AdaControl User Guide

Last edited: 7 October 2014

Table of Contents

1 Introduction

AdaControl is an Ada rules controller. It is used to control that Ada software meets the requirements of a number of parameterizable rules. It is not intended to supplement checks made by the compiler, but rather to search for particular violations of good-practice rules, or to check that some rules are obeyed project-wide. AdaControl can also be handy to make statistics about certain usages of language features, or simply to search for the occurrences of particular constructs; its scope is therefore not limited to enforcing programming rules, although it is of course one of its main goals.

AdaControl is a commercial product of Adalog with professional grade support available. Getting support is highly recommended for industrial projects. Adacontrol can also be customized or extended to match your special needs, please refer to Section 1.2 [Support], page 3 or contact Adalog at `info@adalog.fr`.

1.1 Features

AdaControl analyzes a set of Ada units, according to parameterizable controls. Controls can be given from the command line, from a file, or interactively. There is a wide range of controls available. Some are quite simple (although very useful):

- Control physical layout of the program (Maximum line length, no use of tabulations...)
- Control occurences of special strings in comments (like TBD for "To Be Defined"), with full wildcarding.
- Use of features (goto statement, tasking, pointers, variables in package specifications...)
- Use of any declared entity, with full overloading resolution

Other rules are quite sophisticated:

- Control series of "if"..."elsif" that could be replaced by "case" statements
- Verify usage of declarations (variables that should be constant, variables read but not written...)
- Control declarations that could be moved to a more reduced, internal scope
- Limit the call depth of a program (and diagnose recursive subprograms)
- Enforce a pattern that guarantees that exceptions are not handled silently
- Enforce a pattern for paired calls (like semaphore's "P" and "V") that guarantees that the closing call is always executed, even in presence of exceptions.
- Check that there is no aliasing between out parameters
- Ensure that no protected operation calls a potentially blocking operation

and much, much more... See Chapter 5 [Rules reference], page 37 for the complete reference for all possible controls.

AdaControl is very simple to use. It takes, as parameters, a list of units to process and a list of commands that define the controls to apply. The complete syntax of the commands is described in chapter Chapter 4 [Command language reference], page 27.

AdaControl produces messages to the standard output, unless redirected. Several levels of messages are defined (i.e. error or found), depending on the kind of the control (i.e. check or search).

Rules can be locally disabled for a part of the source code, and various options can be passed to the program.

Ex:

Given the following package:

```
package Pack is
   pragma Pure (Pack);
   ...
end Pack;
```

The following command:

```
adactl -l "search pragmas (pure)" pack
```

produces the following result (displayed to standard output):

```
pack.ads:2:4: Found: PRAGMAS: use of pragma Pure
```

AdaControl integrates nicely in environments such as GPS (see Section 3.6 [Running AdaControl from GPS], page 17), AdaGide (see Section 3.7 [Running AdaControl from AdaGide], page 22), or emacs (see Section 4.2.1 [Control kinds and report messages], page 28). In those environments, you can run AdaControl from menus or by just clicking on a button!

1.2 Support

1.2.1 Commercial support

Adalog provides commercial support for AdaControl. Support includes the following benefits:

- Help with installation procedures.
- Explanations regarding the use of the tool, and help for translating coding standards into AdaControl rules.
- Dedicated account into our MantisBT system for priority handling of problem reports.
- Correction of problems encountered in the use of AdaControl. Pre-releases versions of AdaControl are provided for each corrected problem.
- Access to beta-versions before they are released
- Keeping in sync customer's own custom rules with the latest version of AdaControl.
- Reduced rate for on-demand development of custom rules.
- Priority consideration of enhancement requests. Satisfying enhancement requests is not part of the support contract; however, Adalog is constantly improving AdaControl, and suggestions originating from supported customers are given a high priority in our todo list.

Adalog cannot correct problems whose origin is due to compiler bugs or defects in the implementation of ASIS (contact your compiler provider for support on these problems). However, Adalog will do its best effort to find workarounds for such problems.

In addition, Adalog can provide various services:

- Custom improvements to AdaControl, including application-specific rules;
- consulting services for defining coding standards;

- consulting services in all areas related to Ada, real-time, compilation, etc. See Adalog's site for details.

For pricing information about support contract and other services, please contact info@adalog.fr.

1.2.2 Other support

There is a Wiki for questions about AdaControl at https://sourceforge.net/p/adacontrol/wiki/Home/. This is the place to ask for information, make suggestions, or get help from the community.

For problem reports, please use our MantisBT system at http://sourceforge.net/apps/mantisbt/adacontrol.

1.2.3 Your support to us, too!

If you enjoy AdaControl, there are several things you can do to help us continue and improve this nice project.

- Rate it, or even better post a review, on the SourceForge review page
- Click "I use it" from AdaControl's home page.
- Rate it on AdaControl's Ohloh page
- Get a support contract, or encourage your company, your friends, or anybody else to get a support contract!
- Provide good ideas, new rules, suggestions for improvements...

And remember: developing AdaControl is an expensive effort (according to Ohlo's COCOMO model, it is worth 13 man.year of development). We need support from our users to keep it running!

1.3 History

The development of AdaControl was initially funded by Eurocontrol (http://www.eurocontrol.int), which needed a tool to help in verifying the million+ lines of code that does Air Traffic Flow Management over Europe. Because it was felt that such a tool would benefit the community at-large, and that further improvements made by the community would benefit Eurocontrol, it was decided to release AdaControl as free software. Later, Eurocontrol, Belgocontrol, Ansaldo (formerly CSEE-Transport), and SAGEM-DS sponsored the development of more rules.

The requirements for AdaControl were written by Philippe Waroquiers (Eurocontrol-Brussels), who also conducted extensive testing of AdaControl over the Eurocontrol software. The software was developped by Arnaud Lecanu and Jean-Pierre Rosen (Adalog). Rules, improvements, etc. were contributed by Pierre-Louis Escouflaire (Adalog), Alain Fontaine (ABF consulting), Richard Toy (Eurocontrol-Maastricht), and Isidro Ilasa Veloso (GMV). AdaGide support and improvement of icons were contributed by Gautier de Montmollin. Emmanuel Masker (Alstom) and Yannick Duchene contributed to GPS integration.

See file HISTORY for a description of the various versions of AdaControl, including enhancements of the current version over the previous ones. Users of a previous version are warned that the rules are not 100% upward-compatible: this is necessary to make the rules

more consistent and easier to use. However, the incompatibilities are straightforward to fix and should affect only a very limited number of files. See Appendix C [Non upward-compatible changes], page 152 for details.

1.4 References

1. "On the benefits for industrials of sponsoring free software development", *Ada User Journal*, Volume 26, n 4, december 2005

   http://www.adalog.fr/publicat/Free-software.pdf

2. "A Comparison of Industrial Coding Rules", *Ada User Journal*, Volume 29, n 4, december 2008

   http://www.adalog.fr/publicat/coding-rules.pdf

3. "A Methodology for Avoiding Known Compiler Problems Using Static Analysis", *proceedings of the ACM SIGAda Annual International Conference (SIGAda 2010)*

   http://www.adalog.fr/publicat/compiler-probs.pdf

2 Installation

Like any ASIS application, AdaControl can be run only if the compiler available on the system has exactly the same version as the one used to compile AdaControl itself. The executable distribution of AdaControl will work only with GNAT version GPL 2014, as distributed by AdaCore. If you are using any other version, please use the source distribution of AdaControl and compile it as indicated below.

Another reason for using the source distribution of AdaControl is that the user may not be interested in all provided rules. It is very easy to remove some rules from AdaControl to increase its speed. See [Customizing AdaControl], page 8.

2.1 Building AdaControl from source

This section is only for the source distribution of AdaControl. If you downloaded an executable distribution (and are using the latest version of GNAT GPL), you may skip to the next section.

2.1.1 Getting the correct version of the sources for your Gnat version

ASIS is continuously evolving to support Ada-2005/2012 features, and so is AdaControl. As a consequence, the full set of features of AdaControl is supported only with the latest versions of Gnat, namely GnatPRO 7.2.0 and GnatGPL-2013 (and higher). We refer to these versions as the "new Gnat", and we encourage all users to use these versions.

Some user may however need to use an older version of Gnat. We provide also a version of AdaControl that is compatible with versions GnatPRO 7.0.x and GnatGPL2011 and older (before some incompatible -but necessary- changes in ASIS happened). We refer to these versions as the "old Gnat".

The release whose distribution files start with "adactl" is for the new Gnat, and the one whose distribution files start with "adactl-old" is for the old-gnat. Both versions provide the same features, except that controls related to Ada-2012 (or that depend on new features of ASIS) are not available in the old-gnat version. Moreover, the old-gnat version is now frozen, and will not receive any new features or improvements in the future, unless requested by a supported customer (such requests will be honoured as part of the support contract). See Section 1.2 [Support], page 3 for information on becoming a supported user.

Note that intermediate releases of Gnat (GnatPRO-7.1.x, GnatGPL2012) are not fully compatible with either of these distribution. Depending on exact version, problems may range from compilation errors to incorrect results in some rare (Ada 2012) cases. Compatible sources can be obtained from the Git repository of AdaControl on SourceForge (`http://adacontrol.sourceforge.net`). We will be happy to help our supported customers who must use one of these versions.

2.1.2 Prerequisites

The following software must be installed in order to compile AdaControl from source:

- A GNAT compiler, any version (but please consider [Getting the correct version of the sources for your Gnat version], page 6 above). Note that the compiler must also be available on the machine in order to run AdaControl (all ASIS application need the compiler).

- ASIS for GNAT

Make sure to have the same version of GNAT and ASIS. The version used for running AdaControl must be the same as the one used to compile AdaControl itself.

2.1.3 Build with installer (Windows)

Run the installer (`adactl_src-setup.exe`). This will automatically build and install AdaControl, no other installation is necessary.

2.1.4 Build with project file

Simply go to the `src` directory and type:

```
gnatmake -Pbuild.gpr
```

You're done!

Caveat (*old gnat only*): Due to a bug in some versions, if you are using GNATPro 6.1.2 and above, you must set the variable GNAT_FIX to 1; i.e. invoke the command as:

```
gnatmake -Pbuild.gpr -XGNAT_FIX=1
```

2.1.5 Build with Makefile

The previous method may fail if Asis is not installed in an usual place. As an alternative method, it is possible to build AdaControl with a regular Makefile.

The file `Makefile` (in directory `src`) should be modified to match the commands and paths of the target system. The following variables are to be set:

- ASIS_TOP
- ASIS_INCLUDE
- ASIS_OBJ
- ASIS_LIB
- RM
- EXT

How to set these variables properly is documented in `Makefile`. See also the compilation options in this file; a change is needed if you are using GNATPro 6.1.2 and above.

Then, run the make command:

```
$ cd src
$ make build
```

It is also possible to delete object files and do other actions with this "Makefile", run the following command to get more information:

```
$ make help
```

NOTE: Building AdaControl needs the "make" command provide with GNAT; it works both with WIN32 shell and UNIX shell.

2.1.6 Build with a compiler other than GNAT

It should be possible to compile AdaControl with other compilers than GNAT, although we didn't have an opportunity to try it. If you have another compiler that supports ASIS, note that it may require some easy changes in the package `Implementation_Options` to give proper parameters to the `Associate` procedure of ASIS. Rules that need string pattern matchings need the package `Gnat.Regpat`. If you compile AdaControl with another compiler, you can either port `Gnat.Regpat` to your system, or use a (limited) portable implementation of a simple pattern matching (package `String_Matching_Portable`). Edit the file `string_matching.ads` and change it as indicated in the comments. No other change should be necessary.

Alternatively, if you are using another compiler, you can try and compile your program with GNAT just to be able to run AdaControl. However, compilers often differ in their support of representation clauses, which can cause your program to be rejected by GNAT. In that case, we provide a sed script to comment-out all representation clauses; this can be sufficient to allow you to use AdaControl. See Section 3.8.3 [unrepr.sed], page 23.

2.1.7 Testing AdaControl

Testing AdaControl needs a UNIX shell, so it works only with UNIX systems. However, it is possible to run the tests on a WIN32 system by using an UNIX-like shell for WIN32, such as those provided by CYGWIN or MSYS. To run the tests, enter the following commands:

```
$ cd test
$ ./run.sh
```

All tests must report PASSED. If they don't, it may be due to one of the following issues:

- You are using an old version of GNAT. AdaControl runs without any known problem (and it has been checked against the whole ACATS) only with the latest GNATPro and GNATGPL versions; earlier versions are known to have bugs and unimplemented features that will not allow AdaControl to run correctly in some cases. We strongly recommend to always use the most recent version of GNAT.
- You run an old-gnat version of AdaControl with GNATPro (6.1.2 and above) and you forgot to specify the "-XGNAT_FIX=1" option. See [Build with project file], page 7.
- It may happen that the test `tfw_check` reports "FAILED" on some sytems, because it depends on the order in which the operating system lists files. If this happens, try (from the `test` directory):

  ```
  diff res/tfw_check.txt ref/
  ```

  If the only difference is that some lines are at different places, the test is OK.

2.1.8 Customizing AdaControl

If there are some rules that you are not interested in, it is very easy to remove them from AdaControl:

1. In the `src` directory, edit the file `framework-plugs.adb`. There is a `with` clause for each rule (children of package `Rules`). Comment out the ones you don't want.
2. Recompile `framework-plugs.adb`. There will be error messages about unknown procedure calls. Comment out the corresponding lines.
3. Compile AdaControl normally. That's all!

It is also possible to add new rules to AdaControl. If your favorite rules are not currently supported, you have several options:

1. If you have some funding available, please contact info@adalog.fr. We'll be happy to make an offer to customize AdaControl to your needs.
2. If you *don't* have funding, but have some knowledge of ASIS programming, you can add the rule yourself. We have made every effort to make this as simple as possible. Please refer to the AdaControl programmer's manual for details. If you do so, please send your rules to rosen@adalog.fr, and we'll be happy to integrate them in the general release of AdaControl to make them available to everybody.
3. If you have good ideas, but don't feel like implementing them yourself (nor financing them), please send a note to rosen@adalog.fr. We will eventually incorporate all good suggestions, but we can't of course commit to any dead-line in that case.

2.2 Installing AdaControl

All you need to run AdaControl is the executable named `adactl` under Linux or `adactl.exe` under Windows. In addition, `pfni` (or `pfni.exe` under Windows) is a convenient utility, required by the GPS support. See Section 3.8.1 [pfni], page 22.

If you downloaded the Windows installer executable version of AdaControl, simply run `adactl_exe-setup.exe`. This will install all the files in the recommended locations (as has been done with the Windows installer source version), including GPS support if you have GPS installed and/or AdaGide support if you have AdaGide installed.

If you built AdaControl from source without an installer, the executables are in the `src` directory of the distribution. If you downloaded an executable distribution, they are in the root directory of the distribution. Copy the executables to any convenient directory on your path; a good place, for example, is in the `bin` directory of your GNAT installation.

2.3 Installing support for GPS

Integration of AdaControl into GPS with all functionalities requires GPS version 4.2 or above (delivered since GNAT/GPL2008).

To add AdaControl support to GPS, copy the file `GPS/adacontrol.xml` into the `<GNAT_dir>/share/gprconfig` directory; copy all other files from the `GPS` directory into the `<GPS_dir>/share/gps/plug-ins` directory. Copy also HTML files from the `doc` directory into the `<GPS_dir>/share/doc/gps/html` to access AdaControl's guides from the "Help" menu of GPS.

2.4 Installing support for AdaGide

To add AdaControl support to AdaGide, copy the file `AdaControl.tdf` from the `AdaGide` directory into AdaGide's root directory. Note that AdaControl support requires AdaGide version 7.42 or above.

3 Program Usage

AdaControl is a command-line program, i.e. it is normally called directly from the system shell. *Options* are introduced by a "-" followed by a letter and can be grouped as usual. Some options take the following word on the command line as a value; such options must appear last in a group of options. *Parameters* are words on the command line that stand by themselves. Options and parameters can be given in any order.

The syntax for invoking AdaControl in regular mode is:

```
adactl [-deEirsTuvwx]
       [-p <project file>]     [-f <rules file>]    [-l <rules list>]
       [-o <output file>]      [-t <trace file>]    [-F <format>]
       [-S <statistics level>] [-m <warning limit>] [-M <message limit>]
       {<unit>[+|-<unit>]|[@]<file>} [-- <ASIS options>]
```

AdaControl can process Ada-2005 as well as Ada-95 programs, even if there are currently few Ada-2005 related controls - but we hope to improve that situation in the near future.

If you are using Ada-2005 (or Ada-2012) features, make sure that GNAT is set up for Ada-2005/2012 (this is the default for GNAT-GPL). Due to technical reasons, the -gnat05 option cannot be passed to the compiler in "compile on the fly" mode, but you can do any of the following:

- have a "gnat.adc" file in the current directory that contains a **pragma** `Ada_05;` (or **pragma** `Ada_12;`)
- put a **pragma** `Ada_05` (or **pragma** `Ada_12;`) on top of every compilation unit that uses Ada-2005/2012 features;
- generate the tree files manually (see Section 3.9.2 [Generating tree files manually], page 25) with the "-gnat05" (or "-gnat12)") option. Note that this allows you to pass any other GNAT option as well.

Note that if your program is pure Ada-95 and you are using a version of GNAT where Ada-2005 is the default (especially GNAT-GPL), and in the rare cases where your program would not compile in Ada-2005 mode (notably if you have a function that returns a task type), you can force Ada-95 the same way by using **pragma** `Ada_95` instead.

3.1 Command line parameters and options

3.1.1 Input units

Units to be processed are given as parameters on the command line. Note that they are Ada *compilation unit* names, not *file names*: case is not significant, and there should be no extension! Child units are allowed following normal Ada naming rules: `Parent.Child`, but be aware that specifying a child unit will automatically include its parent unit in the analysis. Subunits are processed during the analysis of the including unit; there is therefore no need to specify subunits explicitely. If you do specify a subunit explicitly, it will result in the whole enclosing unit being analyzed.

However, as a convenience to the user, units can be specified as file names, provided they follow the default GNAT naming convention. More precisely, if a parameter ends in ".ads" or ".adb", the unit name is extracted from it (and all "-" in the name are substituted with

“.”). File names can include a path; in this case, the path is automatically added to the list of directories searched (“-I” ASIS option). The file notation is convenient to process all units in a directory, as in the following example:

```
adactl -f my_rules.aru *.adb
```

In the unlikely case where you have a child unit called `Ads` or `Adb`, use the “-u” option to force interpretation of all parameters as unit names.

By default, both the specification and body of the unit are processed; however, it is possible to specify processing of the specification only by providing the “-s” option. If only file names are given, the “-s” option is assumed if all files are specifications (“.ads” files). It is not possible to specify processing of bodies only, since rules dealing with visibility would not work.

The “-r” option tells AdaControl to process (recursively) all user units that the specified units depend on (including parent units if the unit is a child unit or a subunit). Predefined Ada units and units belonging to the compiler’s run-time library are never processed.

Ex:

```
adactl -r -f my_rules.aru my_main
```

will process `my_main` and all units that `my_main` depends on. If `my_main` is the main procedure, this means that the whole program will be processed.

It is possible to specify more than one unit (not file) to process in a parameter by separating the names with “+”. Conversely, it is possible to specify units that are *not* to be processed, separated by “-”. When a unit is subtracted from the unit list, it is never processed even if it is included via the recursive option, and all its child and separate units are also excluded. This is convenient to avoid processing reusable components, that are not part of a project. For example, if you want to run AdaControl on itself, you should use the following command:

```
adactl -f my_rules_file.aru -r adactl-asis-a4g
```

This applies the rules from the file `my_rules_files.aru` to AdaControl itself, but not to units that are part of ASIS (units `Asis`, `A4G`, and their children) that would be found by the “-r” (recursive) option otherwise.

Alternatively, it is possible to provide units indirectly with a parameter consisting of an “@” followed by the name of a file. This file must contain a list of unit names (not files), one on each line. Only the first “word” of the line is considered, i.e. everything after the first blank is ignored. This can be useful to annotate unit names. All units whose names are given in the file will be processed. If a name in the file starts with “@”, it will also be treated as an indirect file (i.e. the same process will be invoked recursively). If a line in the file starts with “#” or “--”, it is ignored. This can be useful to temporarily disable the processing of some files or to add comments.

Ex:

```
adactl -f my_rules.aru @unit_file.txt
```

3.1.2 Commands

Commands specify which processing AdaControl should apply to units. See Chapter 4 [Command language reference], page 27 for a detailed description of all commands.

Commands can be given directly on the command line with the "-l" option. A commands list must be quoted with "".

Ex:

```
adactl pack.ads proc.adb -l "check instantiations (My_Generic);"
```

It is possible to pass several commands separated by ";", but as a convenience to the user, the last ";" may be omitted.

Commands can also be read from a file, whose name is given after the "-f" option (the ".aru" extension is taken by default). As a special case, if the file name is "-", commands are read from the standard input. This is intended to allow AdaControl to be pipelined behind something that generates commands; if you want to type commands directly to AdaControl, the interactive mode is more appropriate. See Section 3.4 [Interactive mode], page 15.

Ex:

```
adactl -f my_rules.aru proc.adb
```

Note that the "-l" and "-f" options are *not* exclusive: if both are specified, the commands to be performed include those in the file (first) and then those given on the command line.

3.1.3 Output file

Messages produced by controls are output to the output file; by default, it is the standard output, but it can be changed by specifying the "-o" option.

Ex:

```
adactl -f my_rules.aru -o my_output.txt proc.adb
```

If the output file exists, new messages are appended to it. This allows running AdaControl under several directories that make up the project, and gathering the results in a single file. However, if the "-w" option is given, AdaControl overwrites the output file if it exists.

All other messages, including syntax error messages, units processed (in verbose mode), and possible internal error mesages from AdaControl itself are output to the standard error file.

3.1.4 Output format

The "-F" option selects the output format. It must be followed by "Gnat", "Gnat_Short", "CSV", "CSV_Short", "Source", "Source_Short", or "None" (case insensitive). By default, the output is in "Gnat" format. See Section 4.2.1 [Control kinds and report messages], page 28 for details.

The "-S" option selects which statistics are output after each run. It must be followed by a value in the range 0..3. See Section 4.2.1 [Control kinds and report messages], page 28 for details on the various statistics levels.

The "-T" option prints a summary of timing at the end of each run. This indicates how long (in real-time seconds) was spent in processing each rule.

Ex:

```
adactl -F CSV -S 2 -f my_rules.aru -o my_output.csv proc.adb
```

3.1.5 Output limits

The "-m" and "-M" options are used to limit the output of AdaControl. These options are followed by an integer value that specifies the maximum number of error messages ("-m") or warning and error messages ("-M"). If the value is omitted, a previous limitation (comming for example from a command file) is cancelled.

If the indicated number of messages is exceeded during a run, AdaControl stops immediately.

3.1.6 Project files

An emacs project file (the file with a ".adp" extension used by the Ada mode of Emacs) can be specified with the " -p" option. AdaControl will automatically consider all the directories mentioned in "src_dir" lines from the project file.

Ex:

```
adactl -f my_rules.aru -p proj.adp proc.adb
```

Note that AdaControl does not accept ".gpr" project files, because ASIS does not currently accept the "-P" option like other GNAT commands do. However, when run from GPS, the interface will automatically use the source directories from the current (root) project (unless you have explicitely set a ".adp" file in the switches Section 3.6.3 [AdaControl switches], page 19).

If you have a project that uses ".gpr" project files and you want to run AdaControl from the command line (not from GPS), you can generate a ".adp" project file from a ".gpr" project file from within GPS, by using the "Tools/AdaControl/Generate .adp project" menu. See Section 3.6 [Running AdaControl from GPS], page 17. Alternatively, it is also possible to use GPS project files by generating the tree files manually. see Section 3.9.2 [Generating tree files manually], page 25 for details.

3.1.7 Local disabling control

The "-i" option tells AdaControl to ignore disabling markers in Ada source code (see Section 4.2.4 [Disabling controls], page 30); i.e. all controls will be performed, regardless of the presence of disabling markers. This is equivalent to the command "`set ignore ON;`". Note that if you you have many messages, setting this option can speed-up AdaControl considerably. It is therefore advisable to always set this option when you know that there is no disabling marker in your source code.

The "-j" option tells AdaControl to invert the meaning of disabling markers, i.e. only messages marked as disabled will be printed. This is useful to check which messages have been disabled. This is equivalent to the command "`set ignore INVERTED;`".

3.1.8 Verbose and debug mode

In the default mode, AdaControl displays only messages from triggered controls. It is possible to get more information with the verbose option ("-v"). In this mode, AdaControl displays a a progress indicator and unit names as they are processed, and its global execution time when it finishes. Note that the progress indicator includes an indication of the run number if there are more than one "go" command.

The "-d" option enables debug mode. This mode provides more information in case of an internal program error, and is of little interest for the casual user. Note that if you hit

Ctrl-C in debug mode, AdaControl aborts with a message telling the currently active rule and module. This can be useful if you suspect AdaControl to be stuck in an infinite loop.

In debug mode, AdaControl may also, in rare occasions (and only with some versions of GNAT), display ASIS "bug boxes"; this does not mean that something went wrong with the program, but simply that an ASIS failure was properly recovered by AdaControl.

Output of the messages printed by the "-d" option can be directed to a "trace" file (instead of being printed to the standard error file). This is done by the "-t" option, which must be followed by the file name. If the trace file exists, new messages are appended to it.

3.1.9 Treatment of warnings

The "-e" option tells AdaControl to treat warnings as errors, i.e. to report a return code of 1 even if only "search" controls were triggered. See Section 3.2 [Return codes], page 14. It does not change the messages however.

Conversely, the "-E" option tells AdaControl to *not* report warnings at all, i.e. only errors are reported. However, if you ask for statistics, the number of warning messages is still counted. See Section 4.2.1 [Control kinds and report messages], page 28.

3.1.10 Exit on error

If an internal error is encountered during the processing of a unit, AdaControl will continue to process other units. However, if the "-x" option is given, AdaControl will stop on the first error encountered. This option is mainly useful if you want to debug AdaControl itself (or your own rules). See Section 3.10 [In case of trouble], page 26.

Ex:

```
adactl -x -f my_rules.aru proc.adb
```

3.1.11 ASIS options

Everything that appears on the command line after "--" will be treated as an ASIS option, as described in the ASIS user manual.

Casual users don't need to care about ASIS options, except in one case: if you are running AdaControl from the command line (not from GPS), and if the units that you are processing reference other units whose source is not in the same directory, AdaControl needs to know how to access these units (as GNAT would). This can be done either by using an Emacs project file with the "-p" option (see Section 3.1.6 [Project files], page 13), by putting the appropriate directories into the ADA_INCLUDE_PATH environment variable, or by passing "-I" options to ASIS.

It is possible to pass one or several "-I" options to ASIS, to provide other directories where sources can be found. The syntax is the same as the "-I" option for GNAT.

Other ASIS options, like the "-Cx" and/or "-Fx" options, can be specified. Most users can ignore this feature; however, specifying these options can improve the processing time of big projects. See Section 3.9 [Optimizing Adacontrol], page 23.

3.2 Return codes

In order to ease the automation of controlling programs with shell scripts, AdaControl returns various error codes depending on how successful it was. Values returned are:

- 0: At most "search" controls (i.e. warnings) were triggered (no control at all with "-e" option)
- 1: At least one "check" control (i.e. error) was triggered (or at least one "search" or "check" control with "-e" option)
- 2: AdaControl was not run due to a syntax error in the rules or in the specification of units.
- 10: There was an internal failure of AdaControl.

3.3 Environment variable and default settings

If the environment variable "ADACTLINI" is set, its content is taken as a set of commands (separated by semi-colons) that are executed before any other command. Although any command can be specified, this is intended to allow changing default settings with "set" commands. See Section 4.3.6 [Set command], page 33.

For example, you can set ADACTLINI to "set format Gnat_Short" if you prefer having you messages in short format rather than the (default) long format.

3.4 Interactive mode

The "-I" option tells AdaControl to operate interactively. In this mode, commands specified with "-l" or "-f" options are first processed, then AdaControl prompts for commands on the terminal. Note that the "quit" command (see Section 4.3.2 [Quit command], page 32) is used to terminate AdaControl.

The syntax of commands run interactively is exactly the same as the one used for files; especially, each command must be terminated with a ";". Note that the prompt ("Command:") becomes "........:" when AdaControl requires more input because a command is not completely given, and especially if you forget the final ";".

As with files, it is possible to give several commands on a single line in interactive mode. If a command contains syntax errors, all "go" commands (see Section 4.3.1 [Go command], page 32) on the same line are temporarily disabled. Other commands that do not have errors are normally processed however.

The interactive mode is useful when you want to do some analysis of your code, but don't know beforehand what you want to control. Since the ASIS context is open only once when the program is loaded, queries will be much faster than running AdaControl entirely with a new query given in a "-l" option each time. It is also useful to experiment with AdaControl, and to check interactively commands before putting them into a file.

3.5 Other execution modes

In addition to normal usage, AdaControl features special options to ease its use; no Ada unit is analyzed when using these options.

3.5.1 Getting help

The "-h" option provides help about Adacontrol usage. If the "-h" option is given, no other option is analyzed and no further processing happens.

Syntax:

```
adactl -h [<keyword> | <rule name> | variables ["<pattern>"] ...]
<keyword> ::= all | commands | license | list | options | rules | version
```

The "-h" option without parameter displays a help message about usage of the AdaControl program, the various options, and the rule names.

Otherwise, the "-h" must be followed by one or several keywords or rule names (case irrelevant); its effect is:

- <rule name>: if <rule name> is exactly the name of rule, display the help message for the indicated rule. Otherwise, <rule name> is interpreted as a pattern, and help messages for all rules that match the pattern is displayed. Patterns are given using the full Regexp syntax. see Appendix B [Syntax of regular expressions], page 149 for details.
- "variables" lists the values of all variables whose name matches <pattern>, or all variables if there is no <pattern>. Patterns are given using the full Regexp syntax. see Appendix B [Syntax of regular expressions], page 149 for details.
- "all": display the help message for all rules.
- "commands": display a summary of all commands
- "license": display the license information
- "list": display the names of all rules (note that "rules" also displays the list of rules, in a prettier format; the "list" option is mainly useful for the integration of AdaControl into GPS).
- "options": display help about the command-line options
- "rules": display the names of all rules.
- "version": display AdaControl and ASIS implementation version numbers.

Ex:

```
adactl -h pragmas Unnecessary_Use_Clause
adactl -h all
adactl -h version license
adactl -h stat
```

Note in the last example that "stat" is not the name of a rule; it is therefore interpreted as a pattern, and help will be displayed for all rules that include the string "stat" in their name. This can be very convenient to retrieve the name of a rule if you don't remember exactly how it is spelled.

3.5.2 Checking commands syntax

The "-C" option is used to check syntax of commands without executing any control.

Syntax:

```
adactl -C [-dv] [-f <rules file>] [-l <rules list>]
```

In this mode, AdaControl simply checks the syntax of the commands provided with the "-l" option, or of the commands provided in the file named by the "-f" option (at least one of these options must be provided). No other processing will happen.

AdaControl will exit with a return code of 0 if the syntax is correct, and 2 if any errors are found. A confirming message that no errors were found is output if the "-v" option is given.

This option is especially useful when you have modified a rules file, before trying it on many units. The way AdaControl works, it must open the ASIS context (a lengthy operation) *before* analyzing the rules. This option can therefore save a lot of time if the rules file contains errors.

3.5.3 Generating a units list

The "-D" options produces a list of units that can be reused as an indirect file in later runs. Syntax:

```
adactl -D [-rsvw] [-o <output file>] [-p <project file>]
          {<unit>[+|-<unit>]|[@]<file>} [-- <ASIS options>]
```

In this mode, AdaControl outputs the list of units that would be processed. It is especially useful when used with the "-r" option and given the main unit name, since it will then generate the whole list of dependent units (hence the name "D").

This list can be directed to a file with the "-o" option (if the file exists, it won't be overwritten unless the "-w" option is specified). This file can then be used in an indirect list of units. See Section 3.1.1 [Input units], page 10. Note that it is more efficient to create the list of units once and then use the indirect file than to specify all applicable units or use the "-r" option each time AdaControl is run.

3.6 Running AdaControl from GPS

If you want to use AdaControl from GPS, make sure you have copied the necessary files into the required places. See Section 2.2 [Installing AdaControl], page 9.

AdaControl integrates nicely into GPS, making it even easier to use. It can be launched from menu commands, and parameters can be set like any other GPS project parameters. When run from within GPS, AdaControl will automatically retrieve all needed directories from the current GPS project.

After running AdaControl, the "locations" panel will open, and you can retrieve the locations of errors from there, just like with a regular compilation. Errors will be marked in red in the source, warning will be marked orange, and you will have corresponding marks showing the places of errors and warnings in the speedbar. Note that AdaControl errors appear under the "AdaControl" category, but if there were compilation errors, they will appear under the "Compilation" category. Final counts from "count" control kinds will appear under the "Counts summary" category, and statistics under the "Statistics" category.

3.6.1 The AdaControl menu and buttons

GPS now features an "AdaControl" menu, with several submenus:

- "Control Current File (rules file)" runs AdaControl on the currently edited file, with rules taken from the current rules file; this menu is greyed-out if no rules file is defined, if no file window is currently active, or if the associated language is not "Ada". The name of the rules file can be set from the "Library" tab from the "Project/Edit Project Properties" menu.
- "Control Root Project (rules file)" runs AdaControl on all units that are part of the root project, with rules taken from the current rules file; this menu is greyed-out if no

rules file is defined. The name of the rules file can be set from the "Library" tab from the "Project/Edit Project Properties" menu.

- "Control Units from List (rules file)" runs AdaControls on units given in a indirect file, with rules taken from the current rules file. This menu is greyed-out if no rules file is defined or if no indirect file is defined. The name of the rules file and of the indirect file can be set from the "Library" tab from the "Project/Edit Project Properties" menu.
- "Control Current File (interactive)" runs AdaControl on the currently edited file, with a rule asked interactively from a pop-up; this menu is greyed-out if no file window is currently active, or if the associated language is not "Ada".
- "Control Root Project (interactive)" runs AdaControl on all units that are part of the root project, with a rule asked interactively from a pop-up.
- "Control Units from List (interactive)" runs AdaControls on units given in a indirect file, with a rule asked interactively from a pop-up. This menu is greyed-out if no indirect file is defined. The name of the indirect file can be set from the "Library" tab from the "Project/Edit Project Properties" menu.
- "Check Rules File" checks the syntax of the current rules file. This menu is deactivated if the current window does not contain an AdaControl rules file.
- "Open Rules File" opens the rules file. This menu is deactivated if there is no current rules file defined.
- "Open Units File" opens the units file. This menu is deactivated if there is no current units file defined.
- "Create units file" creates a text file containing all units (not files) names from the current root project. This file is appropriate as an indirect file for the "... from list" commands.
- "Create .adp project" creates an Emacs-style project file from the current GPS project, which can be used with the "-p" option if you want to run AdaControl from the command line. This file has the same name as the current GPS project, with a ".adp" extension. See Section 3.1.6 [Project files], page 13.
- "Delete Tree Files" removes existing tree files from the current directory. This is convenient when AdaControl complains that the tree files are not up-to-date. Note that you can set the preferences for automatic deletion of tree files after each run (see below). Note that the name of this menu is changed to "Delete Tree and .ali Files" if you have chosen to delete .ali files in the preferences (see below).
- "Load results file" loads in the location window the result file obtained from a previous run of AdaControl. The file must have been produced with the "Gnat" or "Gnat_Short" format. See Section 4.2.1 [Control kinds and report messages], page 28.

There are also two buttons representing Lady Ada in a magnifier glass in the toolbar, one with a red question mark in the background. These buttons launch AdaControl, by default on the file currently being edited; however, you can change this behaviour from the preferences to control either files from a list, or all files from the project. The button without the question mark uses rules from the current rules file, while the one with the question mark asks for the control to apply interactively.

Here are some tips about using the "interactive" menus (or the button with the question mark):

- When you use the "interactive" menus several times, the previously entered command(s) is used as a default.
- You can enter any command from AdaControl's language in the dialog; you can even enter several commands separated by ";".
- Especially, if you want to run AdaControl with a rules file that is not the one defined by the switches, you can use one of the "interactive" commands, and give "source <file name>" as the command.

3.6.2 Contextual menu

AdaControl adds two entries to the contextual menus (right click) of Ada files. They call the `pfni` utility on the current entity. See Section 3.8.1 [pfni], page 22. The entry "Print full name" displays the full name of the entity in simple form, while the entry "Print full name (with overloading)") prints it with overloading information. If the name refers to an entity which is initialized (or to a parameter with a default value), the initial value is printed. If the entity is a discrete type, its range is printed. If the entity is an array type, the ranges of its indices are printed.

This is convenient to find how to name entities in rule files. See Appendix A [Specifying an Ada entity name], page 145. It is also convenient to find where an entity is declared, and which of several overloaded entities is being referred to.

This is also convenient to find the actual value of a constant from anywhere in the program text, since the printed value is completely evaluated if it is a (static) expression.

3.6.3 AdaControl switches

The tab "switches" from the "Project/Edit Project Properties" menu includes a page for AdaControl, which allows you to set various parameters. Since the GPS interface analyzes the output of AdaControl, you should not set options directly in the bottom window of this page (the one that displays the actual options passed to AdaControl).

3.6.3.1 Files

This section controls the definition of various files used by AdaControl.

- "Rules file". This is the name of a file that contains the definition of the controls to be applied to your project. This file is required for all "control (rules file)" commands.
- "Units file". This is the name of a file that contains the list of units to be controlled. This file is required for all "control from list" commands.
- ".adp project file". This is the name of an emacs project file (.adp). If this name is not empty, AdaControl will use it instead of providing all libraries as "-I" options on the command line. This may be necessary if you have many libraries and the command line that launches AdaControl becomes too long. Note that this file can be created using the "AdaControl/Create .adp project" menu.

