infeasible paths before attempting exception freedom proofs. See the manual "Generation of RTCs for SPARK Programs" for full details.

— Warning :504: Statement contains reference(s) to variable XXX, which may be undefined.
The statement here is a procedure call, and the variable XXX may appear in an actual parameter, whose value is imported when the procedure is executed. If the variable XXX does not occur in the actual parameter list, it is an imported global variable of the procedure (named in its global definition). The Examiner has identified at least one syntactic path to this point where the variable has NOT been given a value. Conditional data flow errors are extremely serious and must be carefully investigated. NOTE: the presence of random and possibly invalid values introduced by data flow errors invalidates proof of exception freedom for the subprogram body which contains them. All reports of data flow errors must be eliminated or shown to be associated with semantically infeasible paths before attempting exception freedom proofs. See the manual "Generation of RTCs for SPARK Programs" for full details.

8.3.2 Data-flow anomalies and ineffective statements

!!! Flow Error :10: Ineffective statement.
Execution of this statement cannot affect the final value of any exported variable of the subprogram in which it occurs. The cause may be a data-flow anomaly (i.e. the statement could be an assignment to a variable, which is always updated again before it is read. However, statements may be ineffective for other reasons - see Section 4.1 of Appendix A.

!!! Flow Error :10: Assignment to XXX is ineffective.
This message always relates to a procedure call or an assignment to a record. The variable XXX may be an actual parameter corresponding to a formal one that is exported; otherwise XXX is an exported global variable of the procedure. The message indicates that the updating of XXX, as a result of the procedure call, has no effect on any final values of exported variables of the calling subprogram. Where the ineffective assignment is expected (e.g. calling a supplied procedure that returns more parameters than are needed for the immediate purpose), it can be a useful convention to choose a distinctive name, such as "Unused" for the actual parameter concerned. The message "Assignment to Unused is ineffective" is then self-documenting.

!!! Flow Error :53: The package initialization of XXX is ineffective.
Here XXX is an own variable of a package, initialized in the package initialization. The message states that XXX is updated elsewhere, before being read.

!!! Flow Error :54: The initialization at declaration of XXX is ineffective.
Issued if the value assigned to a variable at declaration cannot affect the final value of any exported variable of the subprogram in which it occurs because, for example, it is overwritten before it is used.

8.3.3 Invariant conditions and stable exit conditions

!!! Flow Error :22: Value of expression is invariant.
The expression is either a case expression or a condition (Boolean-valued expression) associated with an if-statement, not contained in a loop statement. The message indicates that the expression takes the same value whenever it is evaluated, in all program executions. Note that if the expression depends on values obtained by a call to another other subprogram then a possible source for its invariance might be an incorrect annotation on the called subprogram.

!!! Flow Error :40: Exit condition is stable, of index 0.

!!! Flow Error :40: Exit condition is stable, of index 1.

!!! Flow Error :40: Exit condition is stable, of index greater than 1.
In these cases the (loop) exit condition occurs in an iteration scheme, an exit statement, or an if-statement whose (unique) sequence of statements ends with an unconditional exit statement - see the SPARK Definition. The concept of loop stability is explained in Section 4.4 of Appendix A. A loop exit condition which is stable of index 0 takes the same value at every iteration around the loop, and with a stability index of 1, it always takes the same value after the first iteration. Stability with indices greater than 0 does not necessarily indicate a program error, but the conditions for loop termination require careful consideration.

!!! Flow Error :41: Expression is stable, of index 0.

!!! Flow Error :41: Expression is stable, of index 1.

!!! Flow Error :41: Expression is stable, of index greater than 1.
The expression, occurring within a loop, is either a case expression or a condition (Boolean-valued expression) associated with an if-statement, whose value determines the path taken through the body of the loop, but does not (directly) cause loop termination. Information flow analysis shows that the expression does not vary as the loop is executed, so the same branch of the case or if statement will be taken on every loop iteration. An Index of 0 means that the expression is immediately stable, 1 means it becomes stable after the first pass through the loop and so on. The stability index is given with reference to the loop most closely-containing the expression. Stable conditionals are not necessarily an error but do require careful evaluation; they can often be removed by lifting them outside the loop.

8.3.4 Discrepancies between specified dependency relations and executable code

The dependency relation provided with a subprogram specification (explicitly in the case of a procedure, implicitly for a function), effectively names the subprogram imports and exports, and states which (initial values of) imports may be used in deriving (the final value of) each export. The SPARK Examiner also computes the import-export dependency relation, from the executable code, and compares the specified and actual relations. Any discrepancies are described by messages of the following kinds.

The first two messages indicates specified dependencies which are not found in the code implementation. These are unconditional errors.

!!! Flow Error :50: YYY is not derived from the imported value(s) of XXX.
The item before "is not derived ..." is an export or function return value and the item(s) after are imports of the subprogram. The message indicates that a dependency, stated in the dependency relation (derives annotation) or implied by the function signature is not present in the code. The absence of a stated dependency is always an error in either code or annotation.