3.6.3.2 Processing

This section offers options that control how units are processed.

- "Recursive mode". This sets the "-r" option. See Section 3.1.1 [Input units], page 10.
- "Ignore local deactivation". This sets the "-i" option. See Section 3.1.7 [Local disabling control], page 13.

- "Process specs only". This sets the "-s" option. See Section 3.1.1 [Input units], page 10.
- "Compilation unit mode". This sets the "-u" option. See Section 3.1.1 [Input units], page 10.

3.6.3.3 Debug

This section controls the debugging options of AdaControl.

- "Debug messages". This sets the "-d" option. See Section 3.1.8 [Verbose and debug mode], page 13.
- "Halt on error". This sets the "-x" option. See Section 3.1.10 [Exit on error], page 14.

3.6.3.4 Output

This section offers options that control where and how the output of AdaControl is displayed.

- "Display only errors". This sets the "-E" option. See Section 3.1.9 [Treatment of warnings], page 14.
- "Warnings as errors". This sets the "-e" option. See Section 3.1.9 [Treatment of warnings], page 14.
- "Statistics". This sets the "-S" option from a pull-down menu. See Section 4.2.1 [Control kinds and report messages], page 28.
- "Send results to GPS". When checked (default), the output of AdaControl is sent to the "locations" window of GPS.
- "Send results to File". When checked, the output of AdaControl is sent to the file indicated in the box below.
- "Send results to File and GPS". When checked, the output of AdaControl is sent to the file indicated in the box below, and the content of the file is then automatically reloaded in the "locations" window of GPS. If this option is set, the file format is always "Gnat" (the file format option is ignored).
- "File name". This is the name of the file that will contain the results when sent to "File" or "File and GPS". If the results are sent to "File" and the file exists, AdaControl will ask for the permission to overwrite it. If the results are sent to "File and GPS", the result file is always overriden without asking.
- "File format". This is a pull-down menu that allows you to select the desired format when output is directed to a file ("-F" option). See Section 4.2.1 [Control kinds and report messages], page 28.

3.6.3.5 ASIS

This section controls the ASIS parameters passed to AdaControl. The content of the input field "ASIS options" is used in place of the standard ("-CA -FM") one.

Casual users don't need to change the default ASIS options. For more details, see Section 3.1.11 [ASIS options], page 14.

3.6.4 AdaControl preferences

There is an entry for AdaControl in the "edit/preferences" menu:

- "delete trees". If this box is checked, tree files are automatically deleted after each run of AdaControl. This avoids having problems with out-of-date tree files, at the expanse of

slightly slowing down AdaControl if you run it several times in a row without changing the source files.

- "Delete .ali files with tree files". If this box is checked, the ".ali" files in the current directory will also be deleted together with the tree files (either automatically if the previous box is checked, or when the "AdaControl/Delete Tree Files" menu is selected). This is normally what you want, unless the current directory is also used as the object directory for compilations; in the latter case, deleting ".ali" files would cause a full recompilation for the next build of the project.
- "Help on rule". This allows you to select how rule specific help (from the "Help/AdaControl/Help on rule" menu) is displayed. If you select "Pop-up", a summary of the rule's purpose and syntax is displayed in a pop-up. If you select "User Guide", the user guide opens in a browser at the page that explains the rule. (Caveat: due to a problem in GPS under Windows, the "User Guide" option may not work at all, or the browser will not find the right anchor; hopefully, this will be fixed in an upcomming release of GPS. No such problem under Linux).
- "Use separate categories". If this box is checked, there will be one category (i.e. tree in the locations window) for each rule type or label, otherwise all messages will be grouped under the single category "AdaControl". In practice, this means that with the box checked, messages will be sorted by rules first, then by files, while otherwise, the messages will be sorted by files first, then by rules. In any case, compilation errors appear under the "Compilation" category, final counts under the "Counts summary" category, and statistics under the "Statistics" category.
- "Auto save files". If this box is checked, all modified files are automatically saved without asking before running AdaControl. Otherwise, a dialog appears allowing the user to choose which files to save.
- "Buttons operate on". This defines the behaviour of the buttons. If "Current File" is selected, the buttons operate on the file being currently edited. If "Root Project" is selected, the buttons operate on all files that are part of the current project. If "Units from List" is selected, the buttons operate on all units from the units file.
- "Display AdaControl run". If this box is checked, the command line used to launch AdaControl and the output messages are displayed in the "Messages" window.
- "Max allowed error messages". If non zero, run will stop if the number of error messages exceeds this limit. See Section 3.1.5 [Output limits], page 13.
- "Max allowed messages (all kinds)". If non zero, run will stop if the number of error and warning messages exceeds this limit. See Section 3.1.5 [Output limits], page 13.

3.6.5 AdaControl language

If you check "AdaControl" in the "Languages" tab, GPS will recognize files with extension `.aru` as AdaControl rules files, and provide appropriate colorization.

3.6.6 AdaControl help

The AdaControl User Manual (this manual) and the AdaControl Programmer Manual are available from the "Help/AdaControl" menu of GPS.

The "Help on rule" entry displays the list of all rules; if you click on one of them, you get help for the particular rule. Depending on the setting of the "Help on rule" preference (see

above), it opens a pop-up that displays the rule(s) purpose and the syntax of its parameters, or opens the user guide at the appropriate location.

The "About" entry displays a popup with AdaControl's version number and license condition.

3.6.7 Caveat

GPS may crash when the output of a command is too big (i.e. hundreds of messages with AdaControl). If this happens, use the "preferences" menu to limit the number of messages.

3.7 Running AdaControl from AdaGide

If you want to use AdaControl from AdaGide, make sure you have copied the necessary file into the required place. See Section 2.2 [Installing AdaControl], page 9. Note that AdaGide does not have all the parameterization facilities of sophisticated environments like GPS, but all AdaControl options, like the name of the rules file or the output format, can easily be changed by editing the tool description file `AdaControl.tdf`.

AdaGide now features several AdaControl commands from the "tool" menu:

- "AdaControl" runs AdaControl on the currently edited file, with rules taken from the file named `verif.aru`.
- "AdaControl recursive" works like the previous command, with the addition of the "-r" (recursive) option. When used on the main program, it will analyze the whole set of compilation units in the program.
- "AdaControl interactive" runs AdaControl on the currently edited file, with a rule asked interactively from a pop-up.
- "AdaControl: delete .adt" removes existing tree files from the current directory. This is convenient when AdaControl complains that the tree files are not up-to-date.

3.8 Helpful utilities

This section describe utilities that are handy to use in conjunction with AdaControl.

3.8.1 pfni

The convention used to refer to entities (as described in Appendix A [Specifying an Ada entity name], page 145) is very powerful, but it may be difficult to spell out correctly the name of some entities, especially when using the overloaded syntax.

`pfni` (which stands for *Print Full Name Image*) can be used to get the correct spelling for any Ada entity. The syntax of `pfni` is:

```
pfni [-sofdq] [-p <project-file>] <unit>[:<span>]
     [-- <ASIS options>]
<span> ::=   <line_number>
           | [<first_line>]-[<last_line>]
           | <line_number>:<column_number>
```

or

```
pfni -h
```

If called with the "-h" option, `pfni` prints a help message and exits.

Otherwise, `pfni` prints the full name image of all identifiers declared in the indicated unit, unless there is a "-f" (full) option, in which case it prints the full name image of all identifiers (i.e. including those that are used, but not declared, in the unit). The image is printed without overloading information, unless the "-o" option is given.

In addition, `pfni` prints the initial value of variables if there is one, the range of discrete types, and the range of the indices of array types.

The <unit> is given either as an Ada unit, or as a file name, provided the extension is ".ads" or ".adb" (as in AdaControl). If a span is given, only identifiers within the span are printed. In the first form, the span includes only the indicated line; in the second form, the span includes all lines from <first_line> to <last_line> (if omitted, they are taken as the first and last line of the file, respectively). In the third form, the span includes only the place at the specified <line_number> and <column_number>.

Normally, the source line corresponding to the names is printed above the names. The "-q" (quiet) option suppresses this.

If the "-s" option is given (or the unit is a file name with a ".ads" extension), the specification of the unit is processed, otherwise the body is processed. The "-p" option specifies the name of an Emacs project file, and the "-d" option is the debug mode, as for AdaControl itself. ASIS options can be passed, like for AdaControl, after a "--" (but -FS is the default). See Section 3.1.11 [ASIS options], page 14.

As a side usage of `pfni`, if you are calling a subprogram that has several overloadings and you are not sure which one is called, use `pfni` with the "-o" option on that line: the program will tell you the full name and profile of the called subprogram.

3.8.2 makepat.sed

This file (provided in the "src" directory) is a sed script that transforms a text file into a set of correponding regular expressions. It is useful to generate model header files. See Section 5.21 [Header_Comments], page 74.

3.8.3 unrepr.sed

This file (provided in the "src" directory) is a sed script that comments out all representation clauses. It is typically useful if you use a different compiler that accepts representation clauses not supported by GNAT.

Typically, you would copy all your sources in a different directory, copy "unrepr.sed" in that directory, then run:

```
sed -i -f unrepr.sed *.ads *.adb
```

You can now run AdaControl on the patched files. Of course, you won't be able to check rules related to representation clauses any more...

Note that the script adds "--UNREPR " to all representation clauses. Its effect can thus easily be undone with the following commad:

```
sed -i -e "s/--UNREPR //" *.ads *.adb
```

3.9 Optimizing Adacontrol

There are many factors that may influence dramatically the speed of AdaControl when processing many units. For example, on our canonical test (same controls, same units), the

extreme points for execution time were 111s. vs 13s.! Unfortunately, this seems to depend on a number of parameters that are beyond AdaControl's control, like the relative speed of the CPU to the speed of the hard-disk, or the caching strategy of the file system.

This section will give some hints that may help you increase the speed of AdaControl, but it will not change the output of the program; you don't really need to read it if you just use AdaControl occasionnally. This section is concerned only with the GNAT implementation of ASIS; other implementations work differently.

Bear in mind that the best strategy depends heavily on how your program is organized, and on the particular OS and hardware you are using. Therefore, no general rule can be given, you'll have to experiment yourself. Hint: if you specify the "-v" option to AdaControl, it will print in the end the elapsed time for running the tests; this is very helpful to make timing comparisons.

Note: all options described in this section are ASIS options, i.e. they must appear last on the command line, after a "--".

3.9.1 Tree files and the ASIS context

Since AdaControl is an ASIS application, it is useful to explain here how ASIS works. ASIS (and therefore AdaControl) works on a set of units constituting a "context". Any reference to an Ada entity which is not in the context (nor automatically added, see below) will be ignored; especially, if you specify to AdaControl the name of a unit which is not included in the current context, the unit will simply not be processed.

ASIS works by exploring tree files (same name as the corresponding Ada unit, with a ".adt" extension), which are "predigested" views of the corresponding Ada units. By default, the tree files are generated automatically when needed, and kept after each run, so that subsequent runs do not have to recreate them.

A context in ASIS-for-Gnat is a set of tree files. Which trees are part of the context is defined by the "-C" option:

- -C1 Only one tree makes up the context. The name of the tree file must follow the option.
- -CN Several explicit trees make up the context. The name of the tree files must follow the option.
- -CA All available trees make up the context. These are the tree files found in the current directory, and in any directory given with a "-T" option (which works like the "-I" option, but for tree files instead of source files).

The "-F" option specifies what to do if the program tries to access an Ada unit which is not part of the context:

- -FT Only consider tree files, do not attempt to compile units on-the-fly
- -FS Always compile units on-the-fly, ignore existing tree files
- -FM Compile on-the-fly units for which there is no already existing tree file

Note that "-FT" is the only allowed mode, and *must* be specified, with the "-C1" and "-CN" options.

The default combination used by AdaControl is "-CA -FM".

3.9.2 Generating tree files manually

It is also possible to generate the tree files manually before running AdaControl. Although this mode of operation is less practical, it is recommended by AdaCore for any ASIS tool that deals with many compilation units. Some reasons why you might want to generate the tree files manually are:

- Your project uses GNAT project files, but you don't want to run AdaControl from GPS;
- Your project has several source directories (ASIS had problems with ADA_INCLUDE_PATH, until releases dated later than Sept. 1st, 2006). Note that an alternative solution is to specify source directories with the -I option;
- It is faster to generate tree files once than to use "compile on the fly" mode.

To generate tree files manually, simply recompile your project with the "-gnatct" option. This option can be passed to `gnatmake` normally. Of course, you will need all other options needed by your project (like the "-P" option if you are using GNAT project files).

Tree files may be copied into a different directory if you don't want your current directory to be cluttered by them. In this case, use the "-T" ASIS option to indicate the directory where the tree files are located.

If you chose to generate the tree files manually, you may want to specify the "-FT" ASIS option (see above) to prevent from accidental automatic recompilation.

3.9.3 Choosing an appropriate combination of options

In order to optimize the use of AdaControl, it is important to remember that reading tree files is a time-consuming operation. On the other hand, a single tree file contains not only information for the corresponding unit, but also for the *specifications* of all units that the given unit depends on. Moreover, our measures showed that reading an existing tree file may be *slower* than compiling the corresponding unit on-the-fly (but once again, YMMV).

Here are some hints to help you find the most efficient combination of options.

- If you want to run AdaControl on all units of your program, use the "-D" option to create a file containing the list of all required units, then use this file as an indirect file. Using the the "-r" option (recursive mode) of AdaControl implies an extra pass over the whole program tree to determine the necessary units.
- If you have not disabled any rule (and have many messages), specifying the "-i" option (ignore disabling) saves AdaControl the burden of checking whether rules are disabled, which can result in a sensible speed-up.
- Avoid having unnecessary tree files. All tree files in the context are read by ASIS, even if they are not later used. If you don't want to run AdaControl on the whole project, deleting tree files from a previous run can save a lot of time.
- When using an indirect file, the order in which units are given may influence the speed of the program. As a rule of thumb, units that are closely related should appear close to each other in the file. A good starting point is to sort the file in alphabetical order: this way, child units will appear immediately after their parent. You can then reorder units, and measure if it has a significant effect on speed.
- If you want to check a unit individually, try using the "-C1" option (especially if the current directory contains many tree files from previous runs). Remember that you

must specify the unit to check to AdaControl, and the tree file to ASIS. I.e., if you want to check the unit "Example", the command line should look like:

```
adactl -f rules_file.aru example -- -FT -C1 example.adt
```

provided the tree file already exists.

- For each strategy, first run AdaControl with the default options (which will create all necessary tree files). Compare execution time with the one you get with "-FT" and "-FS". This will tell you if compiling on-the-fly is more efficient than loading tree files, or not.

3.10 In case of trouble

3.10.1 Known issues

If you are using an old version of GNAT and your project includes source files located in several directories, the ADA_INCLUDE_PATH environment variable may not be considered by ASIS, resulting in error messages that tell you that the bodies of some units have not been found (and hence have not been processed). This problem has been fixed in GNAT dated later than Sept. 1st, 2006. If this happens, either provide your source directories as "-I" options (see Section 3.1.11 [ASIS options], page 14), or generate the tree files manually (see Section 3.9.2 [Generating tree files manually], page 25). Note that this problem does not happen if you are using Emacs project files (see Section 3.1.6 [Project files], page 13), nor if you are running AdaControl from GPS.

3.10.2 AdaControl or ASIS failure

Like any sophisticated piece of software, AdaControl may fail when encountering some special case of construct. ASIS may also fail occasionnally; actually, we discovered several ASIS bugs during the development of AdaControl. These were reported to ACT, and have been corrected in the wavefront version of GNAT - but you may be using an earlier version. In this case, try to upgrade to a newer version of ASIS. If an AdaControl or ASIS problem is not yet solved, AdaControl is designed in such a way that an occasionnal bug won't prevent you from using it.

If AdaControl detects an unexpected exception during the processing of a unit (an ASIS error or an internal error), it will abandon the unit, clean up everything, and go on processing the remaining units. This way, an error due to a special case in a unit will *not* affect the processing of other units. AdaControl will return a Status of 10 in this case.

However, if it is run with the "-x" option (eXit on error), it will stop immediately, and no further processing will happen.

If you don't want the garbage from a failing rule to pollute your report, you may chose to disable the rule for the unit that has a problem. See Section 4.3.8 [Inhibit command], page 35.

If you encounter a problem while using AdaControl, you are very welcome to report it through our Mantis bug tracking system (under Windows, you can click on "Report problem" in the AdaControl Start menu). Please include the exact control and the unit that caused the problem, as well as the captured output of the program (with "-dx" option).

4 Command language reference

AdaControl is about *controlling rules*. *Rules* are built in AdaControl; each rule has a name, and may require parameters. For the complete description of each rule, see Chapter 5 [Rules reference], page 37.

To run AdaControl, you need to define which rules you want to apply to your Ada units, what are the parameters, etc. In addition, you may want to define various things, like the file where the results should go, the output format, etc.

AdaControl defines a small command language which is used to describe how you want to process your units. Commands can be specified either on the command line or in a file, that we call here a rules file. Commands can also be given interactively; See Section 3.4 [Interactive mode], page 15.

4.1 General

The command language is not case-sensitive, i.e. the case of the keywords, rule names, and parameters is not significant. The layout of commands is free (i.e. a command can extend over several lines, and spaces are freely allowed between syntactic elements).

Comments are allowed in and between commands. Comments begin with a "#" or a "--", and extend to the end of the line.

Since wide characters are allowed in Ada programs, AdaControl accepts wide characters in commands as well. With GNAT, the encoding scheme is Hex ESC encoding (see the GNAT User-Guide/Reference-Manual). This is the prefered method, since few people require wide characters in programs anyway, and that keeping the default bracket encoding would not conveniently allow brackets for regular expressions, like those used by some rules. See Appendix B [Syntax of regular expressions], page 149.

If a syntax error is encountered in a command, an appropriate error message is output, and analysis of the rules file continues in order to output all errors, but no analysis of user code will be performed.

4.2 Controls

A *control command* is a command that declares one (or several) controls. A control defines how a rule is applied to Ada units. The syntax of a control command is as follows:

```
<control_command> ::= [<label> ":"] <control> {"," <control>} ";"
<control>   ::= <ctrl_kind> <Rule_Name> [<parameters>]
<parameters ::= "(" [<modifiers>] <value> {"," [<modifiers>] <value>} ")"
<ctrl_kind> ::= "check"|"search"|"count"
```

If present, the label gives a name to the control(s); it will be printed whenever each control is activated, and can be used to disable the control(s). See Section 4.2.4 [Disabling controls], page 30. If no label is present, the rule name is printed instead. The label must have the syntax of an Ada identifier, or else the label must be included within double quotes ("), in which case it can contain any character.

Each control consists of a <ctrl_kind> followed by a rule name, and (optionally) parameters. Some parameters may be preceded by modifiers (such as "not" or "case_sensitive"). The meaning of the rule parameters and modifiers depends on the rule.

Here are some examples of commands:

```
check unnecessary_use_clause;
All_Imports: search pragmas (Import);
"Why do you need that?": check entities (Unchecked_Conversion,
                                         all 'Address);
```

Specifying several controls with the same label is a shorthand which is equivalent to specifying the same label for several controls. It is handy when the label is long, and/or to stress that several controls are part of the same programming rule. For example:

```
"Check why this obsolete stuff is still used":
   check entities (obsolete_unit_1),     -- Note comma here!
   check instantiations (some_obsolete_generic);
```

4.2.1 Control kinds and report messages

There are three control kinds: "check", "search", and "count".

"Check" is intended to search for rules that must be obeyed in your programs. Normally, if a "Check" control fails, you should fix the program. "Search" is intended to report some situations, but you should consider what to do on a case-by-case basis. Roughly, use "check" when you consider that the failure of the control is an error, and "search" when you consider it as a warning. AdaControl will exit with a status of 1 if any "Check" control is triggered, and a status of 0 if only "Search" controls were triggered (or no control was triggered at all).

"Count" works like "Search", but instead of printing a message for each control which is triggered, it simply counts occurrences and prints a summary at the end of the run. There is a separate count for each control label (or if no label is given, the rule name is taken instead); if you give the same label to different controls, this allows you to accumulate the counts.

A report message (except for the final report of "count") comprises the following elements:

- the file name (where the control matches)
- the line number (where the control matches)
- the column number (where the control matches)
- the label (if there is one) and/or the rule name (the rule that matches).
- a message (why the control matches). A control whose kind is "check" will produce an error report message (i.e. containing the keyword "Error") and a control whose kind is "search" will produce a found report message (i.e. containing the keyword "Found").

The formatting of the report message depends on the format option, which can be selected with the "-F" command-line option or the "set format" command.

If the format is "Gnat" (the default) or "Gnat_Short", items are separated by ':'; this is the same format as the one used by GNAT error messages. Editors (like Emacs or GPS) that recognize this format allow you to go directly to the place of the message by clicking on it. In order to avoid too long messages, only the label appears, unless there is none, in which case it is replaced with the rule name.

If the format is "CSV" or "CSV_Short", items are separated by ',' and surrounded by double quotes. This is the "Comma Separated Values" format, which can be read by any

known spreadsheet program, except Excel(tm) by default, which uses the semicolon and not the comma to separate fields. Therefore, the formats "CSVX" and "CSVX_Short" do the same thing, but using semi-colons (';') instead of commas. Both the label (replaced by an empty column if there is none) and the rule name appear. Note that when an output file is created in one of the "CSV" formats, a title line is issued as the first line, following normal CSV convention.

If the format is "Source" or "Source_Short", the offending source line is output, and the message is output behind it, with a "!" pointing to the exact location of the problem.

If the format is "None", no error message is output at all. This is useful when only the return code of running AdaControl is desired (just to check if a program is OK or not). Note that this does *not* prevent the output of statistics, since these are under control of the "-S" option or the "set statistics" command. In this case, statistics are output in CSVX format, since asking for statistics with a "none" format is mainly useful for analysing the statistics with a spreadsheet program.

With recent versions of GNAT, the file name includes the full path of the source file. If the "_Short" form of the format option is used, the file name is stripped from any path. This can make it easier to compare the results of controlling units from various directories. Note that with older versions of GNAT, the file name never includes the full path, and the "_Short" form of the format option has no effect.

After each run (see Section 4.3.1 [Go command], page 32), statistics may be output, depending on the statistics level which is set with the "-S" option or the "set statistics" command. The meaning of the various levels is as follows:

- 0: No statistics are output (default)
- 1: A count of error and warning messages is output
- 2: The rule name and label (if any) of any control *not* triggered are output
- 3: The rule name and label (if any) of every control is output, together with a count of each triggering kind ("check", "search", "count"), or "not triggered" if the control was not triggered.

4.2.2 Parameters

Most rules accept parameters. Parameters can be:

- a keyword for the rule
- a numerical value
- a character string (often a regular expression)
- an Ada entity name

A numerical value is given with the syntax of an Ada integer or real literal (underscores and exponents are allowed as in Ada). Based literals are supported for integer values; if somebody can justify a need for supporting them for reals, we'll be happy to add this feature later...

A character string is given within double quotes "". As usual, quotes appearing within the string are doubled. The tilde character ("~") can be used as a replacement delimiter, but the same character must be used at both ends of the string. The latter has been chosen as a character not used by the various shells, and can be useful to pass quoted strings from

parameters on the command line (unfortunately, we could not use the percent ("%") sign, because it plays a special role in DOS/Windows).

An Ada entity name is the full name (prefixed with the names of all units that include it) of something declared in a program. It can be followed by overloading information, in order to uniquely identify the Ada entity. If an Ada entity is overloaded and no overloading information is provided, the rule is applied to all (overloaded) Ada entities that match the name. Alternatively, it can be "all" followed by a simple name, in wich case it applies to all entities with that name. See Appendix A [Specifying an Ada entity name], page 145 for the full description of the syntax. Here are some examples of entity names:

```
Ada.Text_IO.Put                        -- All Put defined in Ada.Text_IO
Ada.Text_IO.Put{Standard.Character}    -- The Put on Character
all Put                                -- All Put
Standard.Integer'Image                 -- The 'Image function on Integer
all 'Image                             -- All 'Image functions
```

4.2.3 Multiple controls

Most rules can be used in more than one control (with different parameters). There is no difference between a single or a multiple configuration rule use: outputs, efficiency, etc. are the same.

The following rules files produce an identical configuration:

```
Search Pragmas (Pure, Elaborate_All);
```

and

```
Search Pragmas (Pure);
Search Pragmas (Elaborate_All);
```

However, the second form can be used to give different labels. Consider:

```
Search Pragmas (Pure);
No_Elaborate: Search Pragmas (Elaborate_All);
```

The messages for pragma `Pure` will contain "PRAGMAS", while those for `Elaborate_All` will contain "No_Elaborate". If a disabling comment mentions `pragmas`, it will disable both controls, but a disabling comment that mentions `No_Elaborate` will disable only the second one.

4.2.4 Disabling controls

It is possible to disable controls on parts of the source code by placing markers in the source code. A marker is an Ada comment, where the comment mark (`--`) is immediately followed by the special tag "`##`" (by default).

There are two kinds of markers: block markers and line markers. Both kinds specify a list of controls to disable/re-enable. A list of controls is a list of rule names (to disable/re-enable all controls on the indicated rule(s)) or control labels (to disable/re-enable all controls with that label), separated by spaces. Alternatively, the list of controls can be the word "all" to disable/re-enable all controls.

In a "`--##`" line, everything appearing after another "##" tag (by default) is ignored. This allows the insertion of a comment explaining why the control is disabled at that point.

Both tags can be changed with the "set" command. See Section 4.3.6 [Set command], page 33.

4.2.4.1 Block disabling

A control is disabled from a "rule off" marker that applies to it until a "rule on" marker that applies to it. If there is no appropriate "rule on" marker, the control is disabled up to the end of file.

Syntax:

```
--## rule off <control_list>
Ada code block
--## rule on <control_list>
```

Ex:

```
--## rule off rule1 rule2 ## Authorized by QA ref 1234
I := I + 1;
Proc (I);
--## rule on rule2
```

4.2.4.2 Line disabling

A control is disabled only for the line where a marker that applies to it appears.

Syntax:

```
Ada code line --## rule line off <rule_list>
```

Ex:

```
I := I + 1; --## rule line off rule3 rule_label_1
```

Conversely, it is possible to re-enable a control for just the current line in a block where it is disabled:

Syntax:

```
Ada code line --## rule line on <rule_list>
```

Ex:

```
--## rule off rule1 rule2
...
I := I + 1; --## rule line on rule2
```

4.2.5 Limitation

Since the disabling is based on special comments, there is a conflict with the rule "header_comments" which is based on the content of comments. Line disabling is not possible with this rule, and block disabling needs special care. See Section 5.21 [Header_Comments], page 74.

4.3 Other commands

In addition to controls, AdaControl recognizes a number of commands. Although these commands are especially useful when using the interactive mode (see Section 3.4 [Interactive mode], page 15), they can be used in command files as well.

4.3.1 Go command

This command starts processing of the controls that have been specified.

Syntax:

```
go;
```

Controls are *not* reset after a "go" command; for example, the following program:

```
search entities (pack1);
go;
search entities (pack2);
go;
```

will first output all usages of `Pack1`, then all usages of both `Pack1` and `Pack2`. See Section 4.3.5 [Clear command], page 33 to reset controls.

If not in interactive mode, a "go" command is automatically added at the end, therefore it is not required in rules files.

4.3.2 Quit command

This command terminates AdaControl.

Syntax:

```
quit;
```

If given in a file, all subsequent commands will be ignored. This command is really useful only in interactive mode. See Section 3.4 [Interactive mode], page 15.

4.3.3 Message command

This command prints a message on the output file.

Syntax:

```
message "<any string>" [pause];
```

The length of the message is limited to 250 characters. If the word "pause" (case irrelevant) is specified after the message, AdaControl will wait for the user to press the Return key before proceeding.

Note that the message is syntactically a string, and must therefore be quoted (double quotes).

4.3.4 Help command

This command prints various informations about the rules and AdaControl itself.

Syntax:

```
Help [<help_item> {,<help_item>}]
<Help_Item> ::=<keyword> | <rule name> | variables ["<pattern>"]
<keyword>   ::= all | commands | license | list | options | rules | version
```

Without any argument, this command prints a summary of all commands and rule names. If given one or more keywords or rule names, it prints the corresponding help message. See Section 3.5.1 [Getting help], page 15 for the details.

4.3.5 Clear command

This command command clears (i.e. removes) controls that have been previously given.

Syntax:

```
Clear all | <rule name>{,<rule name>} ;
```

The command clears all controls given for the indicated rules, of for all rules if the `all` keyword is given. Rule variables (see Section 4.3.6 [Set command], page 33) associated to cleared rules are returned to their default values. For example, the following program:

```
search entities (pack1);
go;
clear all;
search entities (pack2);
go;
```

will first output all usages of `Pack1`, then all usages of `Pack2`. Without the "clear all" command, the second "go" would output all usages of `Pack1` together with all usages of `Pack2`.

4.3.6 Set command

This command sets various parameters of AdaControl.

Syntax:

```
set Format Gnat|Gnat_Short|CSV|CSV_Short|Source|Source_short|None;
set Check_Key|Search_Key "<value>"
set Max_Errors [<value>];
set Max_Messages [<value>];
set Output|New_Output <output file>;
set Statistics <level>;
set Tag1|Tag2 "<value>";
set Trace <trace file>;
set Debug|Exit_On_Error|Verbose|Warning|Warning_As_Error
      On|Off;
set Timing On|Off|Global
set Ignore On|Off|Inverted;
set <Rule_Name>.<Variable> <Value>
```

The "set format" command selects the output format for the messages, like the "-F" option; see Section 4.2.1 [Control kinds and report messages], page 28 for details.

The "set check_key" command defines a string which is used in place of "Error" in messages issued by a "check" control. Similarly, the "set search_key" command defines a string which is used in place of "Found" in messages issued by a "search" control. This can be useful when AdaControl is used, for example, to detect places where manual inspection is required; having the word "Error" in the message could be misleading to the persons in charge of the review. Note however that if you set these keys, the GPS interface will not be able to recognize properly the messages.

The "set max_errors" and "set max_messages" limit the output of AdaControl, like the "-m" and "-M" options; see Section 3.1.5 [Output limits], page 13 for details. If no <value> is given after the command name, the corresponding limitation is removed.

The "set output" and "set new_output" commands redirect the output of subsequent controls to the indicated file. If the string `console` (case irrelevant) is given as the <output file>, output is redirected to the console.

The "set new_output" always create a new file (or overwrites an existing file with the same name).

The "set output" command appends if the file exists, unless the "-w" option is given, in which case it is overwritten. However, the file is overwritten only the first time it is mentionned in an "output" command. This means that you can switch forth and back between two output files, all results from the same run will be kept. Note however that for this to work, you need to specify the output file exactly the same way: if you specify it once as "result.txt", and then as "./result.txt", the second one will overwrite the first one.

The "set statistics" command sets the statistics level, like the "-S" option; see Section 4.2.1 [Control kinds and report messages], page 28 for details.

The "set Tag1|Tag2" command changes the tags used to disable (or enable) rules. "Tag1" is the string that appears immediately after the comment indicator (`--`), and "tag2" is the tag that terminates the special comment. Note that these tags must be given as strings (in quotes) and that case is relevant. See Section 4.2.4 [Disabling controls], page 30 for details.

The "set trace" command redirects the trace messages of the "-d" option to the indicated file. If the string `console` (case irrelevant) is given as the <trace file>, trace messages are redirected to the console. As with the "-t" option, if the file exists, output is appended to it.

The "set Debug|Exit_On_Error|Verbose|Warning|Warning_As_Error" command activates ("on") or deactivates ("off") options. "Debug" corresponds to the "-d" option, "Exit_On_Error" to the "-x" option, "Ignore" to the "-i" option, "Timing" to the "-T" option, "Verbose" to the "-v" option, "Warning" to the "-E" option, and "Warning_As_Error" to the "-e" option. See Section 3.1.8 [Verbose and debug mode], page 13, Section 3.1.10 [Exit on error], page 14, Section 3.1.9 [Treatment of warnings], page 14, Section 3.1.4 [Output format], page 12, and Section 3.1.7 [Local disabling control], page 13 for details.

The "set Timing" command activates ("on") or deactivates ("off") the printing of the time spent in each rule after each "go" command. If set to "global" instead of "on", the timings are accumulated over all "go" commands, and output when the program terminates.

The "set Ignore" command governs handling of disabled messages (see Section 4.2.4 [Disabling controls], page 30). In default mode ("set Ignore Off"), disabled messages are not printed. When set to "on" ("set Ignore On"), all messages are printed, including those that are disabled. Setting this option can result in considerable speed-up of the printing of messages. When set to "Inverted" ("set Ignore Inverted"), *only* disabled messages are printed. This is useful to check which messages have been disabled.

Some rules may also have user-settable global variables that affect their behaviour; the last form of the "set" command allows changing their value. The variable name is of the form of a qualified name (i.e. "rule.var"), and the value depends on the variable. The description of the variables (if any) and appropriate values is given for each rule.

4.3.7 Source command

This command inputs commands from another file.

Syntax:

```
Source <input file>;
```

Commands are read and executed from the indicated file, then control is returned to the place after the "source" command. There is no restriction on the content of the sourced file; especially, it may itself include other "source" commands.

If <input file> is a relative file path, it is taken relatively to the file where the "source" command is given. Especially, if no path is specified, the sourced file will be taken from the same directory as the sourcing file (irrespectively of where the command is being run from). If the file is not found there, it is searched on the path given by the environment variable `ADACTL_PATH`.

The default extension is `.aru`, i.e. if <input file> is not found as given, AdaControl will retry the same name with `.aru` appended. It is a syntax error if the file is not found either.

If the string `console` (case irrelevant) is given as the <input file>, commands are read from the console until a "quit" command is given. This command is of course useful only from files, and allows to pass temporarily control to the user in interactive mode.

4.3.8 Inhibit command

This command prevents execution of certain controls on particular units.

Syntax:

```
Inhibit <rule name>|all ([all] <unit> {,[all] <unit>});
```

Controls refering to the given rule (or all rules if "all" is specified in place of a rule name) for the indicated unit(s) are not performed. In addition, if "all" is specified in front of the unit name, the unit will not be accessed at all, even from rules that follow call graphs, and could thus access this unit while analyzing other units.

There are several reasons why you might want to inhibit a control of a rule for certain units:

- The unit is known not to obey the rule in many places, and you don't want the output to be cluttered with too many messages (of course, you'll fix the unit in the near future!);
- The unit is known to obey the rule, execution of the rule is time-consuming, and you want to save some processing time;
- The unit is known to raise an ASIS bug, and until you upgrade to the appropriate version of GNAT, you don't want to be bothered by the error messages.

The "all" option for a unit is intended for the last case, to prevent ASIS bugs from spoiling any unit that calls something from an offending unit.

4.4 Example of commands

Below is an example of a file with multiple commands:

```
message "Searching Unchecked_Conversion";
search entitities (ada.unchecked_conversion);
set output uc_usage.txt;
go;
clear all;
```

```
message "Searching 'Address";
search entities (all 'Address);
set output address_usage.txt;
go;
```

This file will output all usages of `Ada.Unchecked_Conversion` into the file `uc_usage.txt`, then output all usages of the `'Address` attribute into the file `address_usage.txt`. Messages are output to tell the user about what's happenning.

5 Rules reference

This chapter describes each rule currently provided by AdaControl. Note that the `rules` directory of the distribution contains a file named `verif.aru` that contains an example of a set of rules appropriate to check on almost any software.

A general limitation applies to all rules. AdaControl is a *static* checking tool, and therefore cannot check usages that depend on run-time values. For example, it is not possible to check rules applying to an entity when this entity is aliased and accessed through an access value, or rules applying to subprogram calls when the call is a dispatching call.

5.1 Abnormal_Function_Return

This rule controls functions that may not terminate normally, i.e. where `Program_Error` could be raised due to reaching the end of the function without encountering a **return** statement.

5.1.1 Syntax

```
<control_kind> abnormal_function_return;
```

5.1.2 Action

The rule controls that the sequence of statements of each function body, as well as each of its exception handlers, ends with:

- a **return** statement (including extended return statements)
- a **raise** statement (or equivalently, a call to `Ada.Exceptions.Raise_Exception` or `Ada.Exceptions.Reraise_Occurrence`);
- a call to a procedure which is the target of a **pragma** `No_Return`;
- a block statement, whose last statement of its sequence and any exception handler is one of these;
- an **if** statement that includes an **else** path, and where the last statement of every path is one of these;
- a **case** statement where the last statement of every path is one of these.

This is a sufficient (but of course not necessary) condition to ensure that no function raises `Program_Error` due to reaching the end of its statements without encountering a **return**.

This rule can be specified only once.

Ex:

```
check abnormal_function_return;
```

5.1.3 Tips

This rule checks that a function always returns correctly, but does not prevent multiple **return** statements in functions. If you want to ensure that there is exactly one **return** statement in functions, and that this statement is always the last one, use this rule together with the rule `statements(function_return)`. See Section 5.53 [Statements], page 117.

It is possible to exit from an extended return statement with an **exit** or **goto** statement. If this happens, the return statement is not considered a proper return statement, and an appropriate message is issued.

5.2 Allocators

This rule controls the use of allocators (i.e. dynamic memory allocation).

5.2.1 Syntax

```
<control_kind> allocators [(<target> {, <target>})];
<target>   ::= [anonymous | inconsistent | not] [<category>|<entity>]
<category> ::= ()  | access    | array | delta  | digits |
               mod | protected | range | record | tagged | task
```

5.2.2 Action

If one or several <entity> or <category> are given, only allocators whose allocated type matches the <entity>, or whose type belongs to the indicated <category>, are controlled; otherwise all allocators are controlled. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. The meaning of <category> is:

- "()": The allocated value is of an enumerated type.
- "access": The allocated value is of an access type.
- "array": The allocated value is of an array type.
- "delta": The allocated value is of a fixed point type (it is not currently possible to distinguish ordinary fixed point types from decimal fixed point types).
- "digits": The allocated value is of a floating point type.
- "mod": The allocated value is of a modular type.
- "protected": The allocated value is of a protected type.
- "range": The allocated value is of a signed integer type.
- "record": The allocated value is of an (untagged) record type.
- "tagged": The allocated value is of a tagged type (including type extensions).
- "task": The allocated value is of a task type.

This rule is especially useful for finding memory leaks, since it tells all the places where dynamic allocation occurs.

If a parameter is preceded by the word "not", allocators for the corresponding type or category are not controlled (i.e. they are always allowed). If a control includes only "not" parameters, an implicit check for all allocators is assumed.

If a parameter is preceded by the word "anonymous", only allocators whose expected type is an anonymous access type are controlled.

If a parameter is preceded by the word "inconsistent", only allocators whose allocator subtype (the name after "**new**") is not the same as the designated subtype (from the access type declaration) are controlled. However an allocator is not considered inconsistent when the designated subtype imposes no special constraint:

- when it is a class-wide type, since the allocator subtype will generally be of some descendant specific type;
- when it is an unconstrained array type, since the allocated subtype is necessarily constrained;
- when it is a base type (of the form `T'Base`).

Note that if the access type includes a constraint like in the following example:

```
type Acc is access integer range 1..10;
```

all allocators will necessarilly be inconsistent, since there is no way to repeat the constraint at the place of the allocator.

"Inconsistent" can be given alone, in which case all inconsistent allocators are controlled.

Ex:

```
search allocators (standard.string);
check allocators (T'Class);
check allocators (array);
check allocators (Inconsistent standard.Integer);
check allocators (Inconsistent);

 -- all task allocators, except when the type is called "special":
check allocators (task, not all Special);
```

5.2.3 Tips

The type given as an <entity> in the rule must be a first named subtype, and the rule will also find allocators that use a subtype of this type. If the type is declared within a generic package, the rule will control all corresponding types from instantiations.

The type mentionned in the rule is the one following the **new** keyword, which is not necessarily the same as the expected type in presence of implicit conversions like this:

```
   type T is tagged ...;
   type Class_Access is access T'Class;
   X : Class_Access;
begin
   X := new T;
```

This allocator will be found for type `T`, not for type `T'Class`.

For <categories>, note that the rule "sees through" derived and private types (i.e. it will trigger if the ultimate type belongs to the indicated category).

The reason for the "inconsistent" modifier is that inconsistent allocators may cost a double check. Given:

```
   type Acc is access Positive;
   V : Acc;
begin
   V := new Natural'(...);
```

The compiler will first check the constraint for Natural, then the constraint for Positive. To avoid confusion, it is better to always use the same subtype for the allocator as used in the access type declaration.

The reason for the "anonymous" modifier is that allocators of an anonymous type (especially access parameters) create a terrible mess in accessibility rules, and are better avoided.

5.2.4 Limitations

In some (rare) cases involving anonymous access types as array or record components, ASIS provides no way to determine the target type of the (anonymous) acccess type. Inconsistent allocators will thus not be controlled. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.3 Array_Declarations

This rule controls properties of arrays, by enforcing a consistent value or range of values for the lower or upper bound, or by limiting the possible size. It can also control various aspects of the component type of the array.

5.3.1 Syntax

```
<control_kind> array_declarations (first,      <value> | <bounds>);
<control_kind> array_declarations (last,       <value> | <bounds>);
<control_kind> array_declarations (dimensions, <value> | <bounds>);
<control_kind> array_declarations (length,     <bounds>);
<control_kind> array_declarations (component,  <type> {,<repr_cond>});
<control_kind> array_declarations (index,      <type> | <> {,<type> | <>});
<bounds>    ::= min|max <value> [, min|max <value> ]
<type>      ::= <entity>|<category>
<category > ::= () | access    | array | delta  | digits | mod | private
                  | protected | range | record | tagged | task
<repr_cond> ::= [not] pack | size | component_size
```

5.3.2 Action

The first parameter is a subrule keyword:

- "First" and "Last" control the lower (respectively upper) bound of each dimension of arrays (even unconstrained array types). If a single value is specified without the "min" or "max" modifiers, the subrule controls the bounds that are not exactly this value; otherwise, it controls the bounds that are smaller than the given "min" value or greater than the given "max" value. It is possible, but not required to specify both "min" and "max". If this subrule is given both for "search" and for "check", the value(s) for "search" is interpreted as the prefered one, and the value(s) for "check" is interpreted as an alternative acceptable one; i.e., it is a warning if the value is the one given for "check", and an error if it is neither. In short:

  ```
  search array_declarations (first, 1);
  check array_declarations (first, min -1, max 1);
  ```

  will be silent if the lower bound of an array is 1, it will issue a warning if it is in the range -1 .. 1, and an error otherwise.

- "Dimensions"controls the number of dimensions of arrays. If a single value is specified without the "min" or "max" modifiers, the subrule controls arrays whose number of

dimensions is not exactly this value; otherwise, it controls arrays whose number of dimensions are smaller than the given "min" value or greater than the given "max" value. It is possible, but not required to specify both "min" and "max". If this subrule is given both for "search" and for "check", the value(s) for "search" is interpreted as the prefered one, and the value(s) for "check" is interpreted as an alternative acceptable one; i.e., it is a warning if the value is the one given for "check", and an error if it is neither. In short:

```
search array_declarations (Dimensions, 1);
check array_declarations (Dimensions, min 2, max 3);
```

will be silent for one-dimensional arrays, it will issue a warning for 2- and 3-dimensional arrays, and an error otherwise.

- "Length" controls arrays that have a dimension whose number of elements is smaller than the given "min" value or greater than the given "max" value (except for unconstrained array types). At least one of "min" or "max" must be specified, but it is not required to specify both.
- "Component" controls arrays whose component type is the indicated <entity>, or whose component type belongs to the indicated <category>. If the <entity> is a subtype, only arrays whose components are of that subtype are controlled. If the indicated <entity> is a type, all arrays whose components are of that type (including subtypes) are controlled. The meaning of <category> is:
  - "()": The component is of an enumerated type.
  - "access": The component is of an access type.
  - "array": The component is of an array type.
  - "delta": The component is of a fixed point type (it is not currently possible to distinguish ordinary fixed point types from decimal fixed point types).
  - "digits": The component is of a floating point type.
  - "mod": The component is of a modular type.
  - "private": The component is of a private type (including private extensions).
  - "protected": The component is of a protected type.
  - "range": The component is of a signed integer type.
  - "record": The component is of an (untagged) record type.
  - "tagged": The component is of a tagged type (including type extensions).
  - "task": The component is of a task type.

  If <repr_cond> are specified, the rule controls only arrays to which all the corresponding representation items apply:
  - "pack": A pragma Pack applies to the array.
  - "not pack": No pragma Pack applies to the array.
  - "size": A size representation clause applies to the array.
  - "not size": No size representation clause applies to the array.
  - "component_size": A component_size representation clause applies to the array.
  - "not component_size": No component_size representation clause applies to the array.

- "index" controls arrays whose index types are the indicated <entity>, or whose index types belong to the indicated <category>. If the <entity> is a subtype, only arrays whose indexes are of that subtype are controlled. If the indicated <entity> is a type, all arrays whose indexes are of that type (including subtypes) are controlled. The meaning of <category> is the same as for "component", but obviously only "()", "range", and "mod" are allowed.

  The number of <entity> given determines the dimensionality of the controlled arrays. If a "<>" is given in place of an entity, it means that any type matches at that position.

This rule can be specified several times for the "component" and "index" subrules. For other subrules, it can be specified at most once for each subrule and for each of "check", "search" and "count". It is thus possible for each subrule to have a value considered a warning, and a value considered an error.

Ex:

```
-- All arrays should start at 1:
check array_declarations (first, 1);

-- No arrray of more than 100 elements:
check array_declarations (length, max 100);

-- No empty array:
check array_declarations (length, min 1);

-- Arrays whose component type is private:
check array_declarations (component, private);

-- Packed arrays of Character
check array_declarations (component, Standard.Character, pack);

-- Packed arrays of record without size clause
check array_declarations (component, record, packed, not size);

-- One-dimensional arrays indexed by Integer
check array_declarations (index, standard.integer);

-- Three dimensional arrays whose second index is an enumeration
check array_declarations (index, <>, (), <>);
```

5.3.3 Tips

The subrule `Max_Length` ignores index constraints that are not static. Non static index constraints can be controlled with the rule `Non_Static (Index_Constraint)`. See Section 5.36 [Non_Static], page 95.

Requiring the same *upper* bound for all arrays is not very useful, but:

```
check array_declarations (last, min 1);
```

can be used to check that no array has a negative or zero upper bound.

The subrule "index" controls a precise pattern of types used as indices. To control the use of a type as an index at any position and irrespectively of the number of indices of the array, use the rule "type_usage". See Section 5.57 [Type_Usage], page 125.

5.4 Aspects

This rule controls aspect specifications (new feature in Ada 2012), either all of them or specific ones.

5.4.1 Syntax

```
<control_kind> aspects [(all | <aspect mark> {, <aspect mark>})];
```

5.4.2 Action

Without parameters (or if "all" is given), controls all aspect specifications. Otherwise, controls only the aspect specifications corresponding to the given aspect marks.

Ex:

```
search aspects;
DBC: check aspects (Pre, Post, Pre'Class, Post'Class);
```

5.5 Assignments

This rule controls various issues related to the assignment statement: assignments that involve array sliding, redundant assignments to the same variable, or groups of assignments that are replaceable by aggregate assignment.

5.5.1 Syntax

```
<control_kind> assignments (sliding);
<control_kind> assignments (repeated);
<control_kind> assignments (groupable, <filter> {,<filter>});
<filter> ::= given <min_val> | missing <max_val> | ratio <min_val> |
             total <max_val>
```

5.5.2 Action

The first form (keyword "sliding") controls array assignments where the target variable has a different lower bound than the assigned expression; this is allowed by the language only in so-called "sliding" contexts.

Other subrules control properties of groups of assignment statements. A group is made of consecutive assignments, without any other intervening kind of statements (except null statements).

The second form (keyword "repeated") controls when a same variable (or a same subcomponent of a structured variable) is assigned several times in the same group of assignments. This form of the rule can be given only once.

The third form (keyword "groupable") controls assignments to different subcomponents of a same structured variable; such assignments are often replaceable by a global assignment of an aggregate to the variable. One or several <filter> parameters indicate under which conditions a group is reported:

- "given": <min_val> (an integer value) indicates the minimum number of assigned subcomponents that will trigger the rule (i.e. the rule is triggered if the number of assignments to subcomponents of a same variable is greater or equal to the indicated value).
- "missing": <max_val> (an integer value) indicates the maximum number of subcomponents not assigned that will trigger the rule (i.e. the rule is triggered if the number of subcomponents not assigned to is lesser or equal to the indicated value).
- "ratio": <min_val> (an integer value) indicates the minimum percentage of assigned subcomponents that will trigger the rule (i.e. the rule is triggered if the percentage of assigned subcomponents is greater or equal to the indicated value).
- "total": <max_val> (an integer value) indicates the maximum number of subcomponents of the type that will trigger the rule (i.e. the rule is triggered if the number of subcomponents of the record type is lesser or equal to the indicated value).

If several filters are given, the rule is triggered if all conditions are met ("and" logic). Note however that this rule can be given several times, thus achieving "or" logic.

The rule is *not* triggered on an object if a subcomponent of that object is of a limited type, since global assignment would not be allowed in that case.

For other structured objects, a subcomponent is counted as assigned if it has been assigned in full, or if it *should* have been assigned in full (in other words: if the rule is triggered on those subcomponents as well) - recursively, of course.

Ex:

```
search Assignments (sliding);
check  Assignments (repeated);

-- Warn if a at least 3 fields are given and at most
-- two fields are missing, or if 80% of the fields are given:
search assignments (groupable, given 3, missing 2);
search assignments (groupable, ratio 80);
```

5.5.3 Tip

The "sliding" subrule is not intended to prevent all cases of slidings (the dynamic ones are uncheckable), it is rather an indication of "obvious" cases that could be avoided.

Note that for the "groupable" subrule, it is possible to give 1 for the "given" criterion; in this case, any assignment to parts of a structured variable will be reported, only global assignment is allowed.

5.5.4 Limitations

As usual, AdaControl can control only static aspects of assignments. Therefore, it cannot control assignments whose target is not statically known (like dynamic indexing of arrays). Slices are always considered dynamic (the cases where it would be useful did not seem worth the additional complexity).

For the "sliding" subrule, if the assigned expression is a multidimensional aggregate, only the first dimension is checked for sliding, other dimensions are ignored. This is not considered an important issue, since in any case the rule can detect only static cases, and

the handling of sliding in multi-dimensional array aggregates is extremely touchy (see RM 4.3.3 for details).

For the "groupable" subrule, if the number of subcomponents is not statically determinable (dynamic arrays, discriminated records), only the "given" criterion can be met.

5.6 Barrier_Expressions

Although the language allows any expression as the barrier of a protected entry, it is generally better to use only "simple" expressions. This rule controls the kind of constructs allowed in barrier expressions.

5.6.1 Syntax

```
<control_kind> Barrier_Expressions ([<allowable> {, <allowable>}]);
<allowable>    ::= <entity> | <keyword>
<keyword> ::= allocation          | any_component   | any_variable        |
              arithmetic_operator | array_aggregate | comparison_operator |
              conversion          | dereference     | indexing            |
              function_attribute  | local_function  | logical_operator    |
              record_aggregate    | value_attribute
```

5.6.2 Action

Without parameters, the only elements allowed in barriers are references to boolean components of the protected element and litterals (this corresponds to what is allowed for the Ravenscar profile). Parameters specify other constructs that are allowed:

- Any <entity> (like a global variable, a function...) can be specified and is thus allowed. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.
- "allocation" allows use of allocators.
- "any_component" allows use of protected components that are not of type `Standard.Boolean`.
- "any_variable" allows use of any variable (i.e. variables external to the protected element).
- "arithmetic_operator" allows use of predefined arithmetic operators (`"+"`, `"**"`, etc.).
- "array_aggregate" allows use of array aggregates.
- "comparison_operator" allows use of predefined comparison and membership operators (`"="`, `">"`, **in**, etc.).
- "conversion" allows use of type conversions and type qualifications.
- "dereference" allows use of dereferencing of access types (both implicit and explicit dereferences).
- "indexing" allows use of array indexing and slices.
- "function_attribute" allows use of attributes that are functions (like `'Pred`, `'Image`, etc.).
- "local_function" allows use of (protected) functions declared in the same protected object.

- "logical_operator" allows use of predefined logical operators and short-circuit forms (**and**, **or else**, etc.).
- "record_aggregate" allows use of record aggregates and extension aggregates.
- "value_attribute" allows use of attributes that are simple values (like `'First`, `'Terminated`, etc.).

This rule can be given only once for each of "check", "search" and "count".

Ex:

```
search barrier_expressions;
check  barrier_expressions (logical_operator, comparison_operator,
                            any_component,
                            Pack.Global_State);
```

5.6.3 Tips

The goal of the "Simple_Barrier" restriction from the Ravenscar profile is to ensure that evaluation of barriers never raise exceptions. Even simple things like a qualified expression can raise exceptions, but in practice more than the restriction of the Ravenscar profile can be "reasonably" allowed.

Note that the various "operator" keywords allow only the use of predefined operators. If a user defined operator should be allowed, provide it explicitely as an <entity>. There is no way to allow any function call, since this would boil down to allowing pretty much anything, but you can of course specify explicitely functions that can be called.

You can provide this rule both for "check" and "search", but of course it makes sense only if the set of allowed features for "search" is a superset of those allowed for "check". This way, the use of certain features can be interpreted only as a warning.

5.7 Case_Statement

This rule controls various metrics related to the **case** statement. It is intended for cases where it is desired to limit the complexity of **case** statements.

5.7.1 Syntax

```
<control_kind> Case_Statement (<subrule>, <bound> [, <bound>]);
<subrule> ::= others_span | paths | range_span | values | values_if_others
<bound>   ::= min | max <value>
```

5.7.2 Action

The first parameter is a subrule keyword. The second (and optionnally third) parameter give the minimum and/or maximum allowed values (i.e. the rule will control values outside the indicated interval). If not specified, the minimum value is defaulted to 0 and the maximum value to infinity. The parameters controlled by each subrule are:

- "others_span" controls the number of values covered by **when others** case alternatives.
- "paths" controls the number of paths (i.e. **when** branches).
- "range_span" controls the number of values covered by ranges used as choices.
- "values" controls the number of values covered by the subtype of the **case** selector.

- "values_if_others" is like "values", but is activated only for **case** statements with a **when others** alternative.

This rule can be specified at most once for each subrule and for each of "check", "search" and "count". It is thus possible for each subrule to have a value considered a warning, and a value considered an error.

Ex:

```
check  Case_Statement (others_span, min 1);
search Case_Statement (others_span, min 5);

check  Case_Statement (values, max 10);
check  Case_Statement (paths, min 3, max 30);
```

5.7.3 Tips

To control that no range is used as a choice in a **case** statement:

```
check case_statement (range_span, max 0);
```

To control "**when others**" that cover no value at all:

```
check case_statement (others_span, min 1);
```

5.7.4 Limitations

If some characteristic of the **case** statement depend on a generic formal type, it is not possible to control some of the features statically. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.8 Characters

This rule makes sure that the program text does not use "undesirable" characters.

5.8.1 Syntax

```
<control_kind> characters [(<subrule> {, <subrule>})];
<subrule> ::= control | not_iso_646 | trailing_space | wide
```

5.8.2 Action

The rule controls the occurrence in the source file of characters belonging to the classe(s) defined by the subrules. Without parameters, all classes are controlled. The classes are defined as follows:

- "control": control characters that are allowed by the language (ASCII HT, ASCII VT and ASCII FF).
- "not_iso_646": characters outside the ISO-646 set (aka ASCII).
- "trailing_space": space characters appearing at the end of the source line.
- "wide": wide characters that are not in `Standard.Character`.

This rule can be given only once for each class of characters.

Ex:

```
check characters (control, trailing_space);
search characters (not_iso_646);
```

5.8.3 Limitations

With the "wide" subrule, the error message may seem to not always appear at the right place; this depends on the encoding scheme used. For example, if your source contains (using bracket encoding):

```
S : Wide_String := "["1041"]["1042"]";
```

it will appear to AdaControl as a string containing two characters, and therefore the error message for the second wide character will point at two characters after the opening quote of the string.

This rule controls only the characters in the source file; other means of having characters in the corresponding classes (like using the `'Val` attribute) are not controlled.

5.9 Comments

This rule controls comments that must, or must not, appear in certain cases.

5.9.1 Syntax

```
<control_kind> comments (pattern, "<pattern>" {, "<pattern>"});
<control_kind> comments (position, <value> | <bounds>);
<control_kind> comments (terminating {, "<pattern>" | begin | end});
<control_kind> comments (unnamed_begin, <kind> {, <kind>});
<bounds>    ::= min|max <value> [, min|max <value> ]
<kind>      ::= [<condition>] <unit_kind>
<condition> ::= always | declaration | program_unit
<unit_kind> ::= all | procedure | function | entry | package | task
```

5.9.2 Action

The first parameter is a subrule name which detemines what is being controlled.

- "pattern" controls comments that match one of the given patterns (given as strings). Only the "useful" part of the comment is matched against the patterns, i.e. the part after the "`--`" and spaces following it. Patterns are given using the full Regexp syntax. see Appendix B [Syntax of regular expressions], page 149 for details. Pattern matching is always case insensitive.

  This subrule is especially useful to find lines with comments like "TBSL" (To Be Supplied Later) or "fixme", which are often used to mark places where something should be done before releasing the program.

- "position" controls the starting position of comments. If a single value is specified without the "min" or "max" modifiers, the subrule controls comments that do not start exactly at the indicated column position; otherwise, it controls comments whose starting column is smaller than the given "min" value or greater than the given "max" value. It is possible, but not required to specify both "min" and "max". If this subrule is given both for "search" and for "check", the value(s) for "search" is interpreted as the prefered one, and the value(s) for "check" is interpreted as an alternative acceptable one; i.e., it is a warning if the value is the one given for "check", and an error if it is neither. In short:

  ```
  search comments (position, 1);
  ```

```
check  comments (first, min 1, max 6);
```

will be silent for comments that start in column 1, it will issue a warning for comments that start at columns 2 to 6, and an error otherwise.

- "terminating" controls comments that are at the end of an otherwise non empty line (i.e. that appear on the same line as a declaration or statement). If "begin" is specified, comments appearing on a line that contains only a **begin** are allowed (not reported); similarly, if "end" is specified, comments appearing on a line that contains only an **end** are allowed. Otherwise, the other parameters are patterns that specify forms of comments that are allowed. Patterns are given using the full Regexp syntax. see Appendix B [Syntax of regular expressions], page 149 for details. Pattern matching is always case insensitive.
- "unnamed_begin" controls **begin** of various constructs that do not have a comment that repeats the name of the program unit associated to the **begin**. Except for spaces, the comment must not contain anything else than the unit name.

  The <condition> keyword determines circumstances where the comment is required:

  - "always" (default): the comment is always required.
  - "declaration": the comment is required only if the preceding declaration part is non-empty (not counting pragmas).
  - "program_unit": the comment is required only if the preceding declaration part contains the declarations of other program units (subprograms, packages, protected objects, or tasks).

  The <unit_kind> keyword detemines the kind of program unit to which the rule applies ("all" stands for all kinds). The subrule can be given only once of each kind of program unit.

Ex:

```
check comments (pattern, "TBSL");

-- Report places where rules are disabled:
search comments (pattern, "##.* off");

-- End of line comments are not allowed, except for the
-- comment that repeats the name of a procedure on the "begin"
-- line, and special AdaControl comments
check comments (terminating, begin, "^ *##");

-- Named begin required for packages unless they have no
-- declaration, and subprograms if they have nested units
check comments (unnamed_begin, declaration package);
check comments (unnamed_begin, program_unit procedure);
check comments (unnamed_begin, program_unit function);
```

5.9.3 Tips

Remember that a Regexp matches if the pattern matches any part of the identifier. Use "^" and "$" to match the beginning (resp. end) of the comment, or both.

5.9.4 Limitations

This rule does not support wide characters outside the basic Latin-1 set.

5.10 Declarations

This rule controls usage of various kinds of declarations, possibly only those occurring at specified locations.

5.10.1 Syntax

```
<control_kind> declarations (<subrule> {, <subrule>});
<subrule>     ::= {<location_kw>} <declaration_kw>
<location_kw> ::= all | block   | library | local      | nested |
                  own | private | public  | in_generic | task_body
<declaration_kw> ::=
   any_declaration                   | abstract_function                |
   abstract_operator                 | abstract_procedure               |
   abstract_type                     | access_all_type                  |
   access_constant_type              | access_constrained_array_type    |
   access_def_discriminated_type     | access_formal_Type               |
   access_language_type              | access_nondef_discriminated_type |
   access_protected_type             | access_subprogram_type           |
   access_task_type                  | access_unconstrained_array_type  |
   access_unknown_discriminated_type | access_type                      |
   aliased_array_component           | aliased_constant                 |
   aliased_protected_component       | aliased_record_component         |
   aliased_variable                  | anonymous_access_component       |
   anonymous_access_constant         | anonymous_access_discriminant    |
   anonymous_access_parameter        | anonymous_access_variable        |
   anonymous_subtype_allocator       | anonymous_subtype_case           |
   anonymous_subtype_declaration     | anonymous_subtype_for            |
   anonymous_subtype_indexing        | array                            |
   array_type                        | binary_modular_type              |
   box_defaulted_formal_function     | box_defaulted_formal_procedure   |
   character_literal                 | child_unit                       |
   class_wide_constant               | class_wide_variable              |
   constant                          | constrained_array_constant       |
   constrained_array_type            | constrained_array_variable       |
   controlled_type                   | decimal_fixed_type               |
   defaulted_discriminant            | defaulted_generic_parameter      |
   defaulted_parameter               | deferred_constant                |
   derived_type                      | discriminant                     |
   empty_private_part                | empty_visible_part               |
   enumeration_type                  | entry                            |
   equality_operator                 | exception                        |
   expression_function               | extension                        |
   fixed_type                        | float_type                       |
```

formal_function	formal_package
formal_procedure	formal_type
function	function_call_renaming
function_instantiation	generic
generic_function	generic_package
generic_procedure	handlers
incomplete_type	in_out_generic_parameter
in_out_parameter	initialized_protected_component
initialized_record_component	initialized_variable
instantiation	integer_type
interface_type	library_unit_renaming
limited_private_type	modular_type
multiple_names	multiple_protected_entries
name_defaulted_formal_function	name_defaulted_formal_procedure
name_defaulted_formal_function	name_defaulted_formal_procedure
named_number	non_binary_modular_type
non_identical_operator_renaming	non_identical_renaming
non_joint_ce_ne_handler	non_limited_private_type
non_ravenscar_task	not_operator_renaming
null_defaulted_formal_procedure	null_extension
null_ordinary_record_type	null_procedure
null_procedure_body	null_procedure_declaration
null_tagged_type	operator
operator_renaming	ordinary_fixed_type
ordinary_fixed_type_no_small	ordinary_fixed_type_with_small
ordinary_record_type	ordinary_record_variable
out_parameter	package
package_instantiation	package_statements
predefined_operator	private_extension
procedure	procedure_instantiation
protected	protected_discriminant
protected_entry	protected_type
protected_variable	record_type
renaming	renaming_as_body
renaming_as_declaration	scalar_variable
self_calling_function	self_calling_procedure
separate	signed_type
single_array	single_protected
single_task	subtype
tagged_private_type	tagged_type
tagged_variable	task
task_discriminant	task_entry
task_type	task_variable
type	unconstrained_array_constant
unconstrained_array_type	unconstrained_array_variable
unconstrained_subtype	uninitialized_protected_component
uninitialized_record_component	uninitialized_variable

```
unknown_discriminant          | variable                    |
variant_part
```

5.10.2 Action

The <location_kw> restricts the places where the occurrence of the declaration is controlled. Several <location_kw> can be given, in which case the declaration is controlled at places where all the keywords apply. If there is no <location_kw>, it is assumed to be "all".

- `all`: puts no special restriction to the location. This keyword can be specified for readability purposes, and if specified must appear alone (not with other <location_kw>).
- `block`: only declarations appearing in block statements are controlled.
- `library`: only library level declarations are controlled.
- `local`: only local declarations are controlled (i.e. only declarations appearing in (generic) packages, possibly nested, are allowed).
- `nested`: only declarations nested in another declaration are controlled (i.e. only library level declarations are allowed).
- `own`: only declarations that are local to a (generic) package body are controlled.
- `public`: only declarations appearing in the visible part of (generic) packages are controlled.
- `private`: only declarations appearing directly in a private part are controlled.
- `in_generic`: only declarations appearing directly or indirectly in a generic specification or body are controlled.
- `task_body`: only declarations appearing directly in a task body are controlled. Note that it would not make sense to have a <location_kw> for task *specifications*, since only entries can appear there, and they cannot appear anywhere else.

The <declaration_kw> specifies what kind of declaration to control:

- Declaration keywords that are Ada keywords match the corresponding Ada declarations.
- `any_declaration` controls all declarations. This is of course not intended to forbid all declarations in a program (!), but *counting* all declarations can be quite useful.
- `abstract_function`, `abstract_operator`, and `abstract_procedure` control the declarations of abstract functions, abstract operators, and abstract procedures, respectively.
- `abstract_type` controls the declaration of non-formal abstract types.
- `access_type` controls all access type declarations, while `access_subprogram_type`, `access_protected_type`, and `access_task_type` control only access to procedures or functions, access to protected types, or access to task types, respectively. Similarly, `access_constrained_array_type` and `access_unconstrained_array_type` control access to constrained or unconstrained array types, `access_def_discriminated_type`, `access_nondef_discriminated_type`, and `access_unknown_discriminated_type` control access to types with discriminants with default values, without default values, and unknown discriminants, respectively. `access_formal_type` controls access to (generic) formal types, `access_all_type` control generalized access to variables types (aka "**access all** `T`", and `access_constant_type` control generalized access to

constants types (aka "**access constant** T"). `access_language_type` controls access to language defined private types.

- `aliased_variable` and `aliased_constant` control the declarations of aliased variables or constants, respectively.
- `aliased_array_component` controls the declaration of arrays (array types or single arrays) whose components are declared aliased.
- `aliased_record_component` and `aliased_protected_component` control the declarations of aliased record (respectively protected) components.
- `anonymous_access_component` controls array and record components that are of an anonymous access type (but not discriminants, which are controlled by `anonymous_access_discriminant`). Similarly, `anonymous_access_constant` and `anonymous_access_variable` control constants and variables that are of an anonymous access type (including generic formal **in** and **in out** parameters, respectively). `anonymous_access_parameter` controls subprogram parameters that are of an anonymous access type, the only ones that existed in Ada 95. Note that to avoid unnecessary messages, if a subprogram has an explicit specification, the message for `anonymous_access_parameter` is given on the specification and not repeated on the body.
- `anonymous_subtype_declaration` controls the declarations of anonymous subtypes and ranges that are part of some other declaration. Similarly, `anonymous_subtype_allocator`, `anonymous_subtype_case`, `anonymous_subtype_for`, and `anonymous_subtype_indexing` control anonymous subtype declarations and ranges that are part of allocators, **case** statements (ranges in the **when** path), **for** loop statements, and indexing of slices or array aggregates, respectively.
- `array` controls all array definitions (array types and single arrays), while `array_type` controls only array types and `single_array` controls only single arrays (objects of an anonymous array type). `constrained_array_type` controls only constrained array types, while `unconstrained_array_type` controls only unconstrained array types. `constrained_array_variable` controls variable declarations where the given (or anonymous) array type is constrained, while `unconstrained_array variable` controls variable declarations where the given (or anonymous) array type is unconstrained (and the constraint is provided by the initial value). `constrained_array_constant` and `unconstrained_array_constant` do the same with constants instead of variables.
- `character_literal` controls the declaration of new character literals, i.e. character literals defined as part of the values of an enumeration type.
- `child_unit` controls the declaration of all child units.
- `constant` controls all constants, while `class_wide_constant` control the declaration of constants of a class-wide type, and `deferred_constant` controls the declaration of deferred constants.
- `controlled_type` controls the declaration of controlled types, i.e. descendants of `Ada.Finalization.Controlled` or `Ada.Finalization.Limited_Controlled`. Note that this includes also private types that are not visibly controlled.

- `defaulted_parameter` controls subprogram or entry (**in**) parameters that provide a default value, while `defaulted_generic_parameter` controls generic formal objects that provide a default value.
- `derived_type` controls regular derived types, but not type extensions (derivations of tagged types). These are controlled by `extension` and `private_extension`.
- `discriminant` controls all declarations of types with discriminants, while `protected_discriminant` and `task_discriminant` control only discriminants of protected types and task types, respectively. `defaulted_discriminants` controls only discriminants where default values are provided. `unknown_discriminants` controls only unknown discriminants (AKA "`(<>)`" discriminants).
- `empty_private_part` controls package specification with an empty private part, i.e. where the word **private** appears, but the private part contains no declaration (even if it contains pragmas).
- `empty_visible_part` controls package specifications that contain no declaration in the visible part (before the word **private** if any), even if it contains pragmas.
- `enumeration_type` controls the declaration of enumeration types.
- `exception` controls exception declarations.
- `expression_function` controls declaration of expression functions
- `fixed_type` controls all declarations of fixed point types while `ordinary_fixed_type` controls only ordinary (binary) fixed point types, `ordinary_fixed_type_no_small` controls ordinary fixed point type without a representation clause for `'SMALL`, `ordinary_fixed_type_with_small` controls ordinary fixed point type with an explicit representation clause for `'SMALL`, and `decimal_fixed_type` controls only decimal fixed point types (those can never have a representation clause for `'SMALL`).
- `float_type` controls declarations of floating point types.
- `formal_function`, `formal_package`, `formal_procedure`, and `formal_type` control all generic formal functions, packages, procedures, and types, respectively. `box_defaulted_formal_function`, `box_defaulted_formal_procedure`, `name_defaulted_formal_function`, `name_defaulted_formal_procedure`, and `null_defaulted_formal_procedure` control generic formal functions and procedures with a box default, a name default, and a null default, respectively.
- `generic_function`, `generic_package`, `generic_procedure` control generic function (respectively package, procedure) declarations.
- `handlers` controls the presence of exception handlers in any handled sequence of statements.
- `in_out_parameter` and `out_parameter` control subprogram and entry parameters of modes **in out** and **out** (respectively), while `in_out_generic_parameter` and `out_generic_parameter` do the same for *generic* formal parameters. Note that to avoid unnecessary messages, if a subprogram has an explicit specification, the message is given on the specification and not repeated on the body.
- `incomplete_type` controls incomplete type declaration.
- `initialized_variable` controls variable declarations that include an initialization expression, unless they are of a class-wide type since initialization is required in that case.

- `instantiation` controls all instantiations, while `function_instantiation`, `package_instantiation`, `procedure_instantiation` control function (respectively package, procedure) instantiations.
- `integer_type` controls all declarations of integer types, while `signed_type` controls only signed integer types, and `modular_type` controls only modular types (both kinds); `binary_modular_type` controls only modular types whose modulus is a power of 2, and `non_binary_modular_type` controls only modular types whose modulus is not a power of 2.
- `initialized_record_component` and `initialized_protected_component` control the declaration of record (respectively protected) component that include a default initialization, while `uninitialized_record_component` and `uninitialized_protected_component` control the declaration of record (respectively protected) component that do not include a default initialization, unless they are of a limited type since initialization would not be allowed in that case.
- `limited_private_type` controls limited private type declarations, while `non_limited_private_type` controls regular (non limited) private type declarations. `tagged_private_type` controls tagged private type declarations.
- `multiple_names` controls declarations where more than one defining identifier is given in the same declaration.
- `multiple_protected_entries` controls protected definitions (from protected types or single protected objects) that have more than one entry declaration. Note that a protected definition with a single entry family declaration is counted as a single entry declaration.
- `named_number` controls declarations of named numbers, i.e. untyped constants.
- `non_joint_CE_NE_handler` controls exception handlers whose choices include `Constraint_Error` or `Numeric_Error`, but not both. This is intended for legacy Ada 83 code that required to always handle these exceptions together; it makes little sense for Ada95 or Ada2005 code (and to be honnest, this subrule is provided because Gnatcheck has it).
- `null_extension` controls record extensions (derived tagged types) that contain no new elements. Similarly, `null_ordinary_record_type` and `null_tagged_type` control ordinary records and tagged types that contain no elements. Note that the record definitions may be plain "**null record**" definitions, or full record definitions that contain only null components. However, a definition is not considered null if it contains a variant part.
- `null_procedure_body` controls procedure declarations whose sequence of statements contain only **null** statements (or blocks without declarations and containing only **null** statements). `null_procedure_declaration` controls Ada2005 null procedure declarations (i.e., "**procedure** P **is null**;"). `null_procedure` controls both.
- `operator` controls the definition of operators (things like `"+"`); note that the message is given on the specification if there is an explicit specification, on the body otherwise. `equality_operator` controls only equality operators (`"="` and `"/="`) and `predefined_operator` controls only operator definitions that overload a predefined operator (like `"+"` on a numeric type, for example).

- `package_statements` controls the presence of elaboration statements in the bodies of packages (or generic packages).
- `private_extension` controls private extensions, i.e. derivations from a tagged type with a **with private** extension part.
- `record_type` controls all record type declarations (tagged or not), while `ordinary_record_type` controls only non-tagged record types, and `tagged_type` controls only tagged record types.
- `interface_type` controls interface type declarations.
- `renaming` controls all renaming declarations, while `renaming_as_body` controls only those that are renamings as bodies of subprograms, `renaming_as_declaration` controls only those that are regular renamings of subprograms (i.e. not as bodies), `operator_renaming` controls only those that are renamings of an operator, `not_operator_renaming` controls only those that are *not* renamings of an operator, `function_call_renaming` controls renaming of the result of a function call, and `library_unit_renaming` controls renaming of library units. `non_identical_renaming` controls only renamings where the new name and the old name are not the same, and `non_identical_operator_renaming` does the same, but only for renamings of operators.
- `self_calling_function` controls functions whose body contains only a single (simple) **return** statement, and the return expression is a (recursive) call to the same function. Similarly, `self_calling_procedure` controls procedures whose body contains only a single statement which is a (recursive) call to the same procedure. Note that this corresponds to bodies automatically generated by gnatstub.
- `subtype` controls all explicit subtype declarations (i.e. not all anonymous subtypes that appear at various places in the language), while `unconstrained_subtype` controls only the subtype declarations that do not include a constraint.
- `task` controls task type declarations as well as single tasks declarations while `single_task` and `task_type` control only single task declarations or task type declarations respectively (and similarly for `protected`). `non_ravenscar_task` controls all task type and task object declarations from a unit to which no `pragma Profile (Ravenscar)` applies.
- `type` controls all type (but not subtype) declarations.
- `variable` controls all variable declarations, while `uninitialized_variable` controls only variable declarations that do not include an initialization expression, unless they are of a limited type since initialization would not be allowed in that case. `scalar_variable` controls the declarations of variables of a scalar type (integer, enumeration, float, fixed). `ordinary_record_variable` controls declarations of variables of an untagged record type. `tagged_variable` controls declarations of variables of a tagged type (including class-wide ones), while `class_wide_variable` controls only the declarations of variables of a class-wide type. `task_variable` and `protected_variable` control task and protected objects (respectively), whether given with a named or anonymous type.
- `variant_part` controls variant parts in record defintions.