!!! Flow Error :50: The imported value of XXX is not used in the derivation of YYY.

The variable XXX, which appears in the dependency relation of a procedure subprogram, as an import from which the export YYY is derived, is not used in the code for that purpose. YYY may be a function return value. This version of the message has been retained for backward compatibility.

The following kinds of messages are used to report dependencies found in the code of a subprogram, which do not appear in its specified dependency relation. These are conditional errors only: the dependencies found in the code may be attributable to paths that are not executable.

— Warning :601: YYY may be derived from the imported value(s) of XXX.
Here the item on the left of "may be derived from ..." is an exported variable and the item(s) on the right are imports of a procedure subprogram. The message reports a possible dependency, found in the code, which does not appear in the specified dependency relation (derives annotation). The discrepancy could be caused by an error in the subprogram code which implements an unintended dependency. It could also be in an error in the subprogram derives annotation which omits a necessary and intended dependency. Finally, the Examiner may be reporting a false coupling between two items resulting from a non-executable code path or the sharing of disjoint parts of structured or abstract data (e.g one variable writing to one element of an array and another variable reading back a different element). Unexpected dependencies should be investigated carefully and only accepted without modfiication of either code or annotation if it is certain they are of "false coupling" kind.

— Warning :601: The imported value of XXX may be used in the derivation of YYY.
Here first item is an import and the second is an export of a procedure subprogram. The message reports a possible dependency, found in the code, which does not appear in the specified dependency relation. This version of the message has been retained for backward compatibility.

— Warning :602: The undefined initial value of XXX may be used in the derivation of YYY.
Here XXX is a non-imported variable, and YYY is an export, of a procedure subprogram.

There are two formats for messages 50 and 601. In the first format presented here if there is more than one variable XXX which affects YYY, and does not conform to the specified dependency relation, they will appear on the same error message. Setting the switch /original_flow_errors brings up a separate message for each variable in the second (original) format.

8.3.5 Violation of restriction on imported-only variables

!!! Flow Error :34: The imported, non-exported variable XXX may be redefined.
The updating of imported-only variables is forbidden under all circumstances.

8.3.6 Supplementary error messages

The following supplementary messages are issued, to assist error diagnosis. The meanings of these messages are evident.

!!! Flow Error :30: The variable XXX is imported but neither referenced nor exported.

!!! Flow Error :31: The variable XXX is exported but not (internally) defined.

!!! Flow Error :32: The variable XXX is neither imported nor defined.

!!! Flow Error :33: The variable XXX is neither referenced nor exported.

!!! Flow Error :35: Importation of the initial value of variable XXX is ineffective.
The meaning of this message is explained in Section 4.2 of Appendix A.

8.3.7 Inconsistencies between abstract and refined dependency relations

!!! Flow Error :1: The previously stated updating of XXX has been omitted.
XXX occurred as an export in the earlier dependency relation but neither XXX nor any refinement constituent of it occurs in the refined dependency relation.

!!! Flow Error :2: The updating of XXX has not been previously stated.
A refinement constituent of XXX occurs as an export in the refined dependency relation but XXX does not occur as an export in the earlier dependency relation.

!!! Flow Error :3: The previously stated dependency of the exported value of XXX on the imported value of YYY has been omitted.
The dependency of the exported value of XXX on the imported value of YYY occurs in the earlier dependency relation but in the refined dependency relation, no constituents of XXX depend on any constituents of YYY.

!!! Flow Error :4: The dependency of the exported value of XXX on the imported value of YYY has not been previously stated.
A refined dependency relation states a dependency of XXX or a constituent of XXX on YYY or a constituent of YYY, but in the earlier relation, no dependency of XXX on YYY is stated.

!!! Flow Error :5: The (possibly implicit) dependency of the exported value of XXX on its imported value has not been previously stated.
Either a dependency of a constituent of XXX on at least one constituent of XXX occurs in the refined dependency relation, or not all the constituents of XXX occur as exports in the refined dependency relation. However, the dependency of XXX on itself does not occur in the earlier dependency relation.

9 Verification condition generation

9.1 General description

VC generation is the generation of formulae whose proof demonstrates some property of the associated code. Certain additional errors can be detected during VC generation and these are described below.

9.2 Error messages

The following error messages may be issued during VC generation: the first one is displayed on the screen, and one of the following acts as a further explanation which is placed in the .VCG file.

!!! Error(s) detected by VC Generator.
This message is echoed to the screen if an "unrecoverable" error occurs which makes the generation of VCs for the current subprogram impossible. Another message more precisely identifying the problem will be placed in the .VCG file.

!!! Program has a cyclic path without an assertion.
SPARK generates VCs for paths between cutpoints in the code; these must be chosen by the developer in such a way that every loop traverses at least one cutpoint. If the SPARK Examiner detects a loop which is not broken by a cutpoint, it cannot generate verification conditions for the subprogram in which the loop is located, and instead, issues this warning. This can only be corrected by formulating a suitable loop-invariant assertion for the loop and including it as an assertion in the SPARK text at the appropriate point.