Ex:

```
search declarations (task, exception);
check declarations (block procedure, block function, block package);
check declarations (public task);
```

5.10.3 Tips

Certain keywords are *not* exclusive, and it may be the case that several keywords apply to the same declaration; in this case, they are all reported. For example, if you specify:

```
check declarations (record_type, tagged_type);
```

tagged types will be reported both as "record_type" and "tagged_type".

There is no subrule for checking functions whose result type is from an anonymous access type; these are controlled by the rule `return_type (anonymous_access)`. See Section 5.48 [Return_Type], page 109.

Some of the keyword do not seem very useful; it would be strange to have a programming rule that prevents all type declarations... But bear in mind that the <location_kw> can be used to restrict the check to certain locations; moreover, AdaControl can be used not only for checking, but also for searching; finding all type declarations in a set of units can make sense. As another example, "search declarations (own variable);" will find all variables declared directly in package bodies.

Some modifiers do not make sense with certain declarations; for example, a "private out_parameter" is impossible (a parameter occurs in a subprogram declaration, not *directly* in a private part). This is not a problem as far as the rule is concerned, but don't expect to find any...

Generally, discriminants are considered components of record types. However, discriminants of an anonymous access type (so-called access discriminants) play such a special role in the language that they deserved their own control (`anonymous_access_discriminant`).

Private types are normally followed in determining the kind of access type (i.e., an access to a private type will be controlled according to the full declaration). However, this is not done for an access type that designates a private type defined in a language defined unit (since the full type depends on the implementation); these are controlled as "access_language_type" instead. Of course, language defined *visible* types are controlled normally.

5.10.4 Limitation

In some rare cases, AdaControl may not be able to evaluate the modulus of a modular type definition, thus preventing correct operation of "binary_modular_type" and "non_binary_modular_type" subrules. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.11 Default_Parameter

This rule checks usage (or non-usage) of defaulted parameters.

5.11.1 Syntax

```
<control_kind> default_parameter (<place>, <formal>, <usage>);
<place>  ::= <entity> | calls | instantiations
```

```
<formal> ::= <formal name> | all
<usage>  ::= used | positional | not_used
```

5.11.2 Action

The rule controls subprogram calls or generic instantiations that use the default value for the indicated parameter, or conversely don't use it, either in positional notation or in any notation. If a subprogram is called, or a generic instantiated, whose name matches <entity>, and it has a formal whose name is <formal name>, then:

- If the string `used` (case irrelevant) is given as the third parameter, the rule reports when there is no corresponding actual parameter (i.e. the default value is used for the parameter).
- If the string `positional` (case irrelevant) is given as the third parameter, the rule reports when there is an explicit corresponding actual parameter (i.e. the default is not used for the parameter), and the actual uses positional (not named) notation.
- If the string `not_used` (case irrelevant) is given as the third parameter, the rule reports when there is an explicit corresponding actual parameter (i.e. the default is not used for the parameter), independently of whether it uses positional or named notation.

As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. On the other hand, <formal> is the simple name of the formal parameter.

Alternatively, the <entity> can be specified as `calls`, to control all calls or `instantiations`, to control all instantiations. The <formal name> can be replaced by `all`, in which case all formals are controlled.

Ex:

```
check default_parameter (P, X, used);
check default_parameter (P, Y, not used);
search default_parameter (calls, all, positional);
```

5.11.3 Tip

If the <entity> is a generic subprogram, it is also possible to give a formal parameter (a parameter of the subprogram, not a generic parameter) as the <formal name>; in this case, all instantiations of the indicated generic subprogram will be controlled for the use of the indicated parameter.

5.12 Dependencies

This rule controls dependencies of units (i.e. **with** clauses, parents, child units...), either according to a set of allowed/forbidden units, or by count.

5.12.1 Syntax

```
<control_kind> dependencies (others, <unit> {,<unit>});
<control_kind> dependencies (with, <unit> {,<unit>});
<control_kind> dependencies (public_child | private_child);
<control_kind> dependencies (<counter>, <bound> [, <bound>]);
<counter> ::= raw | direct | parent
<bound>   ::= min | max <value>
```

5.12.2 Action

The kind of action depends on the specified subrule.

The "others" subrule controls semantic dependencies to units other than those indicated. This subrule can be specified only once, and at least one unit must be given.

The "with" subrule controls with clauses that reference the indicated units. At least one unit must be given.

Note that for these two rules, renamings are followed: if you give the basic name of a unit, it will be identified even if used with other names. Similarly, if you give the name of a generic, all of its instantiations will also be controlled.

"public_child" and "private_child" control units that depend on their own public (respectively private) child units. Since these subrules have no parameters, they can be given only once.

Other subrules control that the number of various dependencies is whithin a specified range. The second (and optionnally third) parameter give the minimum and/or maximum allowed values (i.e. the rule will control values outside the indicated interval). If not specified, the minimum value is defaulted to 0 and the maximum value to infinity.

- "raw" controls the number of units textually given in **with** clauses. Redundant **with** clauses are counted, and a child unit counts for one.
- "direct" controls the number of different units that this unit really depends on: if a unit is mentionned in several **with** clauses, it is counted only once, but if a child unit is mentionned, all parents of this child unit are added to the count.
- "parent" counts the number of parents of the current unit. A root unit has no parent, a child of a root unit has one parent, etc.

Ex:

```
check dependencies (others, Ada.Text_IO);
check dependencies (raw, max 15);

-- child units should not be nested more than 5 levels:
check dependencies (parent, max 5);

-- units that depend on nothing:
search dependencies (direct, min 1);

-- units that depend on their public children:
check dependencies (public_child);
```

5.12.3 Tips

If you give a name that's already a renaming to the "others" or "with" subrules, the rule will only apply to this name, not to what has been renamed. Therefore:

```
 -- Allow only Ada.Text_IO:
check dependencies (others, Ada.Text_IO);

-- But not if the plain name Text_IO is used:
check dependencies (with, Text_IO);
```

The notion of public or private for the rules "public_child" or "private_child" refer to the real unit, which is not necessarily the name used in the with clause, if for example you have a private library renaming of a public unit.

There is a slight overlap between this rule and the rule "entities'. But "entities" will find all uses of an entity (not necessarily a compilation unit), while "dependencies" will control occurrences only of compilation units, and only in **with** clauses. See Section 5.15 [Entities], page 62.

In certain contexts, only a set of the Ada predefined units is allowed. For example, it can be useful to forbid units defined in special needs annexes. The `rules` directory of Adacontrol contains files with "Dependencies" rules that forbid the use of various predefined Ada units. Comment out the lines for the units that you want to allow. You can then simply "source" these files from your own rule file (or copy the content) if you want to disallow these units. See Section 6.2 [Rules files provided with AdaControl], page 140.

5.13 Directly_Accessed_Globals

This rule checks that global variables in package bodies are accessed only through dedicated subprograms. Especially, it can be used to prevent race conditions in multi-tasking programs.

5.13.1 Syntax

```
<control_kind> directly_accessed_globals [(<kind> {,<kind>})];
<kind> ::= plain | accept | protected
```

5.13.2 Action

The rule controls global variables declared directly in (generic) package bodies that are accessed outside of dedicated callable entities (i.e. procedure or function, possibly protected, protected entries, and **accept** statements).

This rule can be specified only once. The parameters indicate which kinds of callable entity are allowed: "plain" for non-protected subprograms, "protected" for protected subprograms, and "accept" for **accept** statements). Without parameters, all forms are allowed.

More precisely, the rule ensures that the global variables are read from a single callable entity, and written by a single callable entity. Note that the same callable entity can read and write a variable, but in this case no other callable entity is allowed to read or write the variable.

- Subprograms used to read/write the variables must be declared at the same level as the variable itself (i.e. not nested), and must not be generic.
- Protected subprograms used to read/write the variables must both be part of the same single protected object, which must be declared at the same level as the variable itself (i.e. not nested); they are not allowed to be declared in a protected *type*, since if there are several protected objects of the same type, mutual exclusion would not be enforced.
- **accept** statements used to read/write the variables must both be part of the same single task object, which must be declared at the same level as the variable itself (i.e. not nested); they are not allowed to be declared in a task *type*, since if there are several task objects of the same type, mutual exclusion would not be enforced.

In short, this rule enforces that all global variables are accessed by dedicated access subprograms, and that only those subprograms access the variables directly. If given with the keyword "protected" and/or "accept", it enforces that global variables are accessed only by dedicated protected subprograms or tasks, ensuring that no race condition is possible.

Ex:

```
check directly_accessed_globals
```

5.13.3 Tips

Note that this rule controls global variables from package *bodies*, not those from the specification. This is intended, since it makes little sense to declare a variable in a specification, and then require it not to be accessed directly, but through provided subprograms. Obviously, in this case the variable should be moved to the body.

Note that AdaControl can check that no variable is declared in a package specification with the following rule:

```
check usage (variable, from_spec);
```

see Section 5.65 [Usage], page 135 for details.

5.13.4 Limitations

AdaControl cannot check entities accessed through dynamic names (dynamic renaming, access on aliased variables). Use of such constructs is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

Due to a weakness in the ASIS standard, it is not possible to know the mode (**in**, **out**) of variables used as parameters of dispatching calls. Such variables are considered to be read and written at the point of the call, therefore possibly creating false positives (which is safer than false negatives). Use of such constructs is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.14 Duplicate_Initialization_Calls

This rule checks that some procedures (notably initialization procedures) are not called several times in identical conditions.

5.14.1 Syntax

```
<control_kind> duplicate_initialization_calls (<entity> {, <entity>});
```

5.14.2 Action

This rule controls calls to initialization procedures that are duplicated. The <entity> parameters are the initialization procedures to be controlled. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

More precisely, the initialization procedures must follow one of these patterns:

- The procedure only has **in** parameters. All actual parameters used in calls are static, and not two calls have the same values for all parameters.
- The procedure has exactly one **out** parameter (and no **in out** parameter). Not two calls refer the same actual variable for the **out** parameter.

The rule controls any violation of these patterns. If a procedure passed as parameter does not have a profile that corresponds to one of the above patterns, it is an error.

Ex:

```
check duplicate_initialization_calls (pack.init_proc);
```

5.14.3 Limitation

If a variable passed as an **out** parameter is not statically determinable, it is not controlled by the rule. Such a case is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.15 Entities

This rule is used to control usage of Ada entities, i.e. any declared element (type, variables, packages, etc).

5.15.1 Syntax

```
<control_kind> entities ({<location>} <entity> {, {<location>} <entity>});
<location> ::= block   | library | local      | nested    | own |
               private | public  | in_generic | task_body
```

5.15.2 Action

This rule controls all uses of the indicated entities,or only those that appear within the specified locations. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

When present, the <location_kw> restricts the places where the entity is controlled. Several <location_kw> can be given, in which case the entity is controlled at places where all the keywords apply.

- `block`: the entity appears in a block statement.
- `library`: the entity appears at library level.
- `local`: the entity appears in a local scope (i.e. not in (generic) packages, possibly nested)
- `nested`: the entity appears in a nested context (i.e. not at library level).
- `own`: the entity appers in a (generic) package body.
- `public`: the entity appears in the visible part of a (generic) package.
- `private`: the entity appears directly in a private part.
- `in_generic`: the entity appears directly or indirectly in a generic specification or body.
- `task_body`: the entity appears directly in a task body.

Note that this rules reports on the use of the *entity*, not the *name*: if an entity has been renamed, it will be found under its various names. Similarly, if the given entity is a generic unit or an entity declared inside a generic unit, all corresponding uses in all instances will be reported.

Ex:

```
search entities (Debug.Trace);
check  entities (Ada.Text_IO.Float_IO.Put);
```

The second line will report on any use of a `Put` from any instantiation of `Float_IO`.

5.15.3 Tips

This rule is not intended to replace cross-references, but can be quite handy to check, for example, that a program does not contain any more calls to debugging procedures before fielding it.

This rule can also be used to check for all occurrences of certain attributes with the "`all <Attribute>`" syntax. For example, the following will report on any usage of `'Unchecked_Access`:

```
check entities (all 'Unchecked_Access);
```

If you want to make sure that certain compilation units are not used, it is preferable to use the rule "Depencies (with,...)" rather than "Entities", because "Entities" will control all uses of the unit, while "Dependencies" will control only those in **with** clauses (which is of course sufficient).

In certain contexts, it can be useful to forbid certain entities, like those from `Standard`, `System`, or entities defined in special needs annexes packages. The `rules` directory of Adacontrol contains files with "Dependencies" and "Entities" rules that forbid the use of various predefined Ada elements. Comment out the lines for the elements that you want to allow. You can then simply "source" these files from your own rule file (or copy the content) if you want to disallow these elements. See Section 6.2 [Rules files provided with AdaControl], page 140.

5.15.4 Limitation

GNAT defines `Unchecked_Conversion` and `Unchecked_Deallocation` as separate entities, rather than renamings of `Ada.Unchecked_Conversion` and `Ada.Unchecked_Deallocation`. As a consequence, it is necessary to specify explicitely both forms if you want to make sure that the corresponding generics are not used.

5.16 Entity_Inside_Exception

This rule controls entities that appear within exception handlers.

5.16.1 Syntax

```
<control_kind> entity_inside_exception (<spec> {, <spec>});
<spec> ::= [not] <entity> | calls | entry_calls
```

5.16.2 Action

This rule controls exception handlers that contain references to one or several Ada entities specified as parameters. If the keyword "calls" is given, it stands for all subprogram and entry calls. If the keyword "entry_calls" is given, it stands for all entry calls (task or protected). If an <entity> (or "calls" or "entry_calls") is preceded by the keyword "not", it is not included in the list of controlled entities (i.e. the entity is allowed in the exception hhandler). This allows to make exceptions to a more general specification of an entity, or to allow calls to well-defined procedures if the keyword "calls" is given.

Ex:

```
-- No Put_Line in exception handlers:
check entity_inside_exception (ada.text_io.put_line);

-- No entry calls in exception handlers:
check entity_inside_exception (entry_calls);

-- No calls allowed, except to the Report_Exception procedure:
check entity_inside_exception (calls, not Reports.Report_Exception);

-- No Put allowed, except the one on Strings:
check entity_inside_exception (all Put,
                               not Ada.Text_IO.Put{Standard.String});
```

5.17 Exception_Propagation

This rule controls that certain program units are guaranteed to never propagate exceptions, or that local exceptions cannot propagate out of their scope.

5.17.1 Syntax

```
<control_kind> exception_propagation
   (local_exception);
<control_kind> exception_propagation
   ([<level>,] interface, <convention> {, <convention> });
<control_kind> exception_propagation
   ([<level>,] parameter, <entity> {, <entity>});
<control_kind> exception_propagation
   ([<level>,] task);
<control_kind> exception_propagation
   (<level>, declaration);
```

5.17.2 Action

The "local_exception" subrule controls a design pattern that ensures that a local exception cannot propagate outside the scope where it is declared. If an exception is declared within a block, a subprogram body, an entry body, or a task body, then this body must have either a handler for this exception or for **others**; this handler must not reraise the exception; and no handler is allowed to raise explicitely the exception. The subrule controls explicit **raise** statements and calls to `Raise_Exception` and `Reraise_Occurrence`, but it does not control exceptions raised as a consequence of calling other subprograms.

The other subrules control subprograms, tasks, or all declarations that can propagate exceptions, while being used in contexts where it is desirable to ensure that no exception can be propagated.

A subprogram or task is considered as *not* propagating if:

1. it has an exception handler with a "**when others**" choice
2. no exception handler contains a **raise** statement, nor any call to `Ada.Exception.Raise_Exception` or `Ada.Exception.Reraise_Occurrence`.

3. no declaration from its own declarative part propagates exceptions.

A declaration is considered propagating if it includes elements that could propagate exceptions. This is impossible to assess fully using only static analysis, therefore the <level> parameter determines how pessimistic (or optimistic) AdaControl is in determining the possibility of exceptions. Possible values of the <level> parameter, and their effect, are:

- 0: expressions in declarative parts are not considered as propagating (anything allowed, this is the default value for "interface", "parameter" and "task". Not allowed for "declaration").
- 1: all function calls (including operators) in declarations are considered as potentially propagating exceptions, except those appearing in named number declarations or scalar types declarations, since those are required by the language to be static.
- 2: same as 1, plus every use of variables in expressions is considered as potentially propagating.
- 3: same as 2, plus any declaration of objects (constants or variables) is considered potentially propagating (not very useful for "declaration").

These subrules serve several purposes:

- The "interface" subrule analyzes all subprograms to which an `Interface` or `Export` pragma applies (with the given convention(s)), and reports on those that can propagate exceptions.

  Since it is dangerous to call an Ada subprogram that can propagate exceptions from a language that has no exception (and especially C), any such subprogram should have a "catch-all" exception handler.
- The "parameter" subrule accepts one or more fully qualified formal parameter names (i.e. in the form of the parameter name prefixed by the full name of its subprogram, see Appendix A [Specifying an Ada entity name], page 145). The subrule reports any subprogram that can propagate exceptions and is used as the prefix of a `'Access` or `'Address` attribute that appears as part of an actual value for the indicated formal. Similarly, the indicated formal can also be the name of a formal procedure or function of a generic. In this case, the rule will report on any subprogram that can propagate exceptions and is used as an actual in an instantiation for the given formal.

  Many systems (typically windowing systems) use call-back subprograms. Although the native interface is generally hidden behind an Ada binding, the call-back subprograms will eventually be called from another language, and like for the "interface" subrule, any such subprogram should have a "catch-all" exception handler.
- The "task" subrule reports any task that can propagate exceptions.

  Since tasks die silently if an exception is propagated out of their body, it is generally desirable to ensure that every task has an exception handler that (at least) reports that the task is being completed due to an exception.
- The "declaration" subrule reports any declaration that can propagate exceptions, irrespectively of where it appears. In this case, the specification of <level> is required and cannot be 0.

  It is sometimes desirable to make sure that no declaration raises an exception, ever.

Ex:

```
-- Make sure that C-compatible subprograms don't propagate exceptions:
check exception_propagation (interface, C);

-- Parameter CB of of procedure Pack.Register is used as a call-back
-- Make sure that not procedure passed to it can propagate exceptions.
check exception_propagation (parameter, Pack.Register.CB);

-- Make sure that tasks do not die silently due to unhandled exception:
check exception_propagation (task);

-- Make sure that no exception is raised by elaboration of declarations:
check exception_propagation (2, declaration);
```

The first example will report on any subprogram to which a **pragma** `Interface (C,...)` applies that can propagate exceptions.

If `Proc` is a procedure that can propagate exceptions, the second example will report on every call like:

```
Pack.Register (CB => Proc'Access);
```

The third example will report on any task that can terminate silently due to an unhandled exception.

The fourth example will report on any declaration that makes use of function calls or variables.

5.17.3 Tips

Note that the registration procedure for a call-back can be designated by an access type, but in this case, use the name of the formal for the access type. For example, given:

```
package Pack is
   type Acc_Proc is access procedure;
   type Acc_Reg is access procedure (CB : Acc_Proc);
   ...
   Ptr : Acc_Reg := ...;
```

You can give a rule such as:

```
check exception_propagation (parameter, Pack.Acc_Reg.CB);
```

All procedures registered by a call to `Pack.Ptr.all` will be considered.

The declaration of a **for** loop parameter is not checked by this rule. In other words, the rule "check exception_propagation (2, declaration)" will not issue a message for:

```
for I in Positive range 1 .. X loop ...
```

although formally the *declaration* of I could raise Constraint_Error if X is negative. We consider that for the casual user, Constraint_Error appears to be raised by the **for** loop *statement*.

5.17.4 Limitations

An exception may be raised in a subprogram considered as not propagating by this rule, if an exception handler calls a subprogram that propagates an exception.

The rule will not consider subprograms whose body is missing, or that are not statically known (i.e. if a subprogram is registered through a dereference of a pointer to subprogram), like in the following example:

```
Pack.Register (CB => Pointer.all'Access);
```

Due to a weakness of the ASIS standard, references to subprograms that appear in dispatching calls are not considered. This limitation will be removed as soon as we find a way to work around this problem, but the issue is quite difficult!

These last two cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.18 Expressions

This rule controls usage of various kinds of expressions.

5.18.1 Syntax

```
<control_kind> expressions (<subrule> {, <subrule>});
<subrule> ::= {<category>} <expression_kw>
<expression_kw> ::=
   and                                | and_array                    |
   and_binary                         | and_boolean                  |
   and_then                           | array_aggregate              |
   array_named_others                 | array_non_static_range       |
   array_others                       | array_partial_others         |
   array_positional_others            | array_range                  |
   case                               | complex_parameter            |
   downward_conversion                | extendable_aggregate         |
   extension_aggregate                | explicit_dereference         |
   fixed_multiplying_op               | for_all                      |
   for_some                           | if                           |
   if_elsif                           | if_no_else                   |
   implicit_dereference               | in                           |
   inconsistent_attribute_dimension   | inherited_function_call      |
   mixed_operators                    | not_in                       |
   or                                 | or_array                     |
   or_binary                          | or_boolean                   |
   or_else                            | parameter_view_conversion    |
   prefixed_operator                  | real_equality                |
   record_partial_others              | record_aggregate             |
   record_others                      | slice                        |
   static_membership                  | type_conversion              |
   upward_conversion                  | unconverted_multiplying_op   |
   underived_conversion               | universal_range              |
   unqualified_aggregate              | xor                          |
   xor_array                          | xor_binary                   |
   xor_boolean
<category> ::=
```

```
<>     | ()     | range  | mod | delta   | digits | array    |
record | tagged | access | new | private | task   | protected
```

5.18.2 Action

This rule controls usage of certain forms of expressions. The rule can be specified at most once for each subrule (i.e. subrules that accept categories can be specified once for each combination of categories and expression keyword).

Categories are used by certain subrules to further refine the control. They define categories of types to which they apply:

- "`<>`": Any type
- "()": Enumerated types
- "range": Signed integer types
- "mod": Modular types
- "delta": Fixed point types (no possibility to differentiate ordinary and decimal fixed point types yet).
- "digits": Floating point types
- "array": Array types
- "record": (untagged) record types
- "tagged": Tagged types (including type extensions)
- "access": Access types
- "new": Derived types
- "private": Private types
- "task": Task types
- "protected": Protected types

The subrule define the kind of expression being controlled:

- `and`, `or`, `xor`, `and_then`, `or_else`, `in`, and `not_in` control usage of the corresponding logical operator (or short circuit form, or membership test).
- `and_array`, `or_array`, and `xor_array` do the same, but only for operators whose result type is an array type.
- `and_binary`, `or_binary`, and `xor_binary` do the same, but only for operators whose result type is a modular type.
- `and_boolean`, `or_boolean`, and `xor_boolean` do the same, but only for operators whose result type is `Standard.Boolean`.
- `array_aggregate` and `record_aggregate` control array and record aggregates, respectively, while `unqualified_aggregate` controls aggregates (both arrays and records) that do not appear directly within a qualified expression. `extension_aggregate` controls extension aggregates, while `extendable_aggregate` controls aggregates that are *not* extension aggregates, but whose type is a non-root tagged type, or are extension aggregates whose ancestor part is not their immediate parent (such aggregates could be written as extension aggregates).
- `array_others` and `record_others` control the occurrence of a **others** `=>` association in array and record aggregates, respectively.

- `array_partial_others` and `record_partial_others` do the same, but only if there are other associations in addition to the **others =>** in the aggregate. `array_named_others` and `array_positional_others` do the same, but only for named (respectively positional) array aggregates.
- `array_range` controls array aggregates that include a range (i.e. an association like `A .. B =>`). `array_non_static_range` does the same, but only if (at least) one of the bounds is not static.
- `case` controls **case** expressions (introduced in Ada 2012).
- `complex_parameter` controls complex expressions used as actual parameters in subprogram (or entry) calls. A complex expression is any expression that includes a function call (including operators). This rule is not applied to the parameters of operators, since otherwise it would forbid any expression with more than a single operator.
- `explicit_dereference` controls explicit dereferences of access values (i.e. with an explicit **.all**).
- `fixed_multiplying_op` controls calls to predefined fixed-point multiplication and division (regular fixed-point or decimal-fixed point). `unconverted_fixed_multiplying_op` does the same, but only when both operands are objects (not literals) of a fixed_point type (not Integer); this is when type conversion is required by Ada 83.
- `for_all` and `for_some` control the two forms of quantifiers introduced by Ada 2012.
- `if` controls all **if** expressions (introduced in Ada 2012), while `if_elsif` only controls those that have an **elsif** part, and `if_no_else` only controls those that have no **else** part.
- `implicit_dereference` controls implicit dereferences of access values (i.e. when the **.all** is omitted).
- `inconsistent_attribute_dimension` controls when no dimension is explicitely given for a `'First`, `'Last`, `'Range` or `'Length` attribute and the attribute applies to a multi-dimensional array, or conversely, when an explicit dimension is given, but the attribute applies to a one-dimensional array.
- `inherited_function_call` controls calls to functions that have been inherited by a derived type and not redefined. If a category is specified, only calls whose result type belongs to the category are controlled.

  Derived types are followed, i.e. the "real" category from the original type is used for the matching; as a consequence, the "new" category cannot be specified for this subrule.
- `mixed_operators` controls expressions that involve several different operators, without parentheses. In a sense, it extends the language rule that forbids mixing **and** and **or** in logical expressions to all other operators. Note that for the purpose of this subrule, membership tests (**in**, **not in**) and short circuit forms (**and then**, **or else**) are considered operators.
- `prefixed_operator` controls calls to operators that use prefixed notation (i.e. `"+"(A, B)`). If a category is specified, only calls whose result type belongs to the category are controlled.

  Derived types are followed, i.e. the "real" category from the original type is used for the matching; as a consequence, the "new" category cannot be specified for this subrule.

- `real_equality` controls usage of predefined exact equality or inequality ("=" or "/=") between real (floating point or fixed point) values.
- `slice` controls usage of array slices.
- `static_membership` controls membership tests (**in** and **not in**) where the expression on the left is statically known to belong to the range (or subtype) on the right, and is therefore always True (or false for **not in**).
- `type_conversion` controls all (sub)type conversions, while `underived_conversion` controls conversions between types that do *not* belong to the same derivation family. `downward_conversion` and `upward_conversion` control conversions between types that belong to the same family, converting away from the root or toward the root, respectively. `parameter_view_conversion` controls conversions that appear as **out** or **in out** actual parameters.

  One or two categories can be specified; if only one category is specified, only conversions whose result type belong to that category are controlled. If two categories are specified, only conversions whose souce type belongs to the first category and whose target type belong to the second category are controlled.

  Derived types are followed, i.e. the "real" category from the original type is used for the matching; as a consequence, the "new" category cannot be specified for this subrule.
- `universal_range` controls discrete ranges that are a part of an index constraint, constrained array definition, or for-loop parameter specification (but not type or subtype defintions), and whose bounds are both of type universal_integer.

Ex:

```
search expressions (real_equality, slice);
check  expressions (mixed_operators);

-- Find logical operators that could be replaced by short-circuits forms:
check expressions (and_boolean, or_boolean);

-- Find all conversions between integer and floating point types
search expression (range digits type_conversion);

-- Find all conversions from a fixed point type:
search expressions (delta <> type_conversion);

-- Find all view conversions between array types:
search expressions (array parameter_view_conversions);

-- Find all "structural" conversions between arrays
search expressions (array underived_conversion);

-- Some think that downward conversions of tagged types are evil:
check expressions (tagged downward_conversion);
```

5.18.3 Tips

The `real_equality` subrule does not control calls to an equality operator that has been defined by the user; actually, it would make little sense to write a function and then forbid its use! However, if control of calls to such a function is desired, it can be easily accomplished by using the `entities` rule. See Section 5.15 [Entities], page 62.

This rule does not check the use of allocators (**new**), use the rule `Allocators` instead. See Section 5.2 [Allocators], page 38.

"inherited_function_call" controls only *function* calls. For procedure calls, see rule Section 5.53 [Statements], page 117.

Specifying `array_partial_others` is the same as specifying both `array_named_others` and `array_positional_others`. It is retained for compatibility, and also for symetry with `record_partial_others`.

Per language rules, underived conversions are allowed only between numeric types, and between structurally equivalent array types.

"static_membership" is handy for finding a common misuse of membership tests, where the user assigns an external value (obtained with Unchecked_Conversion for example) to a variable, then checks that the variable belongs to its subtype to make sure the value is valid. Such a check can be optimized away by the compiler; the 'Valid attribute should be used instead.

5.18.4 Limitations

"static_membership" does not control the complex membership tests with several choices that are possible with Ada 2012.

5.19 Generic_Aliasing

This rule controls instantiations where the same actual is given to more than one formal.

5.19.1 Syntax

```
<control_kind> generic_aliasing [(<subrule> {, <subrule>})];
<subrule>   ::= [<condition>] <entity>
<condition> ::= unlikely | possible | certain
<entity>    ::= all | variable | type | subprogram | package
```

5.19.2 Action

This rule identifies instantiations where the same variable, type, subprogram, or package is given several times (to different formal parameters). Such aliasing of variables is dangerous, since it can induce subtile bugs. Other elements are less dangerous, although often questionable (depending on the generic).

The `<entity>` parameter indicates for which elements aliasing is controlled; "all" stands for all kinds of elements.

There are many cases where aliasing cannot be determined statically. The optional parameter specifies how aggressively the rule will check for possible aliasings (see Section 5.40 [Parameter_Aliasing], page 98 for a more detailed description of these modifiers). Possible values are (case irrelevant):

- Certain (default): Only cases where aliasing is statically certain are output.
- Possible: In addition, cases where aliasing may occur depending on the value of an indexed component are output. This can be specified only for variables.
- Unlikely: In addition, cases where aliasing may occur due to access variables designating the same element are output. This can be specified only for variables and subprograms.

Without any parameter, the rule is the same as "certain all". The rule can be specified only once for each combination of <condition> and <entity>.

Ex:

```
check  generic_aliasing (certain  variable);
search generic_aliasing (possible variable, type, subprogram, package);
```

5.19.3 Limitations

Due to a limitation of ASIS for Gnat, AdaControl might not be able to differentiate predefined operators of different types, and may thus give false positives if a generic is instantiated with, for example, two different functions that are actually "+" on Integer and "+" on Float. This possibility of false positives is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.20 Global_References

This rule controls accesses to global elements that may be subject to race conditions, or otherwise shared.

5.20.1 Syntax

```
<control_kind> global_references (<subrule> {, <root>});
<subrule> ::= all | read | written | multiple | multiple_non_atomic
<root>    ::=  <entity> | function | procedure | task | protected
```

5.20.2 Action

This rule controls access to global variables from several entities (the roots). The <entity> must be subprograms, task types, single task objects, protected types, or single protected objects. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. The special keywords `function`, `procedure`, `task`, and `protected` are used to refer to all functions, procedures, tasks, and protected entities, respectively.

The <subrule> determines the kind of references that are controlled. If it is `all`, all references to global elements from the indicated entities are reported. If <subrule> is `read` or `written`, only read (respectively write) accesses are reported. If <subrule> is `multiple`, only global elements that are accessed by more than one of the indicated entities (i.e. shared elements) are reported. Note however that if a reference is found from a task type or protected type, it is always reported, since there are potentially several objects of the same type. If <subrule> is `multiple_non_atomic`, references reported are the same as with `multiple`, except that global variables that are `atomic` or `atomic_components` and written from at most one of the indicated entities are not reported. Note that this latter case corresponds to a safe reader/writer use of atomic variables.

This rule follows the call graph, and therefore finds references from subprogram and protected calls made (directly or indirectly) from the indicated entities. However, calls to subprograms from the Ada standard library are not followed.

Ex:

```
-- Find global variables used by P1 or P2:
search global_references (all, P1, P2);

-- Find global variables modified by functions:
check global_references (written, function);

-- Find possible race conditions:
check global_references (multiple, task, protected);
```

This rule can be given several times, and conflicts (with `multiple`) are reported on a per-rule basis, i.e. given:

```
check global_references (multiple, P1, P2);
check global_references (multiple, P1, P3);
```

the first rule will report on global variables shared between P1 and P2, and the second rule will report on global variables shared between P1 and P3.

5.20.3 Tips

The notion of "global" is relative, i.e. it designates every variable whose scope encloses (strictly) the indicated entities. This means that a same reference may or may not be global, depending on the indicated entity. Consider:

```
procedure Outer is
   Inner_V : Integer;

   procedure Inner_P is
   begin
      Inner_V := 1;
   end Inner_P;
begin
   Inner_P;
end Outer;
```

The rule

```
check global_references (all, outer);
```

will not report any global reference, while the rule

```
check global_references (all, outer.inner_p);
```

will report a reference to `Inner_V`. This is as it should be, since there is no race condition if several tasks call `Outer`, while there is a risk if several tasks (declared inside `Outer`) call `Inner_P`.

Specifying:

```
check global_references (all, function);
```

will report on any function that access variables outside of their scope, i.e. all functions that have potential side effects. On the other hand, this check must follow the whole call graph for any function encountered, and can therefore be quite costly in execution time.

5.20.4 Limitations

Calls through pointers to subprograms and dispatching calls are unknown statically; they are assumed to not access any global. Such calls are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.21 Header_Comments

This rule controls that every compilation unit starts with a standardized comment.

5.21.1 Syntax

```
<control_kind> header_comments (minimum, <comment lines>);
<control_kind> header_comments (model, "<file name>");
```

5.21.2 Action

If the keyword "minimum" is given as first parameter, this rule controls that every compilation unit starts with at least the number of comment lines indicated by the second parameter. If several forms of headers are possible, checking that the headers follow the project's standard requires manual inspection, but this rule is useful to control that unit headers have not been inadvertantly forgotten.

If the keyword "model" is given as first parameter, the second parameter is a string, interpreted as a file name. If the file name is not an absolute path, it is interpreted as relative to the directory of the file that contains the rule, or to the current directory if the rule is given on the command line. Each line of the indicated file is a regular expression, and the rule controls that the corresponding line of the source file matches the expression. See Appendix B [Syntax of regular expressions], page 149. In addition, it is possible to specify a repetition for a line. If the first character of a line is a `'{'`, the line must have the following syntax:

```
{<min>,[<max>]}
```

where <min> and <max> specify the minimum and maximum number of occurrences of the pattern in the line that follows this one. <min> must be at least 0, and <max> must be at least 1, and be equal or greater than <min>. If <max> is omitted, it means that the line may occur any number of times.

As a convenience, if the first character of a line is a `'*'` it means that the next line is a pattern that can occur any number of times (same as `{0,}`). If the first character is a `'+'`, it means that the next line is a pattern that must occur at least once (same as `{1,}`). If the first character is a `'?'`, it means that the next line is an optional pattern (same as `{0,1}`).

Note that the repetition lines all start with a special character which is not allowed at the start of a regular expression; there is therefore no ambiguity. Everything after the special character (or the closing `'}'`) is ignored, and can be used to provide comments.

This rule can be given at most once with "minimum" for each of "check", "search", and "count". The rule can be given only once with "model" (but it can be given together with one or more "minimum" rules).

Ex:

```
check header_comments (minimum, 10);
search header_comments (model, "header.pat");
count header_comments (minimum, 20);
```

This makes an error for every unit that starts with less than 10 comment lines, and a warning for units that do not follow the pattern contained in the file `header.pat`. A count of units that start with less than 20 comment lines is reported.

Example of a pattern file:

```
{1,3} 1 to 3 occurrences of next line
^--$
^-- Author: .+$
^-- Date: \d{2}/\d{2}/\\d{4}$
```

5.21.3 Tips

Remember that the lines of the file are regular expressions; every character that is specially interpreted (like "`+`", "*", etc.) must be quoted with "\" if it must appear textually. To ease the process of generating the model file, the directory `source` contains a script file for sed named `makepat.sed`; if you run this script on a file that contains a standard header, it will produce a pattern file where each line starts with "^", ends with "$", and every special character is quoted with "\".

When the model contains an indication of repeated lines ("*"), the repetition is not "greedy", i.e. matching will stop as soon as what follows the repetition matches. This is very useful to check header comments that have sections, but where you don't want to impose a precise content to each section. Imagine for example that the structure is:

- A comment with "HISTORY"
- Any number of comment lines
- A comment with "AUTHORS"
- Any number of comment lines

the following pattern will work as expected:

```
^-- HISTORY$
*
^--
^-- AUTHORS
*
^--
```

5.21.4 Limitation

Since the "model" subrule analyzes the content of comments, there is a conflict with the disabling mechanism of AdaControl that uses special comments. See Section 4.2.4 [Disabling controls], page 30.

Specifically, line disabling is not possible at all. Block disabling is possible, provided the disabling line is allowed by the pattern. In short, if you want to be able to disable this rule, the first lines of the model file should be:

```
?
--##
```

i.e. allow an optional block disabling comment as the first line of the file. Note that there is no need to re-enable this rule, since it is checked only at the start of a compilation unit.

5.22 Improper_Initialization

This rule enforces a coding pattern that ensures that variables and **out** parameters are properly initialized befor use.

5.22.1 Syntax

```
<control_kind> improper_initialization [(<subrule> {,<subrule>})]
<subrule> ::= {<extra>} <target>
<extra>   ::= access | limited | package | return
<target>  ::= out_parameter | variable | initialized_variable
```

5.22.2 Action

This rule controls variables and/or **out** parameters that are not "properly" initialized, i.e. those that are not "safely" initialized, those that have a useless initialization in their declaration, and those where the value is known to be used before having been assigned. The notion of variable includes the return object of an extended return statement (Ada 2005+).

A variable (or **out** parameter) is considered safely initialized if there is an initialization expression in its declaration, or if it is given a value in the first statements of the corresponding body, before any "non-trivial" statement. The goal is not to perform a complete data-flow analysis, but rather to follow a design pattern where all variables are initialized before entering the "active" part of the algorithm. This makes it easier to ensure that variables are properly initialized.

"Trivial" statements are:

- `null` statements;
- assignment statements;
- procedure calls;
- return statements;
- extended return statements, unless they contain a nested non-trivial statement.
- **if** and **case** statements, unless they contain a nested non-trivial statement.

The <target> parameters determines what is to be checked:

- `out_parameter` controls that **out** parameters are safely initialized before the first non-trivial statement, and before every (trivial) `return` statement.
- `variable` controls that local variables are safely initialized before the first non-trivial statement. If the <extra> modifier `return` is specified, only return objects of extended return statements are controlled.
- `initialized_variable` controls variables that are safely initialized before the first non-trivial statement, but also have an explicit (and therefore useless) explicit initialization

in their declaration. If the modifier `return` is specified, only return objects of extended return statements are controlled.

In all cases, variables used in trivial statements before being initialized are reported.

A variable is considered initialized if it is the target of an assignment statement, or if it is used as an actual for an **out** (but not **in out**) parameter of a procedure call. Variables assigned in **if** or **case** statements must receive a value in all paths to be considered initialized after the statement. Note that the variable must be assigned to globally, i.e. assigning to some elements of an array, or some fields of a record, does not count as an initialization of the variable.

Some variables are *not* controlled, unless the corresponding <extra> modifier is given:

- Variables declared immediately within a (generic) package specification or body, since in general, package state variables are initialized through calls to dedicated procedures. Use the "package" modifier to control also package variables.
- Variables of an access types, or arrays whose components are of an access type, since these are always initialized by the compiler. Use the "access" modifier to control also variables of an access type.
- Variables of a limited type, since global assignment is not available for them. Use the "limited" modifier to control also variables of a limited type.

This rule can be given only once for each value of <target>. Without parameters, it is equivalent to giving all, without any <extra>.

Ex:

```
check improper_initialization (out_parameter);
check improper_initialization (access limited variable);
search improper_initialization (initialized_variable);
```

5.22.3 Tips

`variable` and `initialized_variable` control also return objects from extended return statements, since it would be strange to guarantee safe initialization of local variables and not return objects. On the other hand, the design pattern enforced by this rule may seem to limitative for regular variables, but it might be desirable to enforce it for return objects; hence the possibility to limit the rule to return objects by specifying the `return` modifier.

5.22.4 Limitations

Due to a weakness of the ASIS standard, dispatching calls and calls to procedures that are attributes are not considered for the initialization of variables. Note that for attributes, only `'Read` and `'Input` have an **out** parameter.

In the rare case where a variable is initialized by a dispatching call or an attribute call, this limitation will result in a false positive. Such a case is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126. It is then easy to disable the rule for this variable. See Section 4.2.4 [Disabling controls], page 30.

The rule analyzes only initializations and uses that are directly in the unit, not those from nested units, since these are in the general case not statically checkable.

There are other cases where an object is automatically initialized by the declaration, like controlled types that have redefined the `Initialize` procedure, records where all components have a default initialization, etc. The rule does not consider these as automatically initialized, as it does for access types. Maybe later...

5.23 Instantiations

This rule controls all instantiations of a generic, or only instantiations that are made with specific values of the parameters. Control can be restricted to instantiations in specified places.

5.23.1 Syntax

```
<control_kind> instantiations (<generic_spec>);
<generic_spec> ::= {<location_kw>} <entity> {, <formal_spec>}
<formal_spec>  ::= <entity> | <category> | <> | =
<location_kw>  ::= all | block   | library | local      | nested |
                   own | private | public  | in_generic | task_body
<category>     ::= ()      | access    | array | delta  | digits | mod  |
                   private | protected | range | record | tagged | task
```

5.23.2 Action

The rule controls instantiations of the specified <entity>. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

The <location_kw> restricts the places where the occurrence of the instantiation is controlled. Several <location_kw> can be given, in which case the instantiation is controlled at places where all the keywords apply. If there is no <location_kw>, it is assumed to be "all".

- `all`: puts no special restriction to the location. This keyword can be specified for readability purposes, and if specified must appear alone (not with other <location_kw>).
- `block`: only instantiations appearing in block statements are controlled.
- `library`: only library level instantiations are controlled.
- `local`: only local instantiations are controlled (i.e. only instantiations appearing in (generic) packages, possibly nested, are allowed).
- `nested`: only instantiations nested in another declaration are controlled (i.e. only library level instantiations are allowed).
- `own`: only instantiations that are local to a (generic) package body are controlled.
- `public`: only declarations appearing in the visible part of (generic) packages are controlled.
- `private`: only instantiations appearing directly in a private part are controlled.
- `in_generic`: only instantiations appearing directly or indirectly in a generic specification or body are controlled.
- `task_body`: only instantiations appearing directly in a task body are controlled. Note that it would not make sense to have a <location_kw> for task *specifications*, since instantiations are not allowed there.

An instantiation matches if it appears at a specified location (if any) and either:

1. No <formal_spec> is given in the rule
2. The actual parameters of the instantiation match the corresponding <formal_spec>, in order (there can be more actual parameters in the instantiation than specified in the rule). An actual parameter matches an <entity> at a given place if it is the same entity, or if the <entity> designates a (sub)type and the actual is a subtype of this type. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. In addition, it matches if the actual is a type name that belongs to the indicated category:
   - "()": The parameter is of an enumerated type.
   - "access": The parameter is of an access type.
   - "array": The parameter is of an array type.
   - "delta": The parameter is of a fixed point type (it is not currently possible to distinguish ordinary fixed point types from decimal fixed point types).
   - "digits": The parameter is of a floating point type.
   - "mod": The parameter is of a modular type.
   - "private": The parameter is of a private type (including private extensions).
   - "protected": The parameter is of a protected type.
   - "range": The parameter is of a signed integer type.
   - "record": The parameter is of an (untagged) record type.
   - "tagged": The parameter is of a tagged type (including type extensions).
   - "task": The parameter is of a task type.

   In addition, two special signs can be given instead of an <entity> (or <category>): a box (<>) matches any actual parameter (i.e. it stands for any value), and an equal sign (=) matches if there has been already an instantiation with the same value for this parameter (i.e. it matches the second time it is encountered).

If an actual is an expression (which is possible only for a formal **in** object), it cannot be matched.

Ex:

```
-- Check all instantiations of Unchecked_Deallocation:
search instantiations (ada.unchecked_deallocation);

-- Check all instantiations of Unchecked_Conversion from or to String:
check instantiations (ada.unchecked_conversion, standard.string);
check instantiations (ada.unchecked_conversion, <>, standard.string);

-- Check all instantiations of Unchecked_Conversion from address
-- to an integer type:
check instantiations (ada.unchecked_conversion, system.address, range);

-- Check that Unchecked_Conversion is instantiated only once
-- for any pair of arguments:
check instantiations (ada.unchecked_conversion, =, =);
```

5.23.3 Tips

The various forms of <formal_spec> make the rule quite powerful. For example:

```
-- Not two instantiations of Gen with the same first parameter:
check instantations (Gen, =);

-- Not two instantiations of Gen with the same first and third parameter:
check instantiations (Gen, =, <>, =);

-- Not two instantiations of Gen with the same first parameter if the
-- second parameter is Pack.Proc:
check instantiations (Gen, =, Pack.Proc);

-- Not two instantiations of Gen with the same first parameter if the
-- second parameter is any procedure named Proc:
check instantiations (Gen, =, all Proc);
```

Note that a generic actual wich is a subtype matches all types (and subtypes) above it. Therefore,

```
check instantiations (ada.unchecked_deallocation (standard.natural));
```

will find only instantiations that use `Natural`, while:

```
check instantiations (ada.unchecked_deallocation (standard.integer));
```

will find instantiations that use either `Integer`, `Positive`, or `Natural`.

5.23.4 Limitation

GNAT defines `Unchecked_Conversion` and `Unchecked_Deallocation` as separate entities, rather than renamings of `Ada.Unchecked_Conversion` and `Ada.Unchecked_Deallocation`. As a consequence, it is necessary to specify explicitely both forms if you want to make sure that the corresponding generics are not instantiated.

5.24 Insufficient_Parameters

This rule controls calls to subprograms and entries where the values of parameters does not provide sufficient information to the reader to correctly identify the parameter's purpose.

5.24.1 Syntax

```
<control_kind> insufficient_parameters (<max_allowed> {, <entity>});
```

5.24.2 Action

<max_allowed> is the maximum number of allowed "insufficient" parameters (can be 0). The <entity> parameters designate enumeration types whose values should be included in the check. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

An actual parameter is deemed "insufficient" if it is given in positional (as opposed to named) notation, it is an expression whose primaries are all numeric literals, or enumeration literals belonging to one of the types passed as parameters to the rule (`Standard.Boolean` for example).

This rule can be given once for each of check, search, and count. This way, it is possible to have a level considered a warning (search), and one considered an error (check).

Ex:

```
search Insufficient_Parameters (1, Standard.Boolean);
check  Insufficient_Parameters (2, Standard.Boolean);
```

5.24.3 Tips

This rule does not apply to operators that use infix notation, nor to calls to subprograms that are attributes, since named notation is not allowed for these.

This rule controls the use of positional parameters according to their values; it is also possible to control the use of positional parameters according to the number of parameters with the rule `positional_associations`. See Section 5.42 [Positional_Associations], page 101.

Note also that this rules applies only to calls, while `positional_associations` applies to all forms of associations.

5.25 Local_Access

This rule controls the taking of access values (through the `'Access`, `'Unchecked_Access`, or the GNAT specific `'Unrestricted_Access` attributes) of local (i.e. non global) entities.

5.25.1 Syntax

```
<control_kind> local_access [(<subrule> {,<subrule>})];
<subrule>   ::= constant | variable | procedure | function |
                protected_procedure | protected_function
```

5.25.2 Action

Without parameters, the rule controls all entities given as prefixes of `'Access`, `'Unchecked_Access`, or `'Unrestricted_Access` attributes and reports on those that are not global, i.e. not defined in (possibly nested) library packages.

If parameters are specified, only entities belonging to the corresponding categories are controlled.

Ex:

```
Dangerous_Objects: check local_access (Constant, Variable);
```

5.25.3 Tips

In Ada 95, accessibility rules make sure that taking the `'Access` of an entity cannot create dangling pointers, but this check can be circumvented by using `'Unchecked_Access` (but not on subprograms), or in GNAT, by using `'Unrestricted_Access`. Moreover, Ada 2005 generalized anonymous access types create more cases where accessibility levels are dynamically checked.

Taking an access value on a global entity is never a risk, but every use of access values designating local entities has a potential of a failing dynamic accessibility check or even of a dangling pointer. This rule is helpful in finding the places that need careful inspection - of for disallowing taking accesses on anything but global entities.

5.26 Local_Hiding

This rule controls declarations that hide an outer declaration with the same name.

5.26.1 Syntax

```
<control_kind> local_hiding [(<subrule> {,"<allowed pattern>"})];
<subrule>   ::= {<exception>}  strict | overloading
<exception> ::= not_operator           | not_enumeration       |
                not_identical_renaming | not_different_families
```

5.26.2 Action

If "strict" is given (or if there is no subrule), the rule controls strict hiding (an inner subprogram that overloads an outer one is not considered hiding). If "overloading" is given, only subprograms that overload another subprogram in the same scope or in an outer scope are controlled. Note that following the normal Ada model, the declarations of enumeration literals are considered functions (and thus controlled).

Modifiers are used to exclude some controls (i.e. to allow the corresponding hiding):

- "not_operator": the subrule does not apply to the declarations of operators (i.e. things like "`"+"`").
- "not_enumeration": the subrule does not apply to the hiding/overloading of enumeration literals by other enumeration literals (the rule still applies to the hiding/overloading of functions by enumeration litterals, for example).
- "not_identical_renaming" (only allowed with "strict"): the subrule does not apply to renamings where the renaming name is the same as the name of the renamed entity. Such renamings are commonly used to provide visibility of identifiers in a controlled way.
- "not_different_families" (only allowed with "strict"): the subrule does not apply if the hiding identifier and the hidden one do not belong to the same "family". Families are either data (constant, variables, numbers, etc.), types, subprograms (including entries), packages, generics, exceptions, and labels (including block and loop names).

If one or more `<allowed pattern>` are given, hiding (or overloading) of identifiers that match one of the patterns are not reported. The whole syntax for regular expressions is allowed for the pattern, but the matching is always case insensitive. See Appendix B [Syntax of regular expressions], page 149.

This rule can be given only once for "strict" and once for "overloading".

Ex:

```
Hiding: check local_hiding (strict);
Overloading: search local_hiding (not_operator overloading);
```

5.26.3 Variables

The rule provides a variable that allows to adjust the verbosity of messages for the subrule "overloading".

Variable	Values	Default	Effect
Overloading_Report	compact/detailed	detailed	if "detailed", when a construct that overloads several other constructs is encountered, "overloading" will issue a message for each overloaded construct; if "compact", it will issue a single message mentionning how many constructs are overloaded, and a pointer to the last one.

5.26.4 Tips

If you have a naming convention like having all tagged types named "instance" (with a meaningful name for the enclosing package), and if in addition your package structure follows the inheritance hierarchy (i.e. a descendent class is in a child package), then all "instance" will hide each other - but this is of course intended. Specifying "`^instance$`" as an allowed pattern will prevent error messages for these declarations.

Note that the name is given between "`^`" and "`$`". Otherwise, following normal regexp syntax, any identifier *containing* "instance" would be allowed.

A confusion between names belonging to different "families" (as defined here) always leads to a compilation error; it may be acceptable to allow local hiding of names belonging to different families, since there is no risk involved.

5.27 Max_Blank_Lines

This rule controls excessive spacing in the program text.

5.27.1 Syntax

```
<control_kind> max_blank_lines (<max allowed blank lines>);
```

5.27.2 Action

This rule controls the occurrence of more than the indicated number of consecutive blank lines (empty lines, or lines that contain only spaces). This rule can be given once for each of check, search, and count. This way, it is possible to have a number of blank lines considered a warning (search), and one considered an error (check). Of course, this makes sense only if the number for search is less than the one for check.

Ex:

```
search max_blank_lines (2);
check max_blank_lines (5);
```

5.28 Max_Call_Depth

This rule controls the maximum depth of subprograms (or entry) calls.

5.28.1 Syntax

```
<control_kind> max_call_depth (<allowed depth> | finite);
```

5.28.2 Action

Roughly, the call depth is the number of frames that are stacked by a call: if you call a subprogram that calls another subprogram that calls nothing, then the call depth is 2. Note that a call to a task (not protected) entry has always a depth of 1, since the accept body that corresponds to the entry is executed on a different stack.

The value of the parameter is the maximum *allowed* depth, i.e. the rule will trigger if the call depth is strictly greater than the indicated value. A call to a (directly or indirectly) recursive procedure is considered of infinite depth, and will be therefore signaled (with an appropriate message) for any value of <allowed depth>. Alternatively, the keyword "finite" can be given in place of the <allowed depth>: in this case, only calls to recursive subprograms will be signalled.

This rule can be given once for each of check, search, and count. This way, it is possible to have a call depth considered a warning (search), and one considered an error (check). Of course, this makes sense only if the number for search is less than the one for check.

Ex:

```
search max_call_depth (9);
check  max_call_depth (finite);
```

5.28.3 Tip

It is possible to give the value 0 for <allowed depth>. Of course, it would not make sense to forbid all subprogram calls in an Ada program, but this can be useful for inspection purposes, since every call will be reported, and the message indicates the depth of the call.

If the message says that the call depth "is N", it is exactly N. If the message says that the call depth is "at least N", it means that the call chain includes a call to a subprogram whose depth is unknown (see "Limitations" below); "N" is the call depth if this subprogram does not call anything else. Of course, the rule issues a message if this minimal value is greater than the maximum allowed value.

5.28.4 Limitations

Calls to subprograms that are attributes are assumed to have a depth of 1. Calls to predefined operators are assumed to be in-lined (i.e. a depth of 0).

Calls through pointers to subprograms and dispatching calls are unknown statically; in addition, some subprograms may not have a body available for analysis, like imported subprograms, or possibly subprograms from the standard library; they are all assumed to have a depth of 1. Such calls are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.29 Max_Line_Length

This rule controls that no line exceeds a given length.

5.29.1 Syntax

```
<control_kind> max_line_length (<max allowed length>);
```

5.29.2 Action

This rule controls the maximum length of source lines. This rule can be given once for each of check, search, and count. This way, it is possible to have a length considered a warning (search), and one considered an error (check). Of course, this makes sense only if the length for search is less than the one for check.

Ex:

```
search max_line_length (80);
check max_line_length (120);
```

5.30 Max_Nesting

This rule controls excessive nesting of declarations.

5.30.1 Syntax

```
<control_kind> max_nesting ([<subrule>,] <max allowed depth>);
<subrule> ::= all | generic | separate | task
```

5.30.2 Action

If "all" (or no subrule) is given as the first parameter, this rule controls the nesting of declarative constructs (like subprograms, packages, generics, block statements...) that exceed a given depth. Nesting of statements (**loop**, **case**) is not considered.

If "generic" is given as the first parameter, this rule controls the nesting of generics, ignoring all non-generic units.

If "separate" is given as the first parameter, this rule controls the nesting of separate bodies.

If "task" is given as the first parameter, this rule controls the nesting of tasks (task types and single task objects), ignoring all non-task units.

This rule can be given once for each subrule and each of check, search, and count. This way, it is possible to have a level considered a warning (search), and one considered an error (check). Of course, this makes sense only if the level for search is less than the one for check.

Note that the value given is the maximum *allowed* nesting; f.e. if the value given for "generic" is 1, it means that a generic inside a generic is allowed, but not more.

Ex:

```
search max_nesting (5);
check max_nesting (all, 7);
check max_nesting (generic, 1);
check max_nesting (separate, 0); -- Do not allow separate in separate
check max_nesting (task, 0);     -- Do not allow a task in another task
```

5.31 Max_Size

This rule controls the maximum size, in source lines of code, of various statements and declarations.

5.31.1 Syntax

```
<control_kind> max_size (<subrule>, <max allowed lines>);
<subrule> ::= accept        | block          | case  | case_branch   |
              if            | if_branch      | loop  | simple_block  |
              unnamed_block | unnamed_loop   |
              package_spec  | package_body   | procedure_body |
              function_body | protected_spec | protected_body |
              entry_body    | task_spec      | task_body      |
              unit
```

5.31.2 Action

The first parameter is a subrule keyword that determines which elements are controlled:

- "accept" controls accept statements.
- "block" controls all block statements, while "simple_block" controls only blocks without a **declare** part, and "unnamed_block" controls only blocks without a name.
- "loop" controls all loop statement, while "unnamed_loop" controls only loops without a name.
- "if_branch" and "case_branch" control the length of each alternative of an **if** (respectively **case**) statement.
- "package_spec", "package_body", "procedure_body", "function_body", "protected_spec", "protected_body", "entry_body", "task_spec", and "task_body" control the length of the declaration of the corresponding element.
- "unit" controls the whole length of compilation units.

For each kind of element, the indicated value is the maximum allowed size of the full element; however, for branches ("if_branch" and "case_branch") it is the maximum size of the sequence of statements in the branch (i.e., the line that contains the **elsif** is not counted as part of an "if_branch").

This rule can be given once for each of check, search, and count for each kind of element. This way, it is possible to have a level considered a warning (search), and one considered an error (check). Of course, this makes sense only if the number of lines for search is less than the one for check.

Ex:

```
check Max_Size (if_branch, 30);
search Max_Size (if_branch, 50);
check Max_Size (unnamed_loop, 20);
```

5.31.3 Tip

Note that "procedure_body" and "function_body" apply to protected subprograms as well as regular ones, and that there is no subrule for the length of the declaration of subprograms. Such fine specifications didn't seem useful, but could be added if someone expresses a need for it.

5.32 Max_Statement_Nesting

This rule controls the nesting of compound statements.

5.32.1 Syntax

```
<control_kind> max_statement_nesting (<subrule>, <max allowed depth>);
<subrule> ::= block | case | if | loop | all
```

5.32.2 Action

If one of "block", "case", "if", or "loop" is specified, it controls the nesting of statements of the same kind, i.e. an **if** within a **loop** within an **if** counts only 2 for the "if" keyword. If "all" is specified, all kinds of compound statements are counted together, i.e. an **if** within a **loop** within an **if** counts for 3. This rule can be given once for each of check, search, and count, and for each of the subrules. This way, it is possible to have a level considered a warning (search), and one considered an error(check). Of course, this makes sense only if the level for search is less than the one for check.

Ex:

```
check max_statement_nesting (loop, 3);
search max_statement_nesting (all, 5);
```

5.33 Movable_Accept_Statements

This rule controls statements that are inside accept statements and could safely be moved outside.

5.33.1 Syntax

```
<control_kind> movable_accept_statements (certain|possible {, <entity>})
```

5.33.2 Action

Since it is good practice to block a client for the shortest time possible, any action that does not depend on the accept parameters should not be part of an accept statement.

Statements that involve synchronisation (delay statements, accept or entry calls...) are not movable. Statements (including compound statements) that reference the parameters of the enclosing accept are not movable. In addition, statements that use one of the <entity> given as parameters are never considered movable. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. Note that if a generic entity, or an entity declared in a generic package, is given, all statements that use the corresponding instantiated entity are considered not movable.

If the first parameter of the rule is `certain`, only statements after the last non-movable statement are reported. If the first parameter is `possible`, a simple data flow analysis is performed, and every statement that does not reference a variable that appears to depend (directly or indirectly) on a parameter is also reported.

Ex:

```
check movable_accept_statements (possible, Log.Report_Rendezvous);
```

5.33.3 Tips

The list of <entity> given to the rule can be, for example, procedures whose execution must be part of the accept statement for logical reasons. They can also be global variables, when the rendezvous is intended to prevent concurrent access to these variables.

5.34 Naming_Convention

This rule controls the form of identifiers to make sure that they follow the project's naming conventions. Different naming conventions can be specified, depending on the kind of Ada entity that the name is refering to.

5.34.1 Syntax

```
<control_kind> naming_convention
   ([root] [others] {<location>} [<type_spec>] <filter_kind>,
    [case_sensitive|case_insensitive] [not] "<pattern>"
    {, ...});
<location>  ::= global | local | unit
<type_spec> ::= <entity> | {<category>}
<category>  ::= ()      | access    | array | delta  | digits | mod |
                private | protected | range | record | tagged | task
<filter_kind> ::= All |
                  Type |
                     Discrete_Type |
                        Enumeration_Type |
                        Integer_Type |
                           Signed_Integer_Type |
                           Modular_Integer_Type |
                        Floating_Point_Type |
                        Fixed_Point_Type |
                           Binary_Fixed_Point_Type |
                           Decimal_Fixed_Point_Type |
                     Array_Type |
                     Record_Type |
                        Regular_Record_Type |
                        Tagged_Type |
                        Interface_Type |
                        Class_Type |
                     Access_Type |
                        Access_To_Regular_Type |
                        Access_To_Tagged_Type |
                        Access_To_Class_Type |
                        Access_To_SP_Type |
                        Access_To_Task_Type |
                        Access_To_Protected_Type |
                     Private_Type |
                        Private_Extension |
                     Generic_Formal_Type |
                  Variable |
                     Regular_Variable |
                     Field |
                        Discriminant |
                        Record_Field |
```

```
      Protected_Field |
   Procedure_Formal_Out |
   Procedure_Formal_In_Out |
   Generic_Formal_In_Out |
Constant |
   Regular_Constant |
      Regular_Static_Constant |
      Regular_Nonstatic_Constant |
   Named_Number |
      Integer_Number |
      Real_Number |
   Enumeration |
   Sp_Formal_In |
   Generic_Formal_In |
   Loop_Control |
   Occurrence_Name |
   Entry_Index |
Label |
Stmt_Name |
   Loop_Name |
   Block_Name |
Subprogram |
   Procedure |
      Regular_Procedure |
      Protected_Procedure |
      Generic_Formal_Procedure |
   Function |
      Regular_Function |
      Protected_Function |
      Generic_Formal_Function |
   Entry |
      Task_Entry |
      Protected_Entry |
Package |
   Regular_Package |
   Generic_Formal_Package |
Task |
   Task_Type |
   Task_Object |
Protected |
   Protected_Type |
   Protected_Object |
Exception |
Generic |
   Generic_Package |
   Generic_Sp |
      Generic_Procedure |
```

```
      Generic_Function |
Renaming |
   Object_Renaming |
   Exception_Renaming |
   Package_Renaming |
   Subprogram_Renaming |
      Procedure_Renaming |
      Function_Renaming |
   Generic_Renaming |
      Generic_Package_Renaming |
      Generic_Sp_Renaming |
         Generic_Procedure_Renaming |
         Generic_Function_Renaming
```

5.34.2 Action

The first parameter defines the kind of declaration to which the rule is applicable, and other parameters are strings, interpreted as regular expressions that define the patterns that must be matched (or not). See Appendix B [Syntax of regular expressions], page 149.

If one or more <location> keyword is specified, the pattern applies only to identifiers declared at the corresponding place. Otherwise, the pattern applies to all identifiers, irrespectively of where they are declared. The definition of locations is as follows:

- "unit": The identifier is the defining name of a compilation unit.
- "global": The identifier is declared in a package or a generic package, possibly nested in other packages or generic packages.
- "local": All other cases.

In the case of objects (corresponding to filters in the "variable" and "constant" families) and functions (in the "function" family), it is possible to be more specific, depending on the type of the object (or the return type of the function), as specified by the <type_spec> modifier. The <type_spec> modifier is either a single <entity> giving the type of the object or one or more <category> keywords. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. The meaning of <category> is:

- "()": The object is of an enumerated type.
- "access": The object is of an access type.
- "array": The object is of an array type.
- "delta": The object is of a fixed point type (it is not currently possible to distinguish ordinary fixed point types from decimal fixed point types).
- "digits": The object is of a floating point type.
- "mod": The object is of a modular type.
- "private": The object is of a private type (including private extensions).
- "protected": The object is of a protected type.
- "range": The object is of a signed integer type.
- "record": The object is of an (untagged) record type.

- "tagged": The object is of a tagged type (including type extensions).
- "task": The object is of a task type.

For a given layer of the hierarchy (i.e. "variable", "regular_variable"), only the most specific filter is applicable, i.e. "standard.boolean variable" will apply to all boolean variables, while plain "variable" will apply to other variables. See examples below.

If "case_sensitive" is specified, pattern matching considers casing. Otherwise ("case_insensitive"), casing is irrelevant. The default is "case_insensitive", and can be changed by setting the variable "Default_Case_Sensitivity", see below. Note that the rule checks the name only at the place where it is declared; casing might be different when the name is used later.

If a pattern is preceded by "not", then the pattern must *not* be matched (i.e. the rule reports when there is a match).

The rule will be activated if an identifier is declared that does not match any of the "positive" patterns (the ones without "not"), or if it matches any of the "negative" patterns (the ones with a "not"). If only negative patterns are given, it is implicitely assumed that all other identifiers are OK. In other words, accepted identifiers must have the form of (at least) one of the "positive" patterns (if any), but not the form of one of the "negative" patterns.

The filter kinds are organized hierarchically, as reflected in the syntax above. To be valid, the name must match the patterns specified for its own filter, and for all filters above it in the hierarchy. For example, a modular type declaration must follow the rules (if specified) for "all", "type","discrete_type", "integer_type" and "modular_integer_type". However, if a filter kind is preceded by "others", the rule will apply only if there is no applicable positive pattern deeper in the hierarchy; similarly, if a filter kind is preceded by "root", no rule above it in the hierarchy is considered (neither for itself nor its children). This is useful to make exceptions to a more general rule. For example:

```
-- All identifiers must have at least 3 characters:
check naming_convention (all, "...");
-- And start with an upper-case letter
-- (will not apply to types and access types, because of "others" and
--  other rules given below)
check naming_convention (others all, case_sensitive "^[A-Z]");

-- Exception to the rule for "all":
-- No minimum length for "for loop" identifiers, but must be
-- all uppercase
check naming_convention (root loop_control, case_sensitive "^[A-Z]+$");

-- Types must start with "t", then an upper-case letter:
-- (will not apply to access types, because of "others" and
--  other rule given below)
check naming_convention (others type, case_sensitive "^t[A-Z]");

-- Access types must start with "ta", then an upper-case letter:
check naming_convention (access_type, case_sensitive "^ta[A-Z]");
```

```
-- Boolean variables, and only these, must start with "Is_" or
-- "Has_":
check naming_convention (variable, not "^Is_", not "^Has_");
check naming_convention (standard.boolean variable, "^Is_", "^Has_");

-- Functions returning Wide_String must start with "Wide_", and
-- similarly for Wide_Wide_String, and no other:
check naming_convention (standard.wide_string function,
   "^Wide_",
   not "^Wide_Wide_");
check naming_convention (standard.wide_wide_string function,
   "^Wide_Wide_");
check naming_convention (function, not "^Wide_");
```

It is of course not necessary to specify all the filter kinds, nor to specify filters down to the deepest level; if you specify a rule for "type", it will be applied to all type declarations, whether there is a more specific rule or not.

Subtypes and derived types must follow the rule for their respective original (full) type. Incomplete type declarations are *not* checked, since their corresponding full declaration is (normally) checked. Private types (including of course the full declaration of a private type) follow the rule for private types, *not* the rules for their full type view (otherwise it would be privacy breaking).

Renamings are treated specially: if there is no explicit rule for a given renaming, the applicable rule is the one for the renamed entity.

Ex:

```
-- Predefined name is forbidden:
check naming_convention (all, not "Integer");

-- Types must either start or end with T
check naming_convention (type, case_sensitive "^T_",
                               case_sensitive "_T$");

-- "Upper_Initials" naming convention:
check naming_convention
   (all, case_sensitive "^[A-Z][a-z0-9]*(_[A-Z0-9][a-z0-9]*)*$");

-- All global variables must start with "G_"
check naming_convention (global variable, "G_");
```

5.34.3 Variables

The rule provides a variable that allows to specify the default casing.

Variable	Values	Default	Effect
Default_Case_Sensitivity	on/off	off	if "on", controls that do not explicitely specify case sensitivity are case sensitive.

5.34.4 Tips

The rule only checks the casing of identifiers at the place where they are declared. A useful companion rule is "style (casing_identifier, original)", which ensures that every use of the identifier will use the same casing as in the declaration. See Section 5.54 [Style], page 120. Similarly, in the case of a subprogram and its parameters, the check is not done on the body if there is an explicit specification (since specification and body have to match anyway).

The rule does *not* check the names of operators, since it would make little sense to have naming conventions for things whose name is imposed. If you want to prevent the definition of operators, refer to the rule "declarations" and its subrules "operator", "equality_operator", and"predefined_operator". See Section 5.10 [Declarations], page 50.

Remember that a Regexp matches if the pattern matches any part of the identifier. Use "^" and "$" to match the beginning (resp. end) of the name, or both.

A constant is considered static for the purpose of "Regular_Static_Constant" and "Regular_Nonstatic_Constant" if it is of a discrete type initialized by a static expression, or if it is an aggregate whose components all have static values. This is different from the official definition of "static" in the language, but corresponds to what most users would expect.

"class_type" is applicable to subtypes that designate a class-wide type. Similarly, "access_to_class_type" is applicable to access types whose designated type is class-wide.

If you don't want any special rule for renamings (not even the one that applies to the renamed entity), specify:

```
check naming_convention (renaming, "");
```

This imposes no constraint on renamings, but since it is specified explicitely, the implicit rule for the renamed entity won't apply.

The `rules` directory of Adacontrol contains two files named `no_standard_entity.aru` and `no_system_entity.aru`. These are files that contain a naming_convention rule that forbids the declaration of names declared in packages `Standard` and `System`, respectively. You can simply "source" these files from your own rule file (or copy the content) if you want to disallow these identifiers.

Like usual, naming_convention rule can be given multiple times, and can be disabled. However, consider the following:

```
Rule1 : check naming_convention (constant, "^c_");
Rule2 : check naming_convention (constant, "^const_");
```

The rule will trigger if a constant is declared that does not start with either "c_" or "const_". But here, we have two different rule labels. The message will refer to the first label encountered in the rule file; this is the label that must be mentionned in a disabling comment, unless you simply disable "naming_convention".

5.34.5 Limitations

This rule does not support wide characters outside the basic Latin-1 set.

5.35 No_Operator_Usage

This rule controls integer types that do not use any arithmetic operators, which indicates that they might be replaceable with other kinds of types.

5.35.1 Syntax

```
<control_kind> no_operator_usage [(<parameter> [,<parameter>])];
<parameter> ::= [<filter>] <observed>
<filter>    ::= not | ignore | report
<observed>  ::= relational | logical | indexing
```

5.35.2 Action

This rule controls integer types (both signed and modular) where no arithmetic operator of the type is used in the program.

When such a type is found, it migh be interesting to find out other usages to determine a possible better kind of type. "relational" means that relational operators (<, <=, >, >=, **in**, **not in**) are used, "logical" means that logical operators (**and**, **or**, **xor**) are used, and "indexing" means that the type is used as an index in some array type.

If an <observed> property is given as parameter, only types that feature the property are controlled, or those that do *not* feature the property if the <observed> is preceded by "not". If the <observed> is preceded by "ignore" the type is controlled irrrespectively of the property, and the message does not mention it at all, while if it is preceded by "report", the message still mentions whether the <observed> is used or not.

Without parameters, the rule is equivalent to "ignore relational, ignore logical, ignore indexing" (i.e. it controls all types that do not use any arithmetic operator).

This rule can be given only once for each combination of values of the parameters.

Ex:

```
-- Simply report types that don't use arithmetic operators:
check no_operator_usage;

-- Do the same, but mention if indexing/logical ops are used:
check no_operator_usage (report indexing, report logical);

-- Find integer types that use only logical operators:
check no_operator_usage (logical);

-- Find integer types that don't use artihmetic operators and are
-- not used for indexing nor in relational operators:
check no_operator_usage (not indexing, not relational);
```

5.35.3 Tips

An integer type that uses no operator at all is a good candidate to be replaced by an enumerated type. A modular type where only logical operators are used is likely to be used as a bit field or a set, and is a good canditate for being replaced by an array of booleans.

The rule does not make a distinction between predefined and user-defined operators. On the other hand, only calls to operators are considered, operators used for example as actual generic parameters in instantiations are not considered.

The rule applies also to private types whose full declaration is an integer type.

5.36 Non_Static

This rule controls that expressions used in certain contexts are static.

5.36.1 Syntax

```
<control_kind> non_static [(<subrule> {, <subrule>})];
<subrule> ::= constant_initialization | variable_initialization |
              index_constraint        | discriminant_constraint |
              instantiation           | index_check
```

5.36.2 Action

The <subrule> defines the elements that are required to be static:

- "constant_initialization": expressions used as initial value in constant declarations.
- "variable_initialization": expressions used as initial value in variable declarations.
- "index_constraint": expressions used in index constraints (aka array sizes).
- "discriminant_constraint": expressions used in discriminant constraints
- "instantiation": expressions used as generic actual parameters in instantiations.
- "index_check": expressions used as indices must satisfy statically the index check. I.e., the expression needs not be static, but it should be statically provable that the index check cannot fail.

If no keyword is given, all contexts are controlled.

Ex:

```
check non_static (index_constraint);
```

5.36.3 Limitations

Currently, "constant_initialization" and "variable_initialization" do not control structured (record and array) variables. For access variables, the initial value is considered static only if it is a plain **null**. This may improve in future versions of AdaControl.

5.36.4 Tips

If all index and discriminant constraints are static, the space occupied by data structures is computable from the program text. This rule is useful to enforce this in contexts where the memory space must be statically determined.

5.37 Not_Elaboration_Calls

This rule controls that certain subprograms (or allocators) are called only during program initialization.

5.37.1 Syntax

```
<control_kind> not_elaboration_calls (<entity>|new {, <entity>|new});
```

5.37.2 Action

The <entity> parameters are callable entities (procedure, function or entry calls). As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. This rule controls calls to the indicated callable entities, or allocators if "new" is given, that are performed at any time except during the elaboration of library packages.

Ex:

```
search not_elaboration_calls (Data.Initialize, new);
```

5.37.3 Limitations

Due to an (allowed by ASIS standard) limitation of ASIS-for-Gnat, the rule will not detect calls to subprograms that are implicitely defined, like calling a `"+"` on `Integer`. Fortunately, it is very unlikely that the user would want to forbid that kind of calls in non-elaboration code.

Note also that calls that cannot be statically determined, like calls to dispatching operations or calls through pointers to subprograms cannot be detected either.

5.38 Not_Selected_Name

This rule controls that certain entities are always refered to using selected notation, even in the presence of **use** clauses.

5.38.1 Syntax

```
<control_kind> not_selected_name
   (<exception places>, <entity> {, <entity>});
<exception places> ::= none | unit | compilation | family
```

5.38.2 Action

A name is "selected" if it is prefixed by the name of the construct where it is declared. Only one level of prefix is required, unless the prefix itself is the target of a not_selected_name rule.

The first parameter specifies places where the rule is *not* enforced, i.e. where simple notation is allowed:

- "none": selected notation is always required.
- "unit": selected notation is not required within the program unit where the entity is declared.
- "compilation": selected notation is not required within the compilation unit where the entity is declared.
- "family": selected notation is not required within the compilation unit where the entity is declared, nor within its (direct or indirect) children.

Other parameters indicate the <entity> to which the rule applies. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

Ex:

```
check not_selected_name (unit, all Instance);
search not_selected_name (none, Pack.T);
```

5.38.3 Tip

Note that, as usual, the entity can be given in the form "all name". This is especially useful for types that must always be declared with a special name (like `Instance`, `Object`, `T`) and are intended to be always used with the name of the enclosing package.

5.39 Object_Declarations

This rule controls various aspects of object (constants and variables) declarations.

5.39.1 Syntax

```
<control_kind> object_declarations (min_integer_span, <min_spec>
                                                    {, <min_spec>});
<control_kind> object_declarations (type, <type_spec> {, <type_spec>});
<control_kind> object_declarations (volatile_no_address);
<control_kind> object_declarations (address_not_volatile);
<min_spec>  ::= [constant | variable] <value>
<type_spec> ::= [constant | variable] <entity>
```

5.39.2 Action

The action depends on the subrule.