!!! Unable to create output files.
This message is echoed to the screen if the Examiner is unable to create output files for the VCs or PFs being generated (for instance, if the user does not have write permission for the current directory).

!!! Unexpected node kind in main tree.
This message indicates corruption of the syntax tree being processed by the VC Generator. It should not be seen in normal operation. Please contact Praxis High Integrity Systems if it is produced.

!!! Warning, output file name has been truncated.
Echoed to the screen if an output file name is longer than the limit imposed by the operating system and has been truncated. Section 4.6 describes how the output file names are constructed. If this message is seen there is a possibility that the output from two or more subprograms will be written to the same file name, if they have a sufficiently large number of characters in common.

10 Static limits and associated error messages

The Examiner is w ritten in SPARK and therefore does not make use of access types and dynamic data structures. Because of this it contains a number of data structures of fixed sizes (see the Release Note for the actual table sizes).

If any of these fixed limits is exceeded the Examiner will stop, displaying the message "Internal static tool limit reached" on the terminal. Preceding this will be the cause, for example "Syntax tree overflow". The screen will also echo the consumption of the principal tables in these circumstances. The solution in all such cases is to move to a larger Examiner: three sizes are provided in the distribution kit. If the capacity of the largest Examiner is exceeded then please contact Praxis High Integrity Systems for advice.

A similar form of message will be seen if the tool reaches an operating system limit such as the number of files that can be open at any time. In this case the message will be prefixed by "Operating system limit reached". Such limits must be attended to by the system manager of the system on which the Examiner is being operated. Praxis High Integrity Systems may be able to assist in identifying the parameter that requires alteration.

Fatal errors prefixed by "Unexpected internal error" indicate an internal anomaly in the Examiner; in the unlikely event that errors of this form occur, please contact Praxis High Integrity Systems.

Document Control and References

Praxis High Integrity Systems Limited, 20 Manvers Street, Bath BA1 1PX, UK.

File under

$CVSROOT/userdocs/Examiner_UM.doc (was S.P0468.73.70)

Changes history

Issue 0.1 (12th January 2000) First Draft created from v4.0 manual.

Issue 0.2 (13th June 2000) Second draft, believed complete.

Issue 1.0 (16th June 2000) Definitive, after formal review

Issue 1.1 (14th Feb 2001) Added cfrs: 810, 812, 824, 831, 833, 838, 852. Documented RealRTC and OptFlow

Issue 1.2 (28th March 2001) Added documentation of Version switch (CFR 879).

Issue 1.3 (13th July 2001) Start of updating for Release 6.0

Issue 1.4 (24th August 2001) Adds errors 724, 730, 750–754, 800–805.

Issue 1.5 (24th August 2001) Adds ability to specify Long_Integer attributes in target data file.

Issue 1.6 (31st August 2001) Correct Examiner, Simplifier, and Checker context diagram.

Issue 2.0 (31st October 2001) After review S.P0468.79.74

Issue 2.1 (7th January 2002) Added documentation of target configuration file (CFR 992)

Issue 2.11 (5th February 2002) Added target configuration file to Figure 1 (CFR 992)

Issue 2.12 (19th March 2002) Added /Help command line option

Issue 2.13 (15th May 2002) Modular subtypes are allowed, do remove error 802.

Issue 3.0 (3rd July 2002) Updated, after review, for Release 6.1.

Issue 3.1 (30th September 2002) Updated semantic error message list and minor details to match 6.2 Examiner, changed document title to 6.2 Examiner.

Issue 3.2 (19th November 2002) Update title for Examiner 6.3. No other changes.

Issue 4.0 (15th April 2003): Updated to new template format.

Issue 5.0 (5th June 2003): Changes to new template, final format.

Issue 5.1 (17th February 2004): Add /brief command line switch.

Issue 5.2 (17th May 2004): Allow record subtypes.

Issue 5.3 (24th August 2004): Update documentation of html switch.

Issue 5.4 (2nd November 2004): Add /rules command line switch and changed company name.

Issue 5.5 (6th December 2004): Change to error messages 50 and 601, addition of /original_flow_errors switch.

Issue 5.6 (9th December 2004): Removed /vcg switch from options.

Issue 5.7 (17th December 2004): Regenerated error messages.

Issue 5.8 (4th January 2005): Definitive issue following review S.P0468.79.88

Issue 5.9 (5th July 2005): New warning 9 for hidden exception handlers.

Issue 6.0 (10th August 2005): Include warning control keyword for semantic notes 3 and 4.

Issue 6.1 (14th November 2005): Updated following changes to /warn /nowarn and /nolisting options.

Issue 6.2 (5th January 2006): Updated for Examiner release 7.2d15.

Issue 7.3 (12th January 2006): Updated for Examiner release 7.3.

Changes forecast

Nil.

Document references