- "min_integer_span": controls that every object of an integer type has a subtype that covers at least the indicated number of values. Different values can be specified for variables and constants; if no modifier ("constant" or "variable") is supplied, the value applies to both.

  This subrule can be given only once for each combination of check/search/count and constant/variable.

- "type": controls every object whose (sub)type matches <entity>. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145. If the <entity> is a subtype, only objects of that exact subtype are controlled; if the <entity> is a type, objects declared with the type or any subtype of it are controlled. The control can be restricted to only variables or only constants; if no modifier ("constant" or "variable") is supplied, both are controlled.

  This subrule can be given only once for each combination of <entity> and constant/variable.

- "volatile_no_address": controls variables that are the target of a pragma volatile, but have no address clause. Constants are not controlled, since it would be very strange to have a volatile constant...

  Since this subrule has no parameters, it can be given only once.

- "address_not_volatile": controls variables that have an address clause, but are not the target of a pragma volatile. Constants are not controlled, since it would be very strange to have a volatile constant...

  Since this subrule has no parameters, it can be given only once.

Ex:

```
check object_declarations (min_integer_span, variable 5, constant 10);

count object_declarations (min_integer_span, 8);
-- Same value for variables and constants

search object_declarations (volatile_no_address);
search object_declarations (address_not_volatile);
```

5.39.3 Tip

The "min_integer_span" rule can be useful for detecting variables that should use an enumerated type rather than an integer type.

5.39.4 Limitation

Due to a shortcomming of the ASIS interface, the subrules "volatile_no_address" and "address_not_volatile" will not detect variables of a class-wide type that are volatile due to a pragma volatile applying to the class-wide type. If the pragma applies to the variable, the subrule will work correctly. A pragma volatile applied to a class-wide type is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

Declaring a class-wide *type* as volatile seems very peculiar anyway...

5.40 Parameter_Aliasing

This rule controls aliased use of variables in subprogram calls.

5.40.1 Syntax

```
<control_kind> parameter_aliasing [([with_in] <level>)];
<level> ::= Certain | Possible | Unlikely
```

5.40.2 Action

This rule identifies calls where the same variable is given as an actual to more than one **out** or **in out** parameter, like in the following example:

```
procedure Proc (X, Y : out Integer);
   ...
Proc (X => V, Y => V);
```

If the modifier "`with_in`" is given, aliasing between **out** or **in out** parameters and **in** parameters is also considered (unless the **in** parameter is of a user-defined by-copy type). Although aliasing of **in** parameters is generally considered less of an issue, it can lead to unexpected results when the parameter is passed by reference.

There are many cases where aliasing cannot be determined statically. The optional parameter specifies how aggressively the rule will check for possible aliasings. Possible values are (case irrelevant):

- Certain (default): Only cases where aliasing is statically certain are output.
- Possible: In addition, cases where aliasing may occur depending on the value of an indexed component are output. These may or may not be true aliasing, depending on the algorithm. For example, given:

```
Swap (Tab (I), Tab (J));
```

there is no aliasing, unless `I` equals `J`.

If all expressions used for indexing in both variables are static, the rule will be able to eliminate the diagnosis of aliasing (if the values are different). This avoids unnecessary messages in cases like:

```
Swap (Tab (1), Tab (2));
```

- Unlikely: In addition, cases where aliasing may occur due to access variables pointing to the same variable are output. These may or may not be true aliasing, depending on the algorithm, but should normally occur only as the result of very strange practices, like in the following example:

```
type R is
   record
      X : aliased Integer;
   end record;
X : R;
Y : Access_All_Integer := R.X'access;
   ...
P (X, Y.all);
```

There will be no false positive with "Certain". There will be no false negative with "Unlikely" (but many false positives). "Possible" is somewhere in-between.

The rule may be specified at most once for each value of the parameter. This allows for example to "check" for "Certain" and "search" for "Possible".

Ex:

```
check parameter_aliasing (with_in certain);
search parameter_aliasing (Possible);
```

Note that the rule is quite clever: it will consider partial aliasing (like a record variable as one parameter, and one of its components as another parameter), and will not be fooled by renamings.

5.40.3 Limitation

Due to a weakness of the ASIS standard, dispatching calls are not analyzed. Some calls cannot obviously have aliasing (if there is only one parameter, or if there are no variables in the parameters f.e.); other calls are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.41 Parameter_Declarations

This rule controls various characteristics of the declaration of parameters for all callable entities (i.e. functions, procedures and entries).

5.41.1 Syntax

```
<control_kind> parameter_declarations (<subrule> [,<bounds>] {,<callable>});
<subrule>  ::= all_parameters       | in_parameters         |
               defaulted_parameters | out_parameters        |
               in_out_parameters    | access_parameters     |
```

```
                tagged_parameters    | class_wide_parameters |
                single_out_parameter
<bounds>    ::= min|max <value> [, min|max <value> ]
<callable> ::= function              | procedure              |
                dispatching_function | dispatching_procedure |
                protected_function   | protected_procedure   |
                protected_entry      | task_entry
```

5.41.2 Action

The first parameter is a subrule keyword. "single_out_parameter" has no parameter; all other subrules require one or two bounds.

- "all_parameters": Controls callable entities whose number of parameters is less than the given "min" or greater than the given "max". "min" defaults to 0 and "max" to infinity.
- "in_parameters", "out_parameters", "in_out_parameters": Do the same, counting only parameters of modes **in**, **out**, or **in out** respectively.
- "defaulted_parameters": Does the same, counting only parameters declared with an explicit default expression.
- "access_parameters": Does the same, counting only (anonymous) access parameters.
- "tagged_parameters": Does the same, counting only parameters of a specific tagged type.
- "class_wide_parameters": Does the same, counting only parameters of a class-wide type.
- "single_out_parameter": Controls callable entities that have exactly one **out** parameter. Procedures with a single **out** parameter might be candidates to becoming functions.

If one or more <callable_kind> is specified after the <value>, the rule applies only to the corresponding declaration(s), otherwise it applies to all callable entities. "dispatching_function" and "dispatching_procedure" allow different counts for dispatching subprograms (i.e. primitive subprograms of a tagged type). If "dispatching_function" or "dispatching_procedure" is not explicitely specified, "function" (conversely "procedure") applies also to dispatching functions (conversely dispatching procedures).

This rule can be given once for each of check, search, and count for each subrule and each kind of entity. This way, it is possible to have a level considered a warning (search), and one considered an error (check).

Ex:

```
-- Callable entities should preferably not have more than 5
-- parameters, and in any case not have more that 10 parameters,
check  parameter_declarations (all_parameters, max 10);
search parameter_declarations (all_parameters, max 5);

-- All functions must have parameters and no out or in out
-- parameters (allowed in Ada 2012):
check parameter_declarations (all_parameters,    min 1, function);
check parameter_declarations (out_parameters,    max 0, function);
```

```
check parameter_declarations (in_out_parameters, max 0, function);

-- A regular (not protected) procedure with one out parameter
-- should be replaced by a function
check parameter_declarations (single_out_parameter, procedure);

-- Find all callable entities with class-wide parameters:
search parameter_declarations (class_wide_parameters, max 0);

-- Dispatching operations may have only one parameter of a tagged type:
check parameter_declarations (tagged_parameter,
                              max 1,
                              dispatching_function,
                              dispatching_procedure);
```

5.41.3 Tips

This rule applies to generic subprograms as well as to regular ones. On the other hand, it does not apply to generic formal subprograms, since instantiations would only be possible with subprograms which are supposed to have been already controlled.

Instantiations are also controlled; the number of parameters is taken from the corresponding generic.

Note that this rule controls only "regular" parameters, not generic formal parameters.

Note that dispatching operations have necessarily at least one tagged parameter, although a "max 0" could be specified in the example above. If you do this, all declarations of dispatching subprograms will be controlled. Maybe that's what you want...

5.42 Positional_Associations

This rule controls the use of positional associations (as opposed to named associations) in all kinds of associations.

5.42.1 Syntax

```
<control_kind> positional_associations [(<subrule>, <max_allowed>
                                         [, <category> {, <entity>}])];
<subrule>  ::= all | all_positional | same_type
<category> ::=  [not_operator] call | pragma          | discriminant     |
                instantiation       | array_aggregate | record_aggregate |
                enumeration_representation
```

5.42.2 Action

The rule controls pragmas, discriminants, calls, aggregates, or instantiations that use too many positional associations. The definition of "too many" depends on the subrule:

- "all": when positional associations are given in a place where there is more than <max_allowed> associations (both positional and named).
- "all_positional": when there is more than <max_allowed> positional associations.

- "same_type": when more than <max_allowed> positional parameters are of the same type.

In addition, a <category> can be specified to restrict the rule to specific kinds of associations; if not specified, all associations are controlled. The categories carry their obvious meaning, with the distinction that "array_aggregate" applies only to "true" array aggregates, while "enumeration_representation" applies to the special array aggregate used in enumeration representation clauses. Note that the "same_type" subrule is not allowed for the "pragma" category. For "pragma", "call", and "instantiation", entities can also be specified; such entities are exempted from the rule (i.e. the rule will not control these entities). See examples below.

For calls, positional association is *not* reported for operators that use infix notation (since named notation is not possible); in addition, if the "not_operator" modifier is specified before the "call" keyword (not allowed elsewhere), positional association is never reported for operators, even if they are called with the syntax of a normal function call (i.e. `Pack."+"(A,B)`). Calls to subprograms that are attributes are not reported either, since named notation is not allowed for them.

This rule can be specified once for each combination of <subrule>, <category>, and <control_kind>. This way, it is possible to have a number of positional associations considered a warning (search), and one considered an error (check). Of course, this makes sense only if <max_allowed> for search is greater than the one for check. It is also possible to have different criteria for each category.

If no parameter is given, it is equivalent to "`positional_associations (all, 0)`", i.e. all positional associations are controlled.

Ex:

```
 -- All positional associations:
check positional_associations;

-- All positional associations in aggregates:
check positional_associations(all, 0, array_aggregate);
check positional_associations(all, 0, record_aggregate);

 -- All positional associations with more than 3 elements:
search positional_associations (all, 3);

-- Positional associations in calls with more than 3 params of the same type
search positional_associations (same_type, 3, call);

-- Positional associations in calls with more than 2 elements (except
-- calls to any subprogram called Put)
search positional_associations(all, 2, call, all put);
```

5.42.3 Tips

There are two kinds of calls where the rule does not complain about usage of positional association: infix operator calls (since requiring named notation would not allow infix nota-

tion any more), and calls to subprograms that are attributes (since named notation is not allowed for these).

For the purpose of the "same_type" subrule, integer literals are considered of the same type as any parameter of an integer type, and similarly for other universal values. The reason is that this rule is intended to avoid confusion between parameters, when strong typing would not detect an inversion of parameters for example; such a case would happen between parameters of a universal type.

For calls, another rule controls positional associations according to the value of parameters rather than their number: See Section 5.24 [Insufficient_Parameters], page 80.

5.43 Potentially_Blocking_Operations

This rule controls usage of potentially blocking operations (as defined in LRM 9.5.1 (8..16)) from within protected operations.

5.43.1 Syntax

```
<control_kind> potentially_blocking_operations;
```

5.43.2 Action

The rule follows the call graph, starting from every protected operation, and identifies all (direct and indirect) potentially blocking operations encountered. All protected types in the program are controlled.

Of course, calls to standard subprograms (notably IOs) that are defined to be potentially blocking are recognized.

Ex:

```
check potentially_blocking_operation;
```

5.43.3 Tips

This rule is very clever at finding potentially blocking operations resulting from external calls (or requeues) to the current protected object, even if this happens through a long chain of subprogram calls. Typically, this happens when a protected operation calls a subprogram, which in turn makes a call to an operation of the same protected object. Such calls generally result in dead-locks.

Therefore, it is advisable to run this rule on any program that exhibits mysterious (and hard to find) deadlocks that seem to involve protected objects.

When a single protected object is being analyzed, the rule will diagnose a circularity if there is a call to an operation of the same object in the call chain; however, if a protected type is being analyzed, the rule will diagnose a circularity if there is a call to any object of the same type in the call chain. Although it is possible to construct examples of this latter case where there is no risk of deadlock, it is so contrieved that it certainly deserves being looked at. But since the call is not 100% certain to be potentially blocking, the message will tell "possible external call" instead of "external call" in this case.

5.43.4 Limitation

There is one case defined in LRM E.4(17) which is not recognized: remote subprograms calls.

Calls through pointers to subprograms and dispatching calls are unknown statically; they are assumed to be non potentially blocking. Such calls are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.44 Pragmas

This rule controls usage of one or several specific pragmas.

5.44.1 Syntax

```
<control_kind> pragmas (<pragma spec> {, <pragma spec>});
<pragma spec> ::= [multiple] all|nonstandard|<pragma name>
```

5.44.2 Action

If the special name "nonstandard" is given, then all implementation-defined and unrecognized pragmas will be controlled. If the special name "all" is given, then all pragmas will be controlled. Otherwise, the parameters are the names of pragmas to be controlled. Note that <pragma name> must be the simple name of the pragma, since pragma names are predefined and do not follow the rules for regular Ada entities.

If "multiple" is specified before the pragma spec (or the special name), the corresponding pragma(s) are controlled only if they apply to multiple entities, because one of the parameters is an overloaded name.

Ex:

```
check pragmas (elaborate_all, elaborate_body);

-- Search pragma Convention that apply to several entities:
search pragmas (multiple convention);
```

5.44.3 Tips

If "all" and/or "nonstandard" is given together with a specific pragma name in a "search" or "check" rule, a message is issued only for the most specific occurrence. However, for "count", all appropriate occurrences are counted, i.e. given the following rules:

```
C1 : count pragmas (annotate);
C2 : count pragmas (nonstandard);
C3 : count pragmas (all);
```

Counter C1 will report the number of occurrences of **pragma** `Annotate` (a non-standard GNAT pragma), counter C2 will report the number of non-standard pragmas (including occurrences of `Annotate`), and counter C3 will report the total number of pragmas (including occurrences of `Annotate`).

5.45 Record_Declarations

This rule controls various aspects of the components of records.

5.45.1 Syntax

```
<control_kind> record_declarations (component, <compo_kind> {,<repr_cond>});
<compo_kind> ::= <entity>|<category>
```

```
<category>   ::= () | access    | array | delta  | digits | mod | private |
                     protected | range | record | tagged | task
<repr_cond>  ::= [not] in_variant | aligned | initialized | packed | sized
```

5.45.2 Action

The first parameter is a subrule keyword:

- "Component" controls record components whose type is the indicated <entity>, or whose type belongs to the indicated <category>. If the <entity> is a subtype, only record components that are of that subtype are controlled. If the indicated <entity> is a type, all record components that are of that type (including subtypes) are controlled. The meaning of <category> is:
  - "()": The component is of an enumerated type.
  - "access": The component is of an access type.
  - "array": The component is of an array type.
  - "delta": The component is of a fixed point type (it is not currently possible to distinguish ordinary fixed point types from decimal fixed point types).
  - "digits": The component is of a floating point type.
  - "mod": The component is of a modular type.
  - "private": The component is of a private type (including private extensions).
  - "protected": The component is of a protected type.
  - "range": The component is of a signed integer type.
  - "record": The component is of an (untagged) record type.
  - "tagged": The component is of a tagged type (including type extensions).
  - "task": The component is of a task type.

  If <repr_cond> are specified, the rule controls only record components to which all the corresponding representation items apply:
  - "in_variant": The component appears inside the variant part of the record.
  - "not in_variant": The component appears inside the fixed part of the record.
  - "aligned": Either no component clause applies to the component, or the corresponding first bit is a multiple of `Storage_Unit`.
  - "not aligned": A component clause applies to the component, and the corresponding first bit is not a multiple of `Storage_Unit`.
  - "initialized": The component has a default initialization expression.
  - "not initialized": The component has no default initialization expression.
  - "packed": A pragma Pack applies to the component type.
  - "not packed": No pragma Pack applies to the component type.
  - "sized": A component clause applies to the component (therefore imposing the size).
  - "not sized": No component clause applies to the component.

  This rule can be specified several times for the "component" subrule.

  Ex:

```
-- All record components of a discrete type should be initialized:
check record_declarations (component, (), not initialized);

-- The size of all components of type Hardware_Types.Squeezed must
-- have a component clause:
check record_declarations (component, Hardware_Types.Squeezed, not sized);

-- Find unaligned components of a packed array type:
check record_declarations (component, array, packed, not aligned);
```

5.45.3 Tips

It may seem strange to have a rule with only one subrule, but we expect to add more in the near future. Stay tuned...

5.45.4 Limitations

If "[not] aligned" is specified, there are some rare cases where AdaControl cannot evaluate whether a component is aligned or not; in this case, it will "assume the worse" (i.e. report as if the component had the specified alignment), thus creating possible false positives. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.46 Reduceable_Scope

This rule controls declarations that could be moved to some inner scope.

5.46.1 Syntax

```
<control_kind> reduceable_scope [(<subrule> {, <subrule>})];
<subrule> ::= {<restriction>} all        | variable | constant |
                              subprogram | type     | package  |
                              exception  | generic  | use
<restriction> ::= no_blocks | to_body
```

5.46.2 Action

The rule reports on any declaration that is referenced only from a single, inner scope, or in the case of **use** clauses, it will report on packages named in a **use** clause whose elements are used only in a single, inner scope. For entitities declared in package specifications, the rule reports if they are used only from the corresponding package body.

The initialization of an object is considered a usage of the object at the place where it is declared, thus preventing it from being moved. Therefore, constants and initialized variables are never reported as being movable to inner scopes; they are reported as being movable to package bodies however. Entities that are used as prefixes of a 'Access or 'Address attribute are never reported, since moving them would change their accessibility level. Similarly, task objects are not reported since moving them would change their master. Finally, dispatching operations (primitive operations of tagged types) are not reported either, since they can be the target of an "invisible" (dispatching) call.

If no <subrule> is given, or the <subrule> is "all", all declarations are controlled. If `no_blocks` is specified in front of a <subrule>, the rule will not consider blocks as possible

targets for a reduced scope for the corresponding category. If `to_body` is specified in front of a <subrule>, the rule will report only elements declared in a package specification that could be moved into the body. Specifying "all" explicitely is only useful in the case where there is a <restriction>.

As a side effect, the rule will report about entities that are declared but not used (i.e. whose scope reduces to nothing).

Ex:

```
-- Types and variables shall be declared in the innermost scope
-- where they are useful:
check reduceable_scope (variable, type);

-- Packages and subprograms shall be declared in the innermost
-- scope where they are useful, but they are not allowed in blocks:
check reduceable_scope (no_blocks subprogram, no_blocks package);

-- Use clause should be as restricted as possible:
search reduceable_scope (use);
```

5.46.3 Tips

If you think that **use** clauses are acceptable, but should be limited to the smallest possible scope, you would generally specify:

```
check unnecessary_use_clause;
check reduceable_scope (use);
```

5.46.4 Limitation

Currently, the rule does not report **use** clauses declared in a package specification that could be moved to the body. Such clauses appear as "unused" (but of course, the compiler will complain on the body if the clause is removed).

5.47 Representation_Clauses

This rule controls usage of representation clause.

5.47.1 Syntax

```
<control_kind> representation_clauses [(<subrule> {, <subrule>})];
<subrule> ::= {<category>} <repr_kw> | [global] [object] <attribute>
<repr_kw> ::=
   at                    | at_mod                | enumeration         |
   fractional_size       | incomplete_layout     | layout              |
   non_aligned_component | non_contiguous_layout | no_bit_order_layout |
   overlay
<category> ::=
   ()     | range     | mod    | delta | digits  | array | record   |
   tagged | extension | access | new   | private | task  | protected
```

5.47.2 Action

Without parameter, the rule controls all representation clauses, otherwise it will control the representation clauses given as parameter.

If a representation keyword or attribute is preceded by one or several categories, the rule controls only the representation items that apply to types belonging to the categories (the type of the component for the `non_aligned_component` subrule):

- "()": Enumerated types
- "range": Signed integer types
- "mod": Modular types
- "delta": Fixed point types (no possibility to differentiate ordinary and decimal fixed point types yet).
- "digits": Floating point types
- "array": Array types
- "record": (untagged) record types
- "tagged": Root tagged types
- "extension": Type extensions (tagged derived types)
- "access": Access types
- "new": Derived types
- "private": Private types
- "task": Task types
- "protected": Protected types

The meaning of the representation keywords is:

- "at" controls address clauses given in Ada 83 style ("for XXX use at").
- "at_mod" controls alignment clauses given in Ada 83 style ("for T use record at mod XX;").
- "enumeration" controls enumeration representation clauses.
- "fractional_size" controls size clauses whose value is not an integral multiple of `System.Storage_Unit`.
- "incomplete_layout" controls record representation clauses that miss the specification of some components of the record's type.
- "layout" controls all record representation clauses, while "no_bit_order_layout" controls record representation clauses whose type is not also the target of a bit_order attribute specification (such types have a non-portable representation).
- "non_aligned_component" controls components that do not start on a storage unit boundary. The message gives the offset (in bits) relative to the closest storage unit boundary.
- "non_contiguous_layout" controls record representation clauses where there are unused bits between components (or before the first component). A message is issued for each "gap" between components. In addition, if a size clause is given for the type, the rule will report if there are unused bits at the end of the component (i.e. the size clause is bigger than the end of the last component). In the case of variant records, there can be overlapping fields; the rule will control only the bits that belong to no variant at all.

- "overlay" controls address clauses (given in either style), where the value given is the `'Address` of some other element.

In addition to these keyword, any specifiable attribute can be given (including the initial "`'`"); the rule will control specifications of this attribute. If the modifier "global" is given before the attribute, only attribute specifications for global entities are controlled. If the modifier "object" is given before the attribute, only attribute specifications for objects are controlled (as opposed to types for example). Note that double attributes (like "`'CLASS'INPUT`") can be given, and are considered different from the simple attribute ("`'INPUT`"). It is of course possible to specify both.

Ex:

```
All_Addresses: check representation_clauses (at, 'address);
All_Input: check representation_clauses ('input, 'class'input);
Sized_Objects: check representation_clauses (object 'size);
count representation_clauses ('SIZE);

-- check layout clauses for derived types:
check representation_clauses (new layout);

-- check layout clauses for root tagged types and type extensions:
check representation_clauses (tagged extension layout);
```

5.47.3 Limitation

For the "fractional_size" and "non_contiguous_layout" subrules, there are some rare cases where AdaControl cannot evaluate the given size or elements of the record representation clause, and thus not detect the corresponding situation. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.47.4 Tips

The specifiable attributes (the ones that can be given as parameters to this rule) are `'Address`, `'Size`, `'Component_Size`, `'Alignment`, `'External_Tag`, `'Small`, `'Bit_Order`, `'Storage_Pool`, `'Storage_Size`, `'Write`, `'Output`, `'Read`, `'Input`, and `'Machine_Radix`. See Ada Reference Manual 13.3(77).

Ada allows partial record representation clauses, i.e. it does not require all fields to be specified. This means that if you add a field to a record and forget to update the associated representation clause, there will be no compilation error. The "incomplete_record" subrule is handy for making sure that this does not happen.

Derived types with a representation clause may suffer an efficiency penalty, since calling an inherited subrograms requires a change of representation. Representation clauses for tagged types are dubious, since these types have hidden fields added by the compiler.

5.48 Return_Type

This rule controls that certain form of types are not used for function results.

5.48.1 Syntax

```
<control_kind> return_type [(<subrule> {, <subrule>})];
```

```
<subrule> ::= class_wide                   | limited_class_wide  |
              constrained_array            | protected           |
              task                         | unconstrained_array |
              unconstrained_discriminated | anonymous_access
```

5.48.2 Action

This rule controls functions whose return type belongs to one of the indicated type kinds:

- `class_wide` controls all class-wide types, while `limited_class_wide` controls only limited class-wide types.
- `constrained_array` controls constrained array types
- `unconstrained_discriminated` controls types with discriminants (but not constrained subtypes of such types)
- `unconstrained_array` controls unconstrained array types
- `task` controls task types, or composite types that include tasks as subcomponents.
- `protected` controls protected types, or composite types that include protected objects as subcomponents.
- `anonymous_access` controls anonymous access types.

If no subrule is specified, all type kinds are controlled. Note that more than one kind may apply to a type: for example, a function can return a class-wide type with discriminants that includes tasks and protected objects as subcomponents. In this case, several messages are issued for the same type.

Ex:

```
check return_type (unconstrained_discriminated, unconstrained_array);
```

5.49 Side_Effect_Parameters

This rule controls calls that may depend on the order of evaluation of parameters.

5.49.1 Syntax

```
<control_kind> side_effect_parameters (<entity> {, <entity>});
```

5.49.2 Action

This rule controls subprogram calls or generic instantiations where different actual parameters call functions known to have side effects. This is dangerous practice, since correct behaviour may depend on a certain evaluation order of parameters, which is not specified by the language.

All <entity> are functions that are assumed to interfere, i.e. the rule will signal if any of these functions is called more than once in the parameters of a call. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

It is allowed to give the name of a generic function, or of a function declared in a generic package; in this case, all functions resulting from instantiations of these generics will be considered.

In the case of renamings, you must give the name of the original function; the rule will work correctly if the call is made through a renaming of this function.

Ex:

```
check side_effect_parameters (F1);
check side_effect_parameters (G1, G2);
```

Here, F1 has a side effect, and the rule will signal if it is called more than once. G1 and G2 are assumed to interfere, and therefore the rule will signal if either is called more than once, or if both are called. However, having a call that mentions F1 and G2 is OK.

5.49.3 Limitation

Due to the size of internal structures, this rule may not be given more than 100 times.

Due to an unimplemented feature of ASIS-for-Gnat, this rule will not process defaulted parameters, and hence not detect interferences due to calling a side-effect function through the default value.

5.50 Silent_Exceptions

This rule controls exception handlers that can cause exceptions to silently disappear.

5.50.1 Syntax

```
<control_kind> silent_exceptions (<element> {, <element>});
element      ::= <control-item> | <report-item>
control-item ::= not | with   <entity> | others
report-item  ::= raise   | explicit_raise | reraise | return |
                 requeue | <entity>
```

5.50.2 Action

The rule controls handlers that do *not* call one of the given subprograms (for example a reporting procedure) nor perform other required operations, like returning, requeuing, or re-raising an exception.

A parameter that starts with "not" or "with" is a <control-item> and defines wich exceptions are controlled; the <entity> should be either an exception, or the name of a library unit (in which case, it applies to all exceptions declared in the library unit). As usual, the whole syntax for entities is allowed here. See Appendix A [Specifying an Ada entity name], page 145. If the <entity> is (part of) a generic, then it applies to all exceptions from all corresponding instantiations. If there is no <control-item>, then all exceptions are controlled.

If several <control-item> are given, the ones with "with" add exceptions to the set of controlled exceptions, and the ones with "not" remove exceptions, in order, starting from the empty set if the first <control-item> is a "with", or starting from the set of all exceptions if the first <control-item> is a "not". See examples below.

"**when others**" handlers are always controlled, unless there is an explicit "not others" <control-item>. A "with others" <control-item> can be specified to check *only* "**when others**" handlers.

The other parameters are <report-item> and define the constructs considered "reporting". <entity> should correspond to an Ada callable entity or generic package; as usual,

the whole syntax for entities is allowed here. See Appendix A [Specifying an Ada entity name], page 145. If a generic procedure or function is given, then all corresponding instances are considered reporting subprograms. If a generic package is given, any instantiation (in an inner block of the handler) is considered reporting. In addition, the special names "explicit_raise", "reraise", "return" and "requeue" mark raise statements with an explicit exception name, raise statements without an exception name, return statements (including extended return statements), and requeue statements (respectively) as reporting. "raise" is a shorthand for both "explicit_raise" and "reraise".

If "explicit_raise" is given as a parameter, the procedure `Ada.Exceptions.Raise_Exception` is automatically added to the list of procedures for both Check and Search, unless it is explicitely specified as a parameter in a rule; and similarly `Ada.Exceptions.Reraise_Occurrence` is added for "reraise". This way, it is possible to consider them as reporting procedures for Check (for example) and not for Search.

A handler where *all* exceptions are uncontrolled is not controlled at all (i.e. it is allowed to be non reporting). Otherwise, the rule reports if the handler does not contain at least one of the <report-item> in each possible path of the handler. If the <report-item> appear only in **if** or **case** statements, but not in all possible paths, or if they appear only in the body of **loop** statements, the rule will issue a message asking for a manual verification, since it cannot be statically determined whether the proper treatment happens in every case.

Note that the purpose of this rule is to require the reporting calls to be "eye-visible", i.e. textually written in the exception handler. For example, the rule will accept a call to a procedure inside the sequence of statements of a package body declared in some inner block; however, it will not accept the same call if it is in the sequence of statements of a package instantiation (unless the generic package is itself mentionned as reporting), because the call is not "eye-visible". For the same reason, a call to a reporting function which happens as the default value of an omitted parameter in some other call will not be accepted.

This rule can be given once for each of check, search and count. This way, it is possible to have a level considered a warning (search), and one considered an error (check).

Ex:

```
-- Make an error if exception is not reraised and does not call
-- Reports.Trace, but make it only a warning if the exception is an
-- IO exception or Constraint_Error:
check silent_exceptions (not ada.io_exceptions,
                         not standard.constraint_error,
                         raise,
                         reports.trace);
search silent_exceptions (raise, reports.trace);

-- check handlers that do not reraise the exception, except for
-- IO exceptions:
check silent_exceptions (not Ada.IO_Exceptions, reraise);

-- Same for predefined exceptions, except Constraint_Error:
check silent_exceptions (not Standard, with Standard.Constraint_Error,
                         reraise);
```

```
-- Same for all exceptions named User_Error, wherever they are declared,
-- and no others
check silent_exceptions (with all User_Error, reraise);

-- Same for "when others" handlers
check silent_exceptions (with others, reraise);
```

5.50.3 Limitations

Currently, "return" includes all return statements. It would be nice to separate function returns from procedure or accept returns. This is expected to be done in some future version of AdaControl.

There are two cases that are not statically checkable, and thus may not be identified by this rule: if an exception is raised in an inner block statement and handled locally, and if the exception handler aborts the current task.

If a reporting function is given, there are a few cases where the calls will not be recognized:

- inside a pragma
- in a representation clause
- in a code statement (i.e. as a field of a machine code instruction)

This limitation is intentional: these are such weird places to call a reporting function that it seems better to draw attention to it...

5.51 Simplifiable_Expressions

This rule controls expressions that can be simplified in various ways.

5.51.1 Syntax

```
<control_kind> simplifiable_expressions [(<subrule> {, <subrule>})];
<subrule> ::= conversion   | logical     | logical_false | logical_not |
              logical_true | parentheses | range
```

5.51.2 Action

Without parameters, all kinds of simplifiable expressions are controlled; otherwise, the controlled expressions depend on the subrule:

- "conversion" controls type conversions where the expression is of a universal type (a litteral or named number), or where the target subtype is either the same as the expression's subtype, or the first named subtype of the expression.
- "logical_true" controls redundant boolean expressions of the form `<expr> = True` (or `/=`), and "logical_false" does the same for comparisons with `false`.
- "logical_not" controls **not** operators whose argument is a comparison (which could be inverted).
- "logical" is the same as specifying "logical_true", "logical_false" and "logical_not".

- "parentheses" controls unnecessary parentheses like those surrounding the expression of an assignment, an "if" or a "case" statement, or those that are not required by operators precedence rules.
- "range" controls expressions of the form `T'First .. T'Last` that should be `T'range` (or even simply `T`).

This rule can be given at most once for each subrule.

Ex:

```
search simplifiable_expressions (parentheses);
check  simplifiable_expressions (range, logical);
```

5.51.3 Tips

There are cases where parentheses may seem unnecessary, but are (purposedly) not reported by this rule. Consider for example:

```
X := A + (B + C);
```

Removing the parentheses would change the expression to mean:

```
X := (A + B) + C;
```

If the `"+"` operator has be redefined and is no more associative, this would actually change the meaning of the program. In a less contrieved example, note that:

```
X mod (A*B)
```

is *not* the same as:

```
X mod A * B
```

For these reasons, and to make the rule easier to understand for the user, the rule does not report unnecessary parentheses between operators of identical priority levels.

Conversion of universal value is never necessary, however there are cases where overloading resolution may require the conversion to be replaced by a qualification, rather than being simply removed.

5.52 Simplifiable_Statements

This rule controls statements that can be removed or simplified in various ways without changing the meaning of the program.

5.52.1 Syntax

```
<control_kind> simplifiable_statements [(<subrule> {, <subrule>})];
<subrule> ::= block  | dead | handler        | if          | if_for_case |
              if_not | loop | loop_for_while | nested_path | null
```

5.52.2 Action

Without parameter, all kinds of simplifiable statements are controlled; otherwise, the controlled statements depend on the subrule:

- `block` controls block statements that have no labels, no declarations, and no exception handlers.

- `dead` controls dead code, i.e. statements that are statically known to be never executed. This includes statements that follow a **return**, **requeue**, or **goto** statement, or an **exit** statement that is either unconditional or whose condition is statically known to be true. It includes also **while** statements and **if** statements (including **elsif** paths) whose condition is statically false, and **for** loops whose range is statically empty.
- `handler` controls "trivial" exception handlers, i.e. handlers whose sequence of statements includes only a single **raise** statement without an exception name. However, a handler is not reported if there is also a non trivial handler for **others**. These examples show the situation:

```
exception
   when Constraint_Error =>  -- Reported (no when others)
      raise;
end;

exception
   when Constraint_Error =>  -- Reported (trivial when others)
      raise;
   when others =>         -- Reported
      raise;
end;

exception
   when Constraint_Error =>  -- Not reported (non trivial when others)
      raise;
   when others =>
      Put_Line ("Error");
end;
```

- `if` controls **if** statements with an **else** path that contains only **null** statements (and can thus be removed).
- `if_for_case` controls usage of **if** statements that could be replaced by **case** statements. An **if** statement is assumed to be replaceable if it has at least one **elsif** and all conditions are comparisons (or membership tests, possibly connected by logical operators) of the same discrete variable with static values. Typically, this subrule will spot constructs like:

```
if X = 1 then
   ...
elsif X = 2 or X = 3 or X = 4 then
   ...
elsif X >= 5 and X <= 10 then
   ...
elsif X in 11 .. 20 then
   ...
else
   ...
end if;
```

- `if_not` controls **if** statements with an **else** path and no **elsif** path, and where the condition is given in negative form (i.e. it is a **not**, or a `"/="` comparison). Such statements could be made positive (and thus less error-prone) by interverting the **if** and **else** paths.
- `nested_path` controls paths from **if** statements that can be moved outside. This happens if the **if** has only **then** and **else** paths, and either of them ends with a "breaking" statement (**raise**, **return**, **exit** or **goto**); in this case, the other path needs not be nested inside the **if** statement. However, if both paths end with the *same* "breaking" statement, no error is reported. In short, the rule signals the following examples:

```
if Cond then
   return;
else
   I := 1;
end if;

if Cond then
   I := 1;
else
   return;
end if;
```

  because they can be changed to:

```
if Cond then
   return;
end if;
I := 1;

if not Cond then
   return;
end if;
I := 1;
```

  The rule will not signal the following example, where both paths end with the same "breaking" statement (**return**), because it would break the symetry of the statement:

```
if Cond then
   return 1;
else
   return 2;
end if;
```

- `null` controls **null** statements that serve no purpose and can be removed. Note that if a **null** statement carries a label, it is not considered simplifiable.
- `loop` controls `while` loop statements where the condition is a plain `True`, and can thus be changed to simple loops.
- `loop_for_while` controls simple loop statements whose first statement is an **exit** (for the same loop), and which can therefore be turned into a `while` loop.

This rule can be given at most once for each subrule.

Ex:

```
check simplifiable_statements (block, null);
search simplifiable_statements (if);
```

5.52.3 Tips

`loop` may seem a strange thing to check, since no Ada programmer is supposed to write this. However, experience shows that it is a good indicator of code written by people who did not get proper Ada training. Such code is certainly worth a peer review...

5.53 Statements

This rule controls usage of certain Ada statements.

5.53.1 Syntax

```
<control_kind> statements (<subrule> {, <subrule>};

<subrule> ::=
   any_statement            | abort                   |
   accept                   | accept_return           |
   assignment               | asynchronous_select     |
   block                    | case                    |
   case_others              | case_others_null        |
   code                     | conditional_entry_call  |
   declare_block            | delay                   |
   delay_until              | dispatching_call        |
   effective_declare_block  | entry_call              |
   entry_return             | exception_others        |
   exception_others_null    | exit                    |
   exit_expanded_name       | exit_for_loop           |
   exit_outer_loop          | exit_plain_loop         |
   exit_while_loop          | exited_extended_return  |
   extended_return          | for_in_loop             |
   for_iterator_loop        | for_of_loop             |
   function_return          | goto                    |
   if                       | if_elsif                |
   inherited_procedure_call | labelled                |
   loop_return              | multiple_exits          |
   named_exit               | no_else                 |
   null                     | procedure_return        |
   raise                    | raise_locally_handled   |
   raise_nonpublic          | raise_standard          |
   redispatching_call       | reraise                 |
   requeue                  | selective_accept        |
   simple_block             | simple_loop             |
   terminate                | timed_entry_call        |
   unconditional_exit       | unnamed_block           |
   unnamed_exit             | unnamed_loop_exited     |
   unnamed_for_loop         | unnamed_multiple_loop   |
```

```
unnamed_simple_block     | unnamed_simple_loop    |
unnamed_while_loop       | untyped_for            |
while_loop
```

5.53.2 Action

Subrules that are Ada keywords control the corresponding Ada statements. The meaning of other subrules is as follows:

- `any_statement` controls all statements. This is of course not intended to forbid all statements in a program (!), but *counting* all statements can be quite useful.
- `accept_return` controls return statements that return from an **accept** statement, `entry_return` controls return statements that return from a (protected) entry body, and `procedure_return` controls return statements that return from a procedure. `loop_return` controls return statements (including extended return statements) that appear inside a **loop** statement.
- `assignment` controls all assignment statements.
- `asynchronous_select` controls the **select** ... **then abort** statement. `conditional_entry_call` controls the **select** ... **else** statement. `timed_entry_call` controls the **select** ... **or delay** statement. `selective_accept` controls the regular **select** statement.
- `block` controls all block statements, while `unnamed_block` controls blocks without a name, `declare_block` controls blocks with an explicit **declare** (even if the declarative part is empty), and `effective_declare_block` controls blocks with a declarative part that includes anything else than **use** clauses and pragmas. `simple_block` controls block statements that have no declarative part (or an empty declarative part) and no exception handlers, and `unnamed_simple_block` does the same, but only for blocks without a name.
- `case` controls all `case` statements.
- `case_others` controls any **when others** path in a **case** statement, while `case_others_null` controls only **when others** paths in a **case** statement that contain only **null** statements.
- `code` controls code statements.
- `delay` controls only relative **delay** statements, while `delay_until` controls absolute **delay until** statements.
- `dispatching_call` controls all dispatching calls. Note that this subrule controls dispatching procedure calls as well as dispatching function calls, although the latter is technically an expression and not a statement. `redispatching_call` does the same, but only for dispatching calls that are (directly or indirectly) inside a primitive operation of a tagged type.
- `entry_call` controls all entry call statements, including those that are part of a conditional or timed entry call statement.
- `exit` controls all exit statements, while `exit_for_loop`, `exit_while_loop`, and `exit_plain_loop` control **exit** statements that terminate **for** loops, **while** loops, and plain (neither **for** nor **while**) loops, respectively. `unconditional_exit` controls **exit** statements without a **when** condition. `multiple_exits` controls loop that have more than one **exit** statement. `unnamed_loop_exited` controls exit statements that terminate an

unnamed loop. `exit_outer_loop` controls **exit** statements that exit from an outer loop (i.e. not the innermost one). `exit_expanded_name` controls named **exit** statements where the name is given as an expanded name.

- `exception_others` controls any **when others** exception handler, while `exception_others_null` controls only **when others** exception handlers that contain only **null** statements.
- `extended_return` controls extended return statements (i.e. the Ada 2005 construct "**return** `V : T` **do** `...` **end return**"). `exited_extended_return` controls extended return statements that can be left without actually returning due to an **exit** or **goto** statement within their sequence of statements.
- `for_loop` controls all **for** loops, while `for_in_loop` controls only the traditional form of **for** loop (**for** `I` **in** *`range`* **loop**), `for_iterator_loop` controls the iterator form (**for** `I` **in** *`Iterator`* **loop**), and `for_of_loop` controls the components form (**for** `V` **of** `...` **loop**) (the three latter forms are not available with the old gnat version of AdaControl).
- `function_return` controls return statements (including extended return statements) from functions. Obviously, return statements cannot be forbidden in functions; this keyword controls that there is only one return statement in the body of functions, and at most one return statement in each exception handler of the exception part of functions.
- `if` controls all **if** statements.
- `if_elsif` controls **if** statements that have at least one **elsif**.
- `inherited_procedure_call` controls calls to procedures that have been inherited by a derived type and not redefined.
- `labelled` controls statements with a label (true statement labels, not block and loop names).
- `named_exit` controls **exit** statements with a loop name.
- `no_else` controls **if** statements that have no **else** path.
- `null` controls all **null** statements.
- `raise` controls all **raise** statements.
- `reraise` controls **raise** statements in exception handlers that reraise the same exception, and calls to the `Ada.Exceptions.Reraise_Occurrence` procedure.
- `raise_standard` controls **raise** statements that raise one of the predefined exceptions (those declared in package `Standard`). `raise_nonpublic` controls statements that raise exceptions that are neither predefined nor defined in the visible part of a package which is the enclosing library unit of the statement. `raise_locally_handled` controls statements that raise an exception which is handled by a handler in the same subprogram body as the statement.

  Note that for these subrules, the exception can be raised either by a **raise** statement, or by a call to `Ada.Exceptions.Raise_Exception` where the raised exception is statically determinable.
- `simple_loop` controls simple loops, i.e. those that are neither **while** nor **for** loops.
- `unnamed_exit` controls **exit** statements without a loop name that exits from a named loop.

- `unnamed_for_loop`, `unnamed_simple_loop`, and `unnamed_while_loop` control loops of the given kind that are not named.
- `unnamed_multiple_loop` controls nested loops that are not named (i.e. under this rule, only loops that contain no inner loop, and are not nested in another loop, are allowed not to be named). The kind of loop (plain, **for**, **while**) is not considered.
- `untyped_for` controls **for** loops that use a range without an explicitely named type (i.e. **for** `I` **in** `1..10` **loop**). Using a `'Range` attribute is OK.
- `while_loop` controls all **while** loops.

Ex:

```
search statements (delay);
check  statements (goto, abort);
check  statements (case_others_null, exception_others_null);
```

5.53.3 Tips

It may seem strange to control things like **if** or **case** statements, since no coding standard would prohibit their use. However, this may be useful, especially with "count", for statistical purposes, like measuring the ratio of **if** to **case** statements.

The plain "raise" subrule controls the **raise** statement, and only this one. If you want to check all places where exceptions can be raised, use also the "entities" rule like this:

```
"all raise": check statements (raise),
             check entities   (Ada.Exceptions.Raise_Exception,
                               Ada.Exceptions.Reraise_Occurrence);
```

Other subrules of the "raise" family are more about which kind of exception is being raised, and therefore control also exceptions raised by calling the procedures from `Ada.Exceptions`.

"inherited_procedure_call" controls only *procedure* calls. For function calls, see rule Section 5.18 [Expressions], page 67.

5.54 Style

This rules controls usage of various "general" Ada coding style.

5.54.1 Syntax

```
<control_kind> style;
<control_kind> style (casing_attribute,  <casing_kw> {,<casing_kw>});
<control_kind> style (casing_identifier, <casing_kw> {,<casing_kw>});
<control_kind> style (casing_keyword,    <casing_kw> {,<casing_kw>});
<control_kind> style (casing_pragma,     <casing_kw> {,<casing_kw>});
<control_kind> style (compound_statement);
<control_kind> style (default_in);
<control_kind> style (exposed_literal, <type_kw>, {, <value_place>});
<control_kind> style (formal_parameter_order {, <mode list>});
<control_kind> style (multiple_elements {,<element_kw>});
<control_kind> style (negative_condition);
<control_kind> style (no_closing_name [, <max_lines>]);
```

```
<control_kind> style (numeric_literal, [not] <base> [, <block_size>]);
<control_kind> style (parameter_order {, <mode list>});
<control_kind> style (renamed_entity);

<casing_kw>   ::= uppercase | lowercase | titlecase | original
<element_kw>  ::= [flexible] clause | declaration | statement |
                  handler | begin | end  | then | when |
                  else    | is    | loop | do   | keywords
<mode_list>   ::= <mode> {| <mode>}
<mode>        ::= in   | defaulted_in | access   | in_out  | out |
                  type | procedure    | function | package
<type_kw>     ::= integer | real | character | string
<value_place> ::= <value> | <place>
<value>       ::= [max] <integer number> | <real number> | "<pattern>"
<place>       ::= constant    | exponent | index | number | pragma |
                  repr_clause | var_init | type
```

5.54.2 Action

The first parameter specifies which style aspect is to be checked:

- "casing_attribute", "casing_keyword", "casing_identifier", and "casing_pragma" control that attributes (respectively keywords, identifiers, or pragmas) use the appropriate casing. "original" (which is allowed only for identifiers) means that identifiers must use the same casing as in their declaration.

  If more than one <casing_kw> is given, it means that any of them is allowed.
- "compound_statement" controls that compound statements span at least a minimum number of lines: 3 for **if** statements, **loop** statements, block statements, and **accept** statements with a body; 4 for **case** statements, selective **accept** statements, and timed entry call statements; and 5 for conditional entry call statements and asynchronous select statements.
- "default_in" controls subprograms, entries and generics declarations that omit an explicit `in` mode for a parameter. Access parameters are not reported, since an an explicit `in` is not allowed in that case.
- "exposed_literal" controls the usage of literals (aka "magic values"), that appear outside of allowed places. The second parameter tells to which kind of literals the rule applies. The (optional) indicated values that follow are allowed at any place; for integers, a single value can be preced by "max", to indicate that all literals whose (absolute) value is less or equal are allowed; for strings, the values are regular expressions. See Appendix B [Syntax of regular expressions], page 149. Commonly allowed values are 0 and 1 for integer literals, 1.0 and 0.0 for real literals and `"^$"` (the empty string) for string literals. At most 20 values of each kind may be specified. In addition, one or several <place> keyword can be used to specify constructs where any literal is allowed: "declaration" stands for any declaration, "constant" for constant declarations, "exponent" for the right parameter of an exponentiation (i.e. `"**"`) function call, "index" for array indexing, "number" for named number declarations, "pragma" for pragma arguments, "repr_clause" for representation clauses, "type" for type (and subtype) declarations, and "var_init" for the initialization expression of variable declarations. If no

<place> is given, it is taken as `number, constant`, i.e. any literal is allowed in named numbers and constant declarations.

- "multiple_elements" controls clauses, declarations, statements, and handlers that do not start on a line of their own (i.e. when there are more than one of these on the same line). Similarly, **begin**, **end**, **then** and **when** are required to be on a line of their own, together with the possible keyword or identifier attached to them and the semi-colon. In addition, the **is**, **loop** or **do** that terminates the first part of some declarations or statements is required to be on the same line as the begining of the element, or on a line of its own.

  Extra parameters specify which kind of element to check; if not specified, all kind of elements are controlled. "keywords" is a shorthand for specifying all keywords. If "flexible" is specified in front of "clause" (not allowed otherwise), it allows a **use** clause to be on the same line as a **with** clause, provided all packages named in the **use** clause are also named in the preceding **with** clause.
- "negative_condition" controls "if" statements with an "else" part and no "elsif", where the condition starts with a **not**, and should therefore preferably be expressed positively.
- "no_closing_name" controls declarations, like package or subprograms, that allow (but do not require) repeating the name at the end of the declaration, and where the closing name is omitted (which is considered bad style in general). However, it can be acceptable to allow the omission of closing names for very short constructs; therefore this rule has an optional parameter specifying the maximum number of lines of a construct for which omitting the closing name is allowed. This rule can be given only once for each of check, search and count. This way, it is possible to have a length considered a warning (search), and one considered an error (check). Of course, this makes sense only if the length for search is less than the one for check. If no length is specified, all occurrences of missing closing names are signaled.
- "numeric_literal" controls the presentation of numeric literals, depending on the base (wich, as required by Ada rules, must be in the range 2..16). If "not <base>" is specified as the second parameter, the given base may not be used for based literals. Otherwise, there must be a third (integer) parameter to specify the size of blocks of digits for that base, i.e. there must be an underscore character to separate digits every <block_size> position. Typically, <block_size> is 3 for base 10, 4 for base 2, etc.
- "parameter_order" and "formal_parameter_order" control the order of the declarations of parameters or generic formal parameters, respectively. Each parameter of the rule consists in one or several of the "mode" keywords, and describes, in order, which kind of parameter is allowed. All modes not specified explicitely are allowed after the ones that are specified. See examples below.

  If no parameter is given, the order for regular parameters is "in" or "access" first, then "in_out", then "out", then "defaulted_in". The order for formal_parameters is "type" first, then "in" "defaulted_in" and "access", then "in_out", then "procedure" and "function", then "package".
- "renamed_entity" controls occurrences of identifiers within the scope of a renaming declaration for them; i.e. it enforces that when an entity has been renamed, the original name should not be used anymore.

Ex:

```
search style (no_closing_name);
search style (no_closing_name, 5);
check style (casing_identifier, original);
check style (default_in);
check style (numeric_literal, 10, 3);
check style (exposed_literal, integer, 0, 1);
check style (exposed_literal, real, 0.0, 1.0);

 -- in parameters (with or without default) and access
 -- parameters must be first, then in out parameters, then
 -- out parameters. In parameters are allowed last if they
 -- have defaults.
check style (parameter_order,
               in | defaulted_in | access,
               in_out,
               out
               defaulted_in);

 -- For generics, formal objects must come first, then formal
 -- types, then formal subprograms, then formal package:
check style (formal_parameter_order,
               in | in_out,
               type,
               procedure | function,
               package);
```

Without parameter, the rule will control all style aspects with parameter values that correspond to the most commonly used cases, i.e. it is equivalent to the following:

```
style (no_closing_name);
style (casing_attribute, titlecase);
style (casing_keyword, lowercase);
style (casing_identifier, original);
style (casing_pragma, titlecase);
style (default_in);
style (negative_condition)
style (multiple_elements)
style (literal, 10, 3);
style (exposed_literal, integer, 0, 1)
style (exposed_literal, real, 0.0, 1.0);
```

5.54.3 Tips

For the "Casing_Identifier" subrule, if the value is "original", subprogram and parameter names from the body are checked against those from the specification (if any). This is what the user would expect, although strictly speaking it is not a usage of the name.

Note that operators always follow the casing rule for keywords, even for calls that use the infix notation (i.e. in `"and"(A, B)`).

Having more than one allowed casing is useful if for example you want to require Titlecase, but accept that the original casing be used (maybe because your editor or pretty-printer forces it).

For the "Exposed_Literal" subrule, negative values can be specified as being allowed; negative numbers are handled as if they were literals. This is what the casual user would expect, but to the language lawyer, "-1" is not a negative literal, it is a unary minus operator applied to the positive value "1".

"compound_statement" was a simplistic way of finding badly laid-out statements, at a time when "multiple_elements" did not control the end or intermediate parts of declarations and statements. It is of little use now that "multiple_elements" has been enhanced.

5.54.4 Limitations

If a predefined operator or an attribute is renamed, the "renamed_entity" subrule cannot check that the original entity is not used in the scope of the renaming. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.55 Terminating_Tasks

This rule controls tasks that can terminate.

5.55.1 Syntax

```
<control_kind> terminating_tasks
```

5.55.2 Action

A task is considered a terminating task if its last statement is not an unconditional loop, or this if this loop is exited. It is also considered terminating if it contains a selective accept with a **terminate** alternative.

Since this rule has no parameters, it can be given only once.

Ex:

```
check terminating_tasks;
```

5.55.3 Tips

There is still one case where a task terminates, which is not reported by this rule: when a task is aborted. This is intended, since there are cases (like mode changes) where a logically non-terminating task is aborted.

If aborts are also to be reported, use the rule "statements (abort)". See Section 5.53 [Statements], page 117.

5.56 Type_Initial_Values

This rule controls that a special constant is declared together with each type, for example to serve as a default initial value.

5.56.1 Syntax

```
<control_kind> type_initial_values [("<pattern>")];
```

5.56.2 Action

This rule controls types that do not feature an initialization constant declared in the same declarative part as the type. If no <pattern> is given, any constant is considered an initialization constant for its type; otherwise, only constants whose name matches the given pattern are considered initialization constants.

Ex:

```
check type_initial_values ("^C_Init_");
```

The above example will ensure that every declared type features a constant of the type whose name starts with "C_Init_".

5.57 Type_Usage

This rule controls usage of indicated types, either individually or by category.

5.57.1 Syntax

```
<control_kind> type_usage (<attribute>, <category> {, <aspect>}]);
<control_kind> type_usage (index, <entity>|<category> {, <aspect>}]);
<category> ::= ()  | access    | array | delta  | digits |
               mod | protected | range | record | tagged | task
<aspect>   ::= [not] representation | pack | size | component_size
```

5.57.2 Action

If the first parameter is an attribute (a name starting with a simple quote), the rule controls all occurrences of the attribute where the prefix designates a type belonging to the <category> given as second parameter.

If the first parameter is "index", the rule controls all array types that have an index of the type given by <entity>, or belonging to the <category> given as second parameter. As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

For both subrules, if one or several <aspect> are given, only types featuring (or not featuring if "not" is given) the provided aspects are controlled.

The meaning of <category> is:

- "()": The type is an enumerated type.
- "access": The type is an access type.
- "array": The type is an array type.
- "delta": The type is a fixed point type (it is not currently possible to distinguish ordinary fixed point types from decimal fixed point types).
- "digits": The type is a floating point type.
- "mod": The type is a modular type.
- "protected": The type is a protected type.
- "range": The type is a signed integer type.
- "record": The type is an (untagged) record type.
- "tagged": The type is a tagged type (including type extensions).

- "task": The type is a task type.

The meaning of <aspect> is:

- "representation": the type has an enumeration representation clause or a record representation clause.
- "pack": the type is the target of a pack **pragma**.
- "size" and "component_size": the type has the corresponding attribute specified.

Ex:

```
-- Don't use the 'Pos attribute for enumerated types with a representation
check type_usage ('Pos, (), representation);

-- Don't use modular type for array indexes
check type_usage (index, mod);
```

5.57.3 Tips

The subrule "index" controls the use of a type as an index at any position and irrespectively of the number of indices of the array. To control a precise pattern of types used as indices, use the rule "array_declarations". See Section 5.3 [Array_Declarations], page 40.

The subrule that uses attribute names does not allow an <entity>. To control occurrences of an attribute on a precise type, use the rule "entities". See Section 5.15 [Entities], page 62.

5.58 Uncheckable

This rules controls cases where it is not possible to guarantee the accuracy of checks performed by AdaControl, and where manual inspection may be required.

5.58.1 Syntax

```
<control_kind> uncheckable [(<subrule> [,<subrule>])];
<subrule> ::= false_positive | false_negative | missing_unit
```

5.58.2 Action

If the keyword "missing_unit" is given, this rule controls missing units, i.e. units given on the command line that are not found (and therefore not controlled) will result in an usual error message.

Otherwise, this rule controls constructs that are not static and prevent other rules from being fully reliable. This rule is special, since it really affects the way other rules behave when they encounter a statically uncheckable construct. Therefore, if a label is given, the message will include the label as usual, with an indication of the rule that triggered the message; if no label is given, the message will include the name of the rule that detected the uncheckable construct, not "uncheckable" itself.

If the keyword "false_negative" is given, the rule will control constructs that could result in false negatives, i.e. possible violations that would go undected, while if the keyword "false_positive" is given, it will control constructs that could result in false positives, i.e. error messages when the rule is not really violated. If no keyword is given, both occurrences are controlled.

As far as statistics are concerned (see Section 4.2.1 [Control kinds and report messages], page 28), "uncheckable" messages from rules are counted under the corresponding rule's statistics (like other messages), but there will be also a count of all "uncheckable" messages under the rule "UNCHECKABLE", and also subtotals corresponding to the number of "uncheckables" for each rule.

This rule can be given only once for each of value of the parameters.

Ex:

```
check uncheckable (false_negative);
search uncheckable (false_positive);
check uncheckable (missing_unit);
```

5.58.3 Tips

This rule is especially important when AdaControl is used in safety critical software, since it will detect constructs that could escape verification. Such constructs should be either disallowed, or require manual inspection. On the other hand, in casual software, it may lead to many messages, since for example dispatching calls are uncheckable with many rules.

5.58.4 Limitation

With "missing_unit", the message does not include a reference to a source location, since there is no place in the source which can be considered as the origin of the error. If you run AdaControl from GPS, there will always be a separate category ("Uncheckable") in the locations window, under which the message will appear, with a file name of "none". Don't try to click on the error message, since GPS will find no file named "none"!

5.59 Unit_Pattern

This rule controls various usage patterns of program units and elements declared in them.

5.59.1 Syntax

```
<control_kind> unit_pattern (Single_Tagged_Type);
<control_kind> unit_pattern (Tagged_Type_Hierarchy);
<control_kind> unit_pattern (Context_Clauses_Order {, <clause_list>});
<control_kind> unit_pattern (Declarations_Order, <target>,
                             {, <declaration_list>});

<clause_list>      ::= <clause> {| <clause>}
<clause>           ::= with | use | use_type | pragma
<target>           ::= package_public | package_private | package_body |
                       subprogram
<declaration_list> ::= <declaration> {| <declaration>}
<declaration>      ::= use                    | use_type                 |
                       use_all_type           | number                   |
                       constant               | variable                 |
                       private_type           | full_type                |
                       subtype                | subprogram_spec          |
                       package_spec           | generic_subprogram_spec  |
```

```
generic_package_spec  | task_spec                |
protected_spec        | subprogram_body          |
package_body          | generic_subprogram_body  |
generic_package_body  | task_body                |
protected_body        | object_renaming          |
subprogram_renaming   | package_renaming         |
exception_renaming    | subprogram_instantiation |
package_instantiation | exception                |
others
```

5.59.2 Action

The checked pattern depends on the given subrule:

- "single_tagged_type" controls that at most one tagged type is declared in any package.
- "tagged_type_hierarchy" controls that tagged types follow packages hierarchy, i.e. that the parent of a type extension (derivation of a tagged type) is declared in the parent unit of the one that declared the derivation.
- "context_clauses_order" controls the order of context clauses (and pragmas) given on top of the unit. Each parameter of the rule consists in one or several of the <clause> keywords, and describes, in order, which kind of clause is allowed. Note that "use_type" covers only the regular **use type** clause, specify also "use_all_type" to include the Ada 2012 **use all type** clause as well. Note that all <clause>s not specified explicitely have no place, and thus are not allowed at all.
- "declarations_order" controls the order of declarations (and use clauses) given in various parts, depending on the second parameter:
  - "package_public" controls elements in the visible part of a package specification;
  - "package_private" controls elements in the private part of a package specification;
  - "package_body" controls elements in the body of a package;
  - "subprogram" controls elements in the body of subprograms (procedures and functions) and entries.

  Each parameter of the rule consists in one or several of the <declaration> keywords, and describes, in order, which kind of declaration is allowed. Note that all <declaration>s not specified explicitely have no place, and thus are not allowed at all, unless "others" is given as the last parameter, in which case it covers all elements not part of any of the preceding parameters. See example below.

Ex:

```
check unit_pattern (single_tagged_type);
check unit_pattern (tagged_type_hierarchy);

-- All with clauses must come first, then use and use type clauses
-- (freely mixed), then pragmas
check unit_pattern (context_clauses_order, with, use | use_type | use_all_type, pragma

-- In the public part of a package, declare constants and named numbers
-- first,then private types, then any of regular types, constants, and
```

```
-- variables, then subprograms specifications (including generics and
-- instantiations), then anything else:
check unit_pattern (declarations_order, package_public,
   number | constant,
   private_type,
   full_type | constant | variable,
   subprogram_spec | generic_subprogram_spec | subprogram_instantiation,
   others);
```

5.59.3 Tips

For "context_clauses_order" and "declarations_order", elements given as part of the same parameter (i.e. with a vertical bar between them) can be freely mixed, then followed by any of the elements of the next parameter, etc. An element may appear several times in different parameters. If the last parameter is "others", any element not mentionned at all is allowed after the ones for which you specify an order; this way, it is possible to specify an order for just some elements, and then don't care for the rest.

Expression functions and null procedures are classified as "subprogram_spec" unless they are the completion of an explicit specification, in which case they are classified as "subprogram_body".

If you don't want a declaration to appear at all, you can also use the rule "declarations". See Section 5.10 [Declarations], page 50.

5.60 Units

This rule controls that all necessary units, and only those, are processed by AdaControl.

5.60.1 Syntax

```
<control_kind> units [(<subrule> [,<subrule>])];
<subrule> ::= unreferenced | unchecked
```

5.60.2 Action

If the keyword `unreferenced` is given, the rule controls compilation units that are part of the set of analyzed units, but withed by no other unit. If the keyword `unchecked` is given, the rule controls compilation units that are withed by other unit(s), but not part of the set of controlled units (except standard units).

This rule can only be given once for each of the subrules.

Ex:

```
check units (unchecked);
search units (unreferenced);
```

5.60.3 Tip

The main program will appear as unreferenced, since it is normally part of the controlled units, and not withed by any other unit. As usual, the corresponding message can be disabled by putting the comment:

```
--## rule line off units
```

on the main program.

5.61 Unnecessary_Use_Clause

This rule controls **use** clauses that do not serve any purpose.

5.61.1 Syntax

```
<control_kind> unnecessary_use_clause [(<subrule> {,<subrule>})];
<subrule> ::= unused | qualified | operator | nested | movable
```

5.61.2 Action

The rule controls **use** clauses that can safely be removed, moved, or changed to a **use type** clause. This happens in the following cases:

- "unused": a **use** clause is given, but no element from the corresponding package is mentionned in its scope. The message starts with "unused:".

  In this case, the **use** clause can safely be removed.
- "qualified": a **use** clause is given, but all elements from the corresponding package are refered to using a qualified name (i.e. prefixed by the name of the package). The message starts with "all uses qualified:".

  In this case, the **use** clause can safely be removed, but you may want to keep it for documentation purposes, since the package is actually used within this scope.
- "operator": a **use** clause is given, but the only elements that do not use a qualified name are operators. The message starts with "only used for operators:".

  In this case, and except for some pathological cases (definition of operators that are not primitive operations of the corresponding type), the **use** clause can be replaced by one or several **use type** clause(s).
- "nested": a **use** clause is given within the scope of an enclosing **use** clause for the same package. The message tells the location of the other **use** clause.

  If you also have a message that the outer **use** clause is unnecessary, this means that all references to the package appear inside the inner **use** clauses, and that the outer one can be removed. If not, you can either remove the inner **use** clauses, or remove the outer one and add more local **use** clauses where necessary.
- "movable": a **use** clause is given in a package specification, but all uses are from the corresponding body. The message starts with "use clause can be moved to body:".

  In this case, the **use** clause can safely be moved to the body, unless it appears in a library package, and there are unqualified references to its elements from child units.

If no parameter is given, all cases are controlled, otherwise only cases corresponding to the specified keyword(s) are controlled. This rule can be given only once for each value of the parameters.

Ex:

```
remove: search unnecessary_use_clause (unused);
use_type: check unnecessary_use_clause (operator);
```

5.61.3 Tip

This rule checks only usage of **use** clauses. The rule "reduceable_scope" can be used to check that **use** clauses do not span unnecessarily to wide a scope. See Section 5.46 [Reduceable_Scope], page 106.

5.61.4 Limitations

There are some rare cases where the rule may signal that a **use** clause is not necessary, where it actually is. There is no risk associated to this since if you remove the **use** clause, the program will not compile.

The first one comes from a limitation of the ASIS standard: if the *only* use of the **use** clause is for making the "root" definition of a dispatching call visible.

The second one comes from a limitation in ASIS-for-Gnat. This happens when the *only* use of the **use** clause is for making an implicitely declared operation (an operation which is declared by the compiler as part of a type derivation) visible, and when:

- the operation is the target of a renaming declaration;
- or the operation is passed as an actual to a generic instantiation;
- or all operands of the operation are universal (i.e. untyped).

Since these problems come from intrinsic limitations of ASIS, there is nothing we can do about it. When this happens, you can disable the unnecessary_use_clause rule using the line (or block) disabling feature. See Section 4.2.4 [Disabling controls], page 30. Note that for the third alternative of the second case, you can also qualify one of the parameters, so it is not universal any more.

5.62 Unsafe_Elaboration

This rule controls (generic) packages that may be subject to elaboration order dependencies.

5.62.1 Syntax

```
<control_kind> unsafe_elaboration;
```

5.62.2 Action

The rule controls library packages (or generic packages) whose elaboration calls or instantiates elements from other units (except language defined units) that are not subject to a **pragma** `Elaborate` or `Elaborate_All`. The elaboration of such packages may depend on elaboration order.

Since this rule has no parameters, it can be given only once.

Ex:

```
check unsafe_elaboration;
```

5.62.3 Tips

If the package contains tasks, they are considered as being part of the elaboration code of the package, since tasks could be started by the elaboration of the package. This is somehow pessimistic in the unlikely case where a package would contain a local task type (whose specification is not part of the package specification) and no task object of that type is declared. Anyway, this could create only false positives, therefore there is no risk associated to it.

5.63 Unsafe_Paired_Calls

This rule controls usage of calls to operations that are normally paired (like P/V operations) and do not follow a "safe" coding pattern.

5.63.1 Syntax

```
<control_kind> unsafe_paired_calls
   (<opening procedure>, <closing procedure> [, <lock type>]);
<opening procedure> ::= <entity>
<closing procedure> ::= <entity>
<lock type>         ::= <entity>
```

5.63.2 Action

The following explanations are given in terms of "locks" since this is the primary use of this rule, however the rule can be used for any calls that need to be properly paired.

The rule can deal with three different kinds of locks:

- *abstract state machines*: There is no "lock" object, locking is done directly inside the procedures. The <lock type> parameter of the rule must not be provided in that case.
- *object abstract data types*: The procedure operates on an object (generally of a private type) representing the "lock" object, passed as an "in out" parameter. The third parameter must be the corresponding type, and the rule will control that all matching pairs of calls refer statically to the same variable.
- *reference abstract data types*: The procedure operates on a reference that designates the "lock" object, passed as an "in"parameter. The third parameter must be the corresponding type, which must be discrete or access, and the rule will control that all matching pairs of calls refer statically to the same value (for discrete types) or to the same constant (for access types).

As usual, the whole syntax for entities is allowed for <entity>. See Appendix A [Specifying an Ada entity name], page 145.

The "safe" coding pattern is defined as follows:

- A call to the first procedure is the first statement of a handled sequence of statements;
- A call to the second procedure is the last statement of the same handled sequence of statements;
- Corresponding calls of a pair use the appropriate value for the "lock" parameter (if any), as explained above.
- There is no other call to either operation in the statements of the handled sequence of statements, except in nested blocks or accept statements; calls in such inner statements shall not reference the same values or variables as outer ones.
- There is an exception handler for "others" in the handled sequence of statements.
- Every exception handler of the handled sequence of statements includes a single call to the second operation, using the appropriate value or variable for the lock parameter.

Typically, the "safe" pattern corresponds to the following structures:

```
-- Abstract state machine
begin
```

```
      P;
      -- Do something
      V;
   exception
      when others =>
         V;
         -- handle exception
   end;

   -- Object abstract data type
   declare
      My_Lock : Lock_Type;
   begin
      P (My_Lock);
      -- Do something
      V (My_Lock);
   exception
      when others =>
         V (My_Lock);
         -- handle exception
   end;

   -- Reference abstract data type
   declare
      Lock_Ptr : constant Lock_Access := Get_Lock;
   begin
      P (Lock_Ptr);
      -- Do something
      V (Lock_Ptr);
   exception
      when others =>
         V (Lock_Ptr);
         -- handle exception
   end;
```

Ex:

```
check unsafe_paired_calls (Semaphore.P, Semaphore.V, Semaphore.Lock_Access);
```

5.63.3 Tips

If the <Lock type> parameter is provided, both procedures must have a single parameter of the given type, it must not correspond to an "out" parameter, and if it corresponds to an "in" parameter, the type must be discrete or access.

This rule can be specified several times, and it is possible to have the same procedure belonging to several rules. For example, if you have a `Mask_Interrupt` procedure that should be matched by either `Unmask_Interrupt` or `General_Reset` (all declared in package `IT_Driver`), you can specify:

```
check unsafe_paired_calls (IT_Driver.Mask_Interrupt,
```

```
                              IT_Driver.Unmask_Interrupt);
check unsafe_paired_calls (IT_Driver.Mask_Interrupt,
                              IT_Driver.General_Reset);
```

Normally, the legality of a rule is checked when the rules file is parsed, and execution does not start if there is any error. However, the legality of the provided type can be checked only during the analysis. If the type is incorrect for some reason, a proper error message is issued and execution stops immediately.

5.63.4 Limitation

Due to a weakness of the ASIS standard, dispatching calls are not considered. Especially, this means that the <Lock type> cannot be class-wide. Such calls are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

Due to a size limitation of internal data structures, this rule can be specified at most 32 times.

5.64 Unsafe_Unchecked_Conversion

This rule controls unchecked conversions between types which are not statically known to have identical sizes.

5.64.1 Syntax

```
<control_kind> unsafe_unchecked_conversion
```

5.64.2 Action

This rule controls instances of `Unchecked_Conversion` between types where the following conditions are not met:

- A size clause has been specified for both types
- Both sizes are equal

Moreover, a special message is given if any of the types is a class-wide type (certainly a very questionable construct!).

Ex:

```
check unsafe_unchecked_conversion
```

5.64.3 Limitation

There are cases where a size clause is given for a type, but AdaControl is unable to evaluate it. This happens especially if the size clause refers to a size attribute of a predefined type, like:

```
for T'Size use Integer'size;
```

This can lead to false positives (i.e. detection of instantiations of `Unchecked_Conversion` that are actually OK. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.65 Usage

This rule controls how certain entitities (variables, constants, types, procedures, functions, exceptions, tasks, protected objects, and generics) are used.

5.65.1 Syntax

```
<control_kind> usage
   (variable|object {,[not] <location> | read | written | initialized});
<control_kind> usage
   (constant {,[not]  <location> | read});
<control_kind> usage
   (type {,[not]  <location> | used});
<control_kind> usage
   (procedure {,[not]  <location> | called | accessed});
<control_kind> usage
   (function {,[not]  <location> | called | accessed});
<control_kind> usage
   (exception {,[not]  <location> | raised | handled});
<control_kind> usage
   (task {,[not]  <location> | called | aborted});
<control_kind> usage
   (protected {,[not]  <location> | called});
<control_kind> usage
   (generic {,[not]  <location> | instantiated});
<control_kind> usage
   (all {,[not]  <location>});

<location> ::= from_visible | from_private | from_spec
```

5.65.2 Action

The first parameter defines the class of entities to be controlled. “object” stands for both “constant” and “variable”, “type” stands for both types and subtypes, and “all” stands for all classes.

If only one parameter is given, usage of all entities belonging to the indicated class are reported . Otherwise, other parameter(s) are keyword that restrict the kind of usage being controlled.

“[not] from_visible”, “[not] from_private”, and “[not] from_spec” restrict entities being checked to those that appear (or not) in (generic) package specifications, in the visible part, in the private part, or in any part, respectively. “accessed” (available for subprograms only) restricts entities being checked to those that appear as the prefix of a `'Access` or `'Address` attribute. Other keywords carry their obvious meaning, and are allowed only where appropriate. The rule will output the information only for objects that match all the conditions given. A combination of parameters can be given only once for each of “check”, “search”, and “count”.

The report includes the kind of unit that declares the entity (normal unit, instantiation, or generic unit), the part where it is declared (visible or private) if it is declared in a (generic)

package, and whether the entity is known to be initialized, read, written, raised, handled, called, or aborted, depending on the entity's class. Some combinations give an extra useful message (for example, a variable which is initialized and read but not written will produce a "could be declared constant" message).

Variables of an access type and variables of an array type whose components are of an access type (or arrays of an access type, etc.) are always considered initialized, since they are initialized to `null` by the compiler.

Variables that cannot be assigned to (i.e. variables of an array type with some null dimension, or variables of a discrete type whose range includes no values) are specially recognized as "pseudo-constants": there is no message that they are not written to (since it is not possible), but there is an indication that they are pseudo-constants.

The subrules "procedure" and "function" check only regular subprograms, not protected ones. On the other hand, the subrule "protected" controls all calls to any protected subprogram or entry.

Exceptions raised by calling `Raise_Exception` and tasks aborted by calling `Abort_Task` are properly recognized as exceptions being raised and tasks being aborted, respectively.

In the case of entities declared in generic packages, the rule will report on usage of the entities for each instantiation, as well as on global usage for the generic itself. Usage for an instantiation will include usage in the generic itself (i.e. if the generic writes to a variable, the variable will be marked as "written" for each instantiation). Usage for the generic itself is the union of all usages in all instantiations (i.e., if a variable from any instantiation is written to, the variable from the generic will be marked as written). Therefore, if the rule reports that a variable in a generic package can be declared constant, it means that no instance of this variable from any instantiation is being written to. But bear in mind that this can be trusted only if all units from the program are analyzed. See [limitation], page 137.

Note that usage of entities whose declaration is not processed (like, typically, elements declared in standard packages like `Ada.Text_IO`), is not reported. For the same reason, it is not possible to control usage of predefined operators (since they have no declaration).

Ex:

```
-- No variable in package spec; check usage otherwise
Package_Variable: check usage  (variable, from_spec);
Constantable    : search usage (variable, not from_spec, read,
                                          initialized, not written);
Uninitialized   : check usage  (variable, not from_spec, read,
                                          not initialized, not written);
Removable       : search usage (object,   not from_spec, not read);

-- Check exceptions that are never raised
-- generics that are never instantiated
-- and protected objects that are never called
check usage (exception, not raised);
check usage (generic, not instantiated);
check usage (protected, not called);
```

```
-- Find how many tasks are declared, and report those
-- that may be aborted
count usage (task);
check usage (task, aborted);
```

5.65.3 Tips

Constants that are never used, exceptions that are never raised or handled, tasks that are never called, etc. are suspicious. Moreover, some useful compiler warnings (like those about variables that should be declared constants) are not output for variables declared in library packages, and even in some other contexts (at least with GNAT). This rule can check these kind of things, project wide.

Some of these checks make sense only for entities declared in package specifications; for example, variables are often discouraged in package specifications, or need at least some extra control. That's why it can be useful to restrict some checks to package specifications.

Note that an unspecified parameter in a rule stands for two rules (positive and negative form of the missing parameter). I.e.:

```
search usage (variable, from_spec, read, written);
```

is the same as:

```
search usage (variable, from_spec, read, written, initialized);
search usage (variable, from_spec, read, written, not initialized);
```

Therefore, the following example will complain on the second line that the rule has already been given for this combination of parameters:

```
search usage (variable, from_spec, read, written);
search usage (variable, from_spec, read, written, not initialized);
```

Note that the notion of constants for this rule includes named numbers.

5.65.4 Limitations

The report of this rule is output at the end of the run, and is meaningful only for the units that have been processed; i.e., if it reports "variable not read", it should be understood as "not read by the units given". In order to have meaningful results, it is therefore advisable to use this rule on the complete closure of the program.

An exception can be raised by passing its `'Identity` to a procedure that will in turn call `Raise_Exception` (and similarly for `Abort_Task`). These cases are not statically determinable, and therefore not recognized by AdaControl. However, these cases can be identified by searching the use of the `'Identity` attribute with the following rule:

```
check entity (all 'Identity);
```

If an object is the prefix of a `'Access`, `'Unchecked_Access`, or `'Address` attribute, it can be used through the access (or address) value in ways that are not statically analyzable. The same happens if objects are targets of dynamic renamings. Such cases are detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

Due to a weakness of the ASIS standard, it is not possible to know the mode (**in**, **out**) of variables used as parameters of dispatching calls. Such variables are considered to be read and written at the point of the call, therefore possibly creating false positives (which is safer than false negatives). Use of such constructs is detected by the rule "uncheckable". See Section 5.58 [Uncheckable], page 126.

5.66 Use_Clauses

This rule controls usage of **use** clauses.

5.66.1 Syntax

```
<control_kind> use_clauses
   [([<subrule>,] <package name> {, <package name>})];
<subrule> ::= package | local | global | type | type_local | type_global
```

5.66.2 Action

The rule controls every **use** or **use type** clause, *except* those that name one of the mentioned packages/types. It is therefore possible to allow **use** or **use type** clauses just for certain packages/types.

If the keyword "package" is given (or no keyword at all), all package **use** clauses are controlled. If the keyword "global" is given, only **use** clauses that appear in context clauses (i.e. together with the **with** clauses) are controlled; if the keyword "local" is given, only **use** clauses that appear as declarations are controlled.

If the keyword "type" is given, all **use type** clauses are controlled. If the keyword "type_global" is given, only **use type** clauses that appear in context clauses (i.e. together with the **with** clauses) are controlled; if the keyword "type_local" is given, only **use type** clauses that appear as declarations are controlled.

This rule can be given at most once for each of check, search and count. This way, it is possible to have a level considered a warning (search), and one considered an error (check).

Ex:

```
-- Global use clauses are disallowed, local ones only for IO:
check use_clauses (global);
check use_clauses (local, Ada.Text_IO, Ada.Wide_Text_IO);

-- No use type in context clauses, count types that are "use type"'d
check (type_global);
count (type);
```

5.67 With_Clauses

This rule controls **with** clauses that should be removed or moved to a better place.

5.67.1 Syntax

```
<control_kind> with_clauses [(<subrule> [, <subrule>])];
<subrule> ::= multiple_names | reduceable | inherited
```

5.67.2 Action

The parameters are subrule keywords that determine which kind of control is performed:

- `multiple_names` controls any **with** clause that mentions more than one unit name.
- `reduceable` reports:
  - Redundant **with** clauses, i.e. clauses given more than once for the same unit. This includes the case where the same **with** clause is given in a specification and the

corresponding body, and the case of renamings of a same unit (i.e. `Text_IO` and `Ada.Text_IO`). Note that giving a **with** clause in a unit, and repeating it in a child unit (or subunit) is *not* considered redundant.

- Unused **with** clauses, i.e. when nothing from the withed unit is referenced in the corresponding unit. Use of a package name in a **use** clause is *not* considered a usage of the package. The rule signals when a withed unit is not used in a unit, but used in one or more of its subunits. If an unused **with** clause is given on a package specification, the message reminds that it migh be useful for child units.
- Moveable **with** clauses, i.e. when the withed unit is not used in the specification, but only in the body, and should be moved to the body, or when the withed unit is only used in the private part, and could be replaced by a **private with**.

- `inherited` controls child units and subunits that reference a unit which is not directly withed, i.e. when withed only from a parent (or enclosing) unit. Although Ada rules imply that a **with** clause carries on to child units and subunits, it can be considered better practice to ensure that every compilation unit withes directly the units it needs.

Each of the keywords can be given at most once. If no keyword is given, both `reduceable` and `inherited` are assumed.

Ex:

```
check with_clauses (multiple_names, reduceable);
search with_clauses (inherited);
```

5.67.3 Variables

Variable	Values	Default	Effect
Check_Private_With	off/on	on	if you are working in pure Ada 95, you may not want messages that a **with** can be replaced with a **private with**. Setting this variable to off disables these messages.

5.67.4 Tips

A **with** clause can safely be removed if it is unused, and no child unit (or subunit) reports that the unit is inherited.

6 Examples of using AdaControl for common programming rules

In most projects, there are *programming rules* that define the way a program should be written. AdaControl performs controls, i.e. it finds occurrences of certain kinds of constructs. In this chapter, we give examples of commonly found programming rules, and how the corresponding controls can be written.

6.1 Migrating from Gnatcheck

The file `gnatcheck.aru` in directory `rules` gives the AdaControl equivalents of rules checked by Gnatcheck. This version of AdaControl covers most of Gnatcheck rules. For rules where Gnatcheck requires a parameter, the AdaControl rule is given for the default value, or with an example value. Small differences in semantics are indicated by a comment that starts with "Difference:".

This file is not intended to be used directly, but as an example on how to convert Gnatcheck rules into AdaControl rules. Note that in many cases, AdaControl is much more general than Gnatcheck. The file follows as strictly as possible the rules as defined by Gnatcheck, but if you are migrating from Gnatcheck to AdaControl, you may want to use the more powerful forms provided by AdaControl.

6.2 Rules files provided with AdaControl

The `rules` directory provides also rules files that can be sourced to enforce some commonly encountered general rules.

Identifiers from Standard shall not be redefined

Use file `no_standard_entity.aru`.

Identifiers from System shall not be redefined

Use file `no_system_entity.aru`.

Predefined IO packages shall not be used

Use file `no_io.aru`.

Standard package XXX shall not be used

File `no_standard_unit.aru` controls usage of *all* standard packages. Comment out those that you do want to allow.

Obsolescent features shall not be used

Use file `no_obsolescent_features.aru`. Not all obsolescent features are controlled, but most of them (those that are most worth checking) are.

Gnat specific attributes shall not be used

Use file `no_gnat_attribute.aru`

Features from annex X shall not be used

Use file `no_annex_X.aru`.

The Ravenscar profile shall be enforced

Use file `ravenscar.aru`.

Note that not all of the restrictions of the Ravenscar profile are currently controlled, but many are, and we expect later releases of AdaControl to increase the number of controlled features. In some cases (like "Detect_Blocking"), AdaControl does a better job than the profile, since it can detect statically situations that the profile only requires to be detected at run-time. The rule file is also slightly more restrictive than the profile; for example, the restriction "no_task_allocation" only disallows task allocators, while this rule file controls the declaration of access types on tasks.

NASA coding guidelines shall be enforced

Use file `nasa.aru`. This file is an example of how to convert guidelines (available from http://fsw.gsfc.nasa.gov/gds/code_standards_ada.pdf) into an AdaControl rules file.

Ada 83 unit names shall not be used (i.e. use `Ada.Text_IO`, not `Text_IO`)

Use file `no_83_unit_name.aru`.

New reserved words of Ada 2005/2012 shall not be used

Use file `reserved_2005.aru`. (the file name mentions only 2005, but it checks also for 2012 - after all, there is only one extra reserved word).

Measurements for the SQALE method

AdaControl can provide measurements required by the SQALE quality measurement method. The corresponding file is `SQALE.aru`.

For information about the SQALE method, please refer to J-P Rosen's paper at http://www.adalog.fr/publicat/sqale.pdf

6.3 Automatically checkable rules

Below are examples of rules that can be directly checked by AdaControl.

Goto statement shall not be used

```
check statements (goto);
```

Functions shall not have out or in out parameters (Ada 2012)

```
check parameter_declarations (out_parameters,    max 0, function);
check parameter_declarations (in_out_parameters, max 0, function);
```

Short circuit forms should be preferred over corresponding logical operators

```
Use_Short_Circuit: search expressions (and, or);
```

Aggregates should be used for full assignments to structured variables, unless it is a record with a single component

```
check multiple_assignments (groupable, given 2, ratio 100);
```

All loops that contain exit statements must be named, and the name must be given in the exit statement

```
check statements (unnamed_loop_exited);
check statements (unnamed_exit);
```

All type names must start with "T_"

```
check naming_convention (type, "^T_");
```

All program units must repeat their name after the "end"

```
check style (no_closing_name);
```

Pragma Suppress is not allowed

```
check pragmas (suppress);
```

Ada tasking must not be used

```
check declarations (task);
```

"=" and "/=" shall not be used between real types

```
check expressions (real_equality);
```

All tasks must provide an exception handler that calls "Failure" in the case of an unhandled exception

```
check exception_propagation (task);
check silent_exceptions (failure);
```

Unchecked_Conversion shall not be used

```
check entities (ada.unchecked_conversion);
```

No global variable shall be declared in the visible part of a package specification

```
check usage (variable, from_spec);
```

Predefined numeric types of the language shall not be used

```
check entities (standard.Integer,
                standard.short_integer,
                standard.long_integer,
                standard.Float,
```

```
                    standard.short_float,
                    standard.long_float);
```

Access to subprograms shall not be used

```
check declarations (access_to_sp);
```

Abort statements shall not be used

```
check statements (abort);
```

There shall be only one instantiation of Ada.Numerics.Generic_Elementary_Functions for each floating point type

```
-- Put a --##RULE LINE OFF GEF
-- for the one which is allowed
GEF: check Instantiations (Ada.Numerics.Generic_Elementary_Functions);
```

A local item shall not hide an outer one with the same name

```
check Local_Hiding;
```

There shall be no IOs in exception handlers

```
check entity_inside_exception (ada.Text_IO.put, ada.Text_IO.put_line,
                               ada.Text_IO.get, ada.Text_IO.get_line);
```

Note that this checks for all overloaded procedures, but only those dealing with characters and strings (those defined directly within Ada.Text_IO). If the names "get" and "put" are not used for anything else than IOs, a more general form can be given as:

```
check entity_inside_exception (all get,      all put,
                               all get_line, all put_line);
```

This will check that no entity with the corresponding names appear in exception handlers.

Exceptions shall not be used

```
No_Exception: check declarations (exception, handlers);
No_Exception: check statements (raise);
No_Exception: check entities (Ada.Exceptions);
```

This will check that no exception is declared, no exception handler is provided, and no exception is raised, not even through the services of the package `Ada.Exceptions`.

No procedure exported to C shall propagate exceptions

```
check exception_propagation (interface, C);
```

There shall be no Unchecked_Conversion to or from Address

```
check instantiations (ada.unchecked_conversion, system.address);
check instantiations (ada.unchecked_conversion, <>, system.address);
```

There shall be no use clause except for Text_IO

```
check use_clauses(ada.text_IO);
```

Use explicit list of values in case statements rather than "when others"if the "when others" would cover less than 10 values

```
check Case_Statement(min_others_span, 10);
```

If a block is more than 20 lines long, it must be named

```
check Max_Size(unnamed_block, 20);
```

Exceptions shall not be handled except by main program

```
check declaration (handlers)
```

This check will be disabled for the exception handler of the main program.

Each unit has a header starting with a fixed format, and must contain at least 10 lines of comments

```
check header_comments (model, "header.txt");
check header_comments (minimum, 10);
```

The file `header.txt` contains the required header (as regexps), like:

```
^--*{50}$
^-- This is a header$
```

6.4 Rules that need manual inspection

Below are examples of rules that require manual inspection, but where AdaControl can be used to identify suspicious areas.

All usages of the 'ADDRESS attribute shall be justified and documented

```
search entities (all 'address);
```

Specifying an address for a variable shall be restricted to hardware interfacing

```
search representation_clauses(address);
```

There shall be no memory leakage

```
search Allocators;
```

This rule identifies all allocations, and thus can be used to check that all allocated elements are properly deallocated.

Appendix A Specifying an Ada entity name

A.1 General syntax

Many rules can take Ada entities as parameters. Each time a rule uses the category <entity>, it refers to an Ada entity that can be specified with the following syntax:

```
<entity> ::= <full_name> | "all" <simple_name> | "all" <attribute>
```

<full_name> is the full name of the Ada entity, using normal Ada dot notation (with some extensions, see below). Full name means that you give the full expanded name, starting from a compilation unit. This name must be the actual full name, i.e. it must not include any renaming (otherwise the name will not be recognized). For example, the usual Put_Line must be given as Ada.Text_IO.Put_Line, not as Text_IO.Put_Line. Predefined elements (Integer, Constraint_Error) must be given in the form Standard.Integer or Standard.Constraint_Error, since they are logically declared in the package Standard.

<simple_name> is a single identifier, possibly followed by overloading information. No qualification is allowed.

<Attribute> is an attribute name, including the quote. No overloading information is allowed.

<full_name> designates a single entity or several overloaded entities declared in the same place (as identified by the prefix), while all <simple_name> designates all identifiers with the given name in the program, irrespectively of where they appear. all <Attribute> designates all occurrences of the given attribute, irrespectively of what the attribute applies to.

A utility is provided with AdaControl to help you find the full name of an entity. See Section 3.8.1 [pfni], page 22. If you are using GPS with AdaControl plug-ins, it can be accessed directly from the contextual menu. See Section 3.6.2 [Contextual menu], page 19.

A.2 Overloaded names

In Ada, names can be overloaded. This means that you can have several procedures P in package Pack, if they differ by the types of the parameters. If you just give the name Pack.P as the <entity>, the corresponding rule will be applied to all elements named P from package Pack. If you want to distinguish between overloaded names, you can specify a profile after the element's name. A profile has the syntax:

```
"{" [ ["access"] <type-name>
     { ";" ["access"] <type-name> } ]
     ["return" <type-name>] "}"
```

You must specify the *type* name, even if the <entity> declaration uses a subtype of the type; this is because Ada uses types for overloading resolution, not subtypes. Anonymous access parameters are specified by putting access in front of the type name. An overloaded name for a procedure without parameters uses just a pair of empty brackets. If the subprogram is a function, you must provide the return <type-name> part for the return type of the function. The types must also be given as a unique name, i.e. including the full path: if the type is T declared in package Pack, you must specify it as Pack.T. As a convenience, the Standard. is optional for predefined types, so you can write Standard.Integer

as `Integer`. There is no ambiguity, since a type is always declared within some construct. Note that omitting `Standard` works only for *types* that are part of the profile used to distinguish between overloaded Ada entities but that the *Ada entity name* must always contain Standard if it is a predefined element.

Overloaded names can be also be used with the `all <simple_name>` form of the <entity>. In this case, the rule will be applied to all names that are subprograms with the given identifier and matching the given profile, irrespectively of where they appear.

Note that if you use an overloaded name, all overloadable names that are part of the <entity>, including those of the profile, must use the overloaded syntax. For example, given the following program

```
procedure P is
   procedure Q (I : Integer) is
      ...
   end Q;
   procedure Q (F : Float) is
      ...
   end Q;
begin
   ...
end P;
```

If you want to distinguish between the two procedures `Q`, you must specify them as `P{}.Q{Integer}` and `P{}.Q{Float}` (note the `P{}` which specifies an overloaded name for a procedure `P` without parameters).

The names of entities which can not be overloaded (like package, exception, ...) must not be suffixed by braces (e.g. `Ada.Text_IO.Put_Line{Standard.String}`).

A.3 Enumeration literals

Following normal Ada rules, an enumeration literal is considered a parameterless function. If you want to distinguish between overloaded enumeration literals, you can use overloaded names for them. For example, given:

```
package Pack is
   type T1 is (A, B);
   type T2 is (B, C);
end Pack;
```

Ada entities names are:

- `Pack.B{return Pack.T1}`
- `Pack.B{return Pack.T2}`

A.4 Operators

AdaControl handles operators (i.e. functions like `"+"`) correctly. Of course, you must specify such operations using normal Ada syntax: if you define the integer type `T` in package `Pack`, an overloaded name for the addition would be `Pack."+"{Pack.T; Pack.T return Pack.T}`.

A.5 Attributes

It is also possible to designate attributes of entities, using the normal notation (i.e. `Standard.Integer'First`). If the name of an attribute which is a function appears in a name that uses the overloaded syntax, it is not necessary (and actually not allowed) to provide its profile, since there is no possible ambiguity in that case. For example, given:

```
procedure P (I : Integer) is
   type T is range 1 .. 10;
begin
   ...
end P;
```

You can designate the `'Image` attribute for type `T` as `P{Standard.Integer}.T'Image` (the profile of the `'Image` function is not given, as would be necessary for a normal function).

To designate all occurrences of an attribute, use **all** in front of the attribute. To designate only occurrences of an attribute whose prefix is a (sub) type (but any type or subtype), give it as `type'Attr` (i.e. the keyword "type" is put in front of the quote).

all may be used in place of an attribute name to mean "any attribute". See examples below.

```
check entities (all 'Image);      -- Find all occurrences of 'Image
check entities (all type'Length); -- Find all occurrences of 'Length
                                  -- applied to a type

check entities (Standard.Integer'all); -- Find all attributes applied
                                       -- to type Integer
Check entities (all type'all);         -- Find all attributes applied
                                       -- to a type
check entities (all 'all);             -- Find all attributes
```

A.6 Anonymous constructs and extended return statements

There is a special case for elements that are defined (directly or indirectly) within unnamed loops or block statements. Everything happens as if the unnamed construct was named `_anonymous_`. Therefore if you have the following program:

```
procedure P is
begin
   for I in 1..10 loop
      declare
         J : Integer;
      begin
         ...
      end;
   end loop;
end P;
```

You can refer to `I` as `P._anonymous_.I`, and to `J` as `P._anonymous_._anonymous_.J`.

Similarly, an extended return statement is considered "named" `return`. Therefore if you have the following program:

```
function F return Integer is
   I : Integer;
begin
   return I : Integer do
      ...
   end return;
end F;
```

You can refer to the `I` declared in `F` as `F.I`, and to the return object `I` as `F.return.I`.

A.7 Record and protected types components

You can designate the name of a record or protected type component (a "field" name), but to identify it uniquely, you must precede its name by the name of the type. This is a small extension to Ada syntax, but it is the simplest and most natural way to deal with this case. For example, given:

```
procedure P is
   type T is
      record
         Name : Integer;
      end record;
   ...
```

The Ada entity name is `P.T.Name`.

A.8 Formals of access to subprogram types

Similarly, you can designate the formal of an access to subprogram type by prefixing it by the access type. For example, given:

```
procedure P is
   type T is access procedure (X : Integer);
   ...
```

The Ada entity name of the formal is `P.T.X`.

A.9 Limitation

Due to a limitation of ASIS for GNAT, it is not possible to specify a profile with predefined operators; predefined operators without a profile work normally.

```
-- This will not recognize "<" on Standard.Integer:
check entities (Standard."<"{Standard.Integer,
                             Standard.Integer
                             return Standard.Boolean});

-- This will correctly recognize all predefined "<":
check entities (Standard."<");
```

Appendix B Syntax of regular expressions

The following syntax gives the complete definition of regular expressions, as used by several rules. It is taken from the specification of the package `gnat.regpat`, where additional information is available.

```
regexp ::= expr
       ::= ^ expr               -- anchor at the beginning of string
       ::= expr $               -- anchor at the end of string

expr   ::= term
       ::= term | term          -- alternation (term or term ...)

term   ::= item
       ::= item item ...        -- concatenation (item then item)

item   ::= elmt                 -- match elmt
       ::= elmt *               -- zero or more elmt's
       ::= elmt +               -- one or more elmt's
       ::= elmt ?               -- matches elmt or nothing
       ::= elmt *?              -- zero or more times, minimum number
       ::= elmt +?              -- one or more times, minimum number
       ::= elmt ??              -- zero or one time, minimum number
       ::= elmt { num }         -- matches elmt exactly num times
       ::= elmt { num , }       -- matches elmt at least num times
       ::= elmt { num , num2 }  -- matches between num and num2 times
       ::= elmt { num }?        -- matches elmt exactly num times
       ::= elmt { num , }?      -- matches elmt at least num times
                                   non-greedy version
       ::= elmt { num , num2 }? -- matches between num and num2 times
                                   non-greedy version

elmt   ::= nchr                 -- matches given character
       ::= [range range ...]    -- matches any character listed
       ::= [^ range range ...]  -- matches any character not listed
       ::= .                    -- matches any single character
                                -- except newlines
       ::= ( expr )             -- parens used for grouping
       ::= \ num                -- reference to num-th parenthesis

range  ::= char - char          -- matches chars in given range
       ::= nchr
       ::= [: posix :]          -- any character in the POSIX range
       ::= [:^ posix :]         -- not in the POSIX range

posix  ::= alnum                -- alphanumeric characters
       ::= alpha                -- alphabetic characters
       ::= ascii                -- ascii characters (0 .. 127)
```

```
        ::= cntrl               -- control chars (0..31, 127..159)
        ::= digit               -- digits ('0' .. '9')
        ::= graph               -- graphic chars (32..126, 160..255)
        ::= lower               -- lower case characters
        ::= print               -- printable characters (32..127)
        ::= punct               -- printable, except alphanumeric
        ::= space               -- space characters
        ::= upper               -- upper case characters
        ::= word                -- alphanumeric characters
        ::= xdigit              -- hexadecimal chars (0..9, a..f)

char    ::= any character, including special characters
            ASCII.NUL is not supported.

nchr    ::= any character except \()[].*+?^ or \char to match char
            \n means a newline (ASCII.LF)
            \t means a tab (ASCII.HT)
            \r means a return (ASCII.CR)
            \b matches the empty string at the beginning or end of a
               word. A word is defined as a set of alphanumerical
               characters (see \w below).
            \B matches the empty string only when *not* at the
               beginning or end of a word.
            \d matches any digit character ([0-9])
            \D matches any non digit character ([^0-9])
            \s matches any white space character. This is equivalent
               to [ \t\n\r\f\v]  (tab, form-feed, vertical-tab,...
            \S matches any non-white space character.
            \w matches any alphanumeric character or underscore.
               This include accented letters, as defined in the
               package Ada.Characters.Handling.
            \W matches any non-alphanumeric character.
            \A match the empty string only at the beginning of the
               string, whatever flags are used for Compile (the
               behavior of ^ can change, see Regexp_Flags below).
            \G match the empty string only at the end of the
               string, whatever flags are used for Compile (the
               behavior of $ can change, see Regexp_Flags below).
...     ::= is used to indication repetition (one or more terms)
```

Embedded newlines are not matched by the ^ operator. It is possible to retrieve the substring matched a parenthesis expression. Although the depth of parenthesis is not limited in the regexp, only the first 9 substrings can be retrieved.

The operators '*', '+', '?' and '{}' always match the longest possible substring. They all have a non-greedy version (with an extra ? after the operator), which matches the shortest possible substring.

For instance:

```
regexp="<.*>"   string="<h1>title</h1>"   matches="<h1>title</h1>"
regexp="<.*?>"  string="<h1>title</h1>"   matches="<h1>"
```

'{' and '}' are only considered as special characters if they appear in a substring that looks exactly like '{n}', '{n,m}' or '{n,}', where n and m are digits. No space is allowed. In other contexts, the curly braces will simply be treated as normal characters.

Note that if you compiled AdaControl with the `String_Matching_Portable` package, only basic wildcards are available, i.e. only "*" and "?" are supported, where "*" matches any string of character and "?" matches a single character.

Appendix C Non upward-compatible changes

This chapter is intended to users of a previous version of AdaControl, who want to migrate rule files to the latest version. Although we understand the burden of non upward-compatible changes, we consider that making AdaControl more powerful and easier to use is sometimes more important than strict compatibility. Moreover, in most cases the changes are very straightforward and can be done easily by hand, or with scripts if many files are involved.

C.1 Migrating from 1.15r5

C.1.1 Array_Declarations

The extension of aspects to more rules required a slight change in the syntax of the "component" subrule: the keywords "packed", "sized", and "component_sized" have been changed to "pack", "size", and "component_size", respectively.

C.1.2 Multiple_Assignments

Due to new functionalities, and expecting more in the future, the rule has been renamed to "Assignments".

C.1.3 No_Operator_Usage

The syntax has been changed, due to the introduction of "indexing". Moreover, the rule was not consistent, in that the result of "none" was affected by the presence or absence of "logical" (without "logical", "none" included all types, while with it, it counted only those not counted with "logical"). If you want that exact same behaviour (which might not be desirable), change:

```
-- (1)
check no_operator_usage (none);

-- (2)
check no_operator_usage (logical);

-- (3)
check no_operator_usage (none, logical)
                        -- or no parameters
```

to:

```
-- (1)
check no_operator_usage(ignore indexing, ignore logical);
                        -- or no parameters

-- (2)
check no_operator_usage (logical);

-- (3)
check no_operator_usage (not logical),
check no_operator_usage (logical);
```

C.1.4 Object_Declarations

Due to the necessity of avoiding a syntactic ambiguity in the new subrule "type", the keyword "all" is no more allowed in the syntax for the subrule "min_integer_span" (specifying neither "variable" or "constant" still means the subrule applies to both, as before). Change:

```
count object_declarations (min_integer_span, all 8);
```

to:

```
count object_declarations (min_integer_span, 8);
```

C.1.5 Statements

The subrule "exit" was documented as controlling all exit statements, but it did not report exits from **for** and **while** loops if "exit_for_loop" (respectively "exit_while_loop") was also specified. It now behaves as documented, i.e. it controls all **exit** statements.

Note that if you want separate messages for each kind of loop, the new rule "exit_plain_loop" controls exit from plain loops.

C.1.6 Style

The subrule "positional_association" is now a rule of its own, "positional_associations". The order of parameters is different, due to various subrules of the new rule. Typically, change:

```
check style (parameter_association, call, 1);
```

to:

```
check parameter_associations (all, 1, call);
```

Note that the new rule distinguishes between regular array aggregates and aggregates used for enumeration representation clauses.

Modes of the subrules "parameter_order" and "formal_parameter_order" are now separated by "|". With the previous syntax, forgetting a comma was changing the meaning of the rule without introducing a syntax error. Typically, change:

```
check style (parameter_order, in defaulted_in, out in_out);
```

to:

```
check style (parameter_order, in | defaulted_in, out | in_out);
```

C.2 Migrating from 1.14r9

C.2.1 Local_Hiding

Due to the introduction of extra parameters for allowed patterns, it is no more possible to specify the rule several times in the same command. Change:

```
check local_hiding (strict, overloading);
```

to:

```
check local_hiding (strict);
check local_hiding (overloading);
```

The special subrule "overloading_short" has been replaced by a rule variable to choose the report format. Change:

```
check local_hiding (overloading_short);
```

to:

```
set local_hiding.overloading_report compact;
check local_hiding (overloading);
```

C.2.2 Max_Nesting

The value given is now the *nesting* level (consistent with the rule name), no more the maximum *depth*. This is more natural (Max_Nesting(1) means that the construct can be nested once), but it is one less than in previous versions. For example, change:

```
check Max_Nesting (5);
```

to:

```
check Max_Nesting (4);
```

C.2.3 Parameter_Declarations

The subrules have been generalized, using the same syntax for bounds as other rules. Change:

```
check parameter_declarations (min_parameters, 1);
check parameter_declarations (max_parameters, 5);
check parameter_declarations (max_defaulted_parameters, 3);
```

to:

```
check parameter_declarations (all_parameters, min 1, max 5);
check parameter_declarations (defaulted_parameters, max 3);
```

C.3 Migrating from 1.11r4

C.3.1 Expressions

The subrule `Real_Equality` does not control user-defined equality operators any more. This is intended to be more of an improvement than an incompatibily.

C.3.2 Special_Comments

Since the number of subrules is growing, and do not only address 'special" comments, this rule has been renamed to "comments".

C.4 Migrating from 1.10r10

C.4.1 GPS integration

Due to a bug/feature of the GPS interface, if a units file was specified, it did not reappear later in the corresponding box of the Switch/AdaControl dialog. This has been fixed, but you must reenter the units file name in the dialog.

C.4.2 Representation_Clauses

The introduction of categories made some subrules syntactically ambiguous or redundant. In consequence, the subrules "derived_record", "extension_record", and

"tagged_record" have been removed, and the subrules "record", "incomplete_record", and "non_contiguous_record" have been renamed as "layout", "incomplete_layout", and "non_contiguous_layout" respectively. Change:

```
check representation_clause (derived_record);
check representation_clause (extension_record);
check representation_clause (tagged_record);
check representation_clause (record);
check representation_clause (incomplete_record);
check representation_clause (non_contiguous_record);
```

to:

```
check representation_clause (new layout);
check representation_clause (extension layout);
check representation_clause (tagged layout);
check representation_clause (layout);
check representation_clause (incomplete_layout);
check representation_clause (non_contiguous_layout);
```

C.5 Migrating from 1.9r4

C.5.1 Array_Declarations

The subrule "Max_Length" has been changed to "Length", with the possibility to specify both min and max values. Change:

```
check array_declarations (max_length, 100);
```

to:

```
check array_declarations (length, max 100);
```

C.5.2 Declarations

The subrule names "initialized_record_field", "uninitialized_record_field", "initialized_protected_field", and "uninitialized_protected_field" have been changed to "initialized_record_component", "uninitialized_record_component", "initialized_protected_component", and "uninitialized_protected_component", respectively, to be more consistent with official Ada terminology. Change:

```
check declarations (initialized_record_field,
                    uninitialized_record_field,
                    initialized_protected_field,
                    uninitialized_protected_field);
```

to:

```
check declarations (initialized_record_component,
                    uninitialized_record_component,
                    initialized_protected_component,
                    uninitialized_protected_component);
```

The subrule "aliased" has been split into "aliased_constant" and "aliased_variable". The old rule controlled both at the same time, but did not control aliased components (there are now other subrules to that effect). Change:

```
check declarations (aliased);
```

to:

```
check declarations (aliased_constant, aliased_variable);
```

C.5.3 Default_Parameter

The <place> is no more allowed to be "all", because it was ambiguous with the "all <name>" syntax of <entity>. If you used "all", duplicate the control with "calls" and "instantiations". Change:

```
My_label : check default_parameter (all, ...);
```

to:

```
My_label : check default_parameter (calls, ...),
           check default_parameter (instantiations, ...);
```

C.5.4 Improper_Initialization

By default, variables declared directly within (generic) package specifications and bodies are no more checked. To get the previous behaviour, add the "package" modifier. Change:

```
check improper_initialization (variable);
```

to:

```
check improper_initialization (package variable);
```

C.6 Migrating from 1.8r8

C.6.1 CSV(X) format

If the output format is CSV or CSVX, the file name, line number and column number are generated as three different spreadsheet columns, instead of forming a single message. This makes it easier to use a spreadsheet program for per-file statistics.

C.6.2 Default_Parameter

Due to the introduction of the "positional" keyword, "not used" is now spelled "not_used". Change:

```
check default_parameter (proc, param, not used);
```

to:

```
check default_parameter (proc, param, not_used);
```

C.6.3 Other_Dependencies

This rule has been changed into a subrule of the (new) rule "Dependencies". Change:

```
check Other_Dependencies (pack1, pack2);
```

to:

```
check Dependencies (others, pack1, pack2);
```

C.6.4 Special_Comments

Due to the introduction of another subrule, add "pattern" as the first parameter to the rule. Change:

```
check Special_Comments ("TBSL");
```

to:

```
check Special_Comments (pattern, "TBSL");
```

C.6.5 Statements

The "raise" subrule now reports all occurrences of the **raise** statement, even if another control is applicable to the same statement.

The "reraise" subrule now reports calls to `Ada.Exceptions.Reraise_Occurrence`.

The "raise_standard" subrule now reports exceptions raised by calls to `Ada.Exceptions.Raise_Exception`.

C.7 Migrating from 1.7r9

C.7.1 Case_Statement

This rule now allows the specification of both min and max values for each subrule. Subrule names have been changed accordingly. Change:

```
check Case_Statement (max_range_span, 5);
check Case_Statement (max_values, 10);
check Case_Statement (min_others_span, 4);
check Case_Statement (min_paths, 6);
```

to:

```
check Case_Statement (range_span, max 5);
check Case_Statement (values, max 10);
check Case_Statement (others_span, min 4);
check Case_Statement (paths, min 6);
```

C.7.2 Max_Parameters

This rule has been changed into a subrule of the (new) rule "Parameter_Declarations". Change:

```
check Max_Parameters (10);
```

to:

```
check Parameter_Declarations (Max_Parameters, 10);
```

C.8 Migrating from 1.6r8

C.8.1 "message" command

The message is now syntactically a string, and must always be enclosed in double quotes (quotes were optional in previous versions).

C.8.2 "source" command

If a "source" command is given in a rules file, and the sourced file is given with a relative path, it is interpreted relatively to the sourcing file (it was interpreted relatively to the current directory previously). This should make "chained" sourcing easier, since the interpretation does not depend on where the sourcing file is being called from.

C.8.3 Control_Characters

This rule is now called "Characters" and can process other kinds of characters in addition to control characters. Control characters correspond to the "control" parameter of the rule. Change:

```
check control_characters;
```

to:

```
check characters (control);
```

C.8.4 If_For_Case

This rule has been changed into a subrule of the (new) rule "simplifiable_statements". Change:

```
check if_for_case;
```

to:

```
check simplifiable_statements (if_for_case);
```

C.8.5 Instantiations

The rule does not print the number of instantiations any more, since the same effect can be achieved with the "count" control kind.

C.8.6 Local_Instantiation

This rule has been removed, since its effect can now be achieved with other rules: the rule "declarations" to check for local instantiations of any generic, and the rule "instantiations" to check for local instantiations of specified generics. Change:

```
R1: check Local_Instantiation;
R2: search Local_Instantiation (Ada.Unchecked_Conversion);
```

to:

```
R1: check  declarations   (local instantiation);
R2: search Instantiations (local Ada.Unchecked_Conversion);
```

C.8.7 Naming_Convention

Quotes are no more optional around patterns.

The <location> modifier is now before the <filter_kind> (it was before the pattern previously). This may require splitting the rule in two in some cases. For example, change:

```
check naming_convention (object, local "^L_", global "^G_");
```

to:

```
check naming_convention (local object, "^L_");
check naming_convention (global object, "^G_");
```

C.8.8 No_Safe_Initialization

The name of this rule has been changed to "improper_initialization", since it now controls other cases of improper initialization.

C.8.9 Special_Comments

Quotes are no more optional around patterns.

C.8.10 Statements

Two subrules of this rule have migrated to the new rule "simplifiable_statements" (with slightly different names). Change:

```
check statements (unnecessary_null);
check statements (while_true);
```

to:

```
check simplifiable_statements (null);
check simplifiable_statements (loop);
```

C.9 Migrating from 1.5r24

C.9.1 Declarations

The subrule "Formal_In_Out" has been renamed as "In_Out_Generic_Parameter", for consistency with the new "In_Out_Parameter" subrule.

The subrules "renames" and "not_operator_renames" have been renamed to "renaming" and "not_operator_renaming".

As a consequence of being able to specify the location of any construct, the subrules "nested_function_instantiation", "nested_generic_function", "nested_generic_package", "nested_generic_procedure", "nested_package", "nested_package_instantiation", and "nested_procedure_instantiation" have been removed and replaced with the corresponding general construct (without "nested_"). You can have the same effect by specifying the "nested" modifier in front of them. I.e., change:

```
check declarations (nested_generic_function);
```

to:

```
check declarations (nested generic_function);
```

C.9.2 Naming_Convention

The <location> keyword is placed before the <Filter_Kind> keyword instead of before the <Pattern>, which looks more natural. The "Any" keyword has been removed, since omitting the <location> keyword has the same effect. Change:

```
check naming_convention (variable, global "^G_");
check naming_convention (package, any "^Pack_");
```

to:

```
check naming_convention (global variable, "^G_");
check naming_convention (package, "^Pack_");
```

C.9.3 Non_Static_Constraint

This rule is now called Non_Static, since it is no more restricted to constraints. The parameters "index" and "discriminant" have been changed to "index_constraint" and "discriminant_constraint", respectively. Change:

```
check non_static_constraint (index, discriminant);
```

to:

```
check non_static (index_constraint, discriminant_constraint);
```

C.9.4 Positional_Parameters

This rule has been renamed to `Insufficient_Parameters`, since it does no more handle the "maximum" subrule. Controlling positional parameters according to their number is now done by the rule `style (positional_association)`. Change:

```
check positional_parameters (maximum, 3);
check positional_parameters (insufficient, 2, Boolean);
```

to:

```
check style (positional_association, call, 3);
check insufficient_parameters (2, Boolean);
```

C.9.5 Real_Operator

This rule is no more a rule of its own, it is a subrule of the (new) rule Expressions, whose name is Real_Equality. Change:

```
check Real_Operators;
```

to:

```
check expressions (Real_Equality);
```

C.9.6 Style

The name of the subrule "casing" has been changed to "casing_identifier" since the casing of attributes and pragmas can now also be checked. The casing style is no more optional.

The name of the subrule "literal" has been changed to "numeric_literal" (since characters and strings are also literals, but are not handled by this subrule).

The subrule "exposed_literal" now requires an extra parameter to tell whether it applies to integer literals, real literals, character literals or string literals. Allowed values are provided after this parameter, and must of course be of the appropriate type. In short, if you had:

```
check style (exposed_literal, 0, 1, 0.0, 1.0);
```

you must change it to:

```
check style (exposed_literal, integer, 0, 1)
check style (exposed_literal, real, 0.0, 1.0);
```

The "aggregate" parameter of the subrule "positional_association" has been split into "array_aggregate" and "record_aggregate". For example, change:

```
check style (positional_association, aggregate);
```

into:

```
check style (positional_association, record_aggregate, array_aggregate);
```

C.10 Migrating from 1.4r20

C.10.1 GPS integration

The XML file used to describe AdaControl features to GPS used to be called `adactl.xml`. It is now called `zadactl.xml`, since GPS processes its initialization files in alphabetical order. This avoids shuffling the menus when AdaControl support is activated.

Make sure to remove the old `adactl.xml` file from the GPS plug-ins directory before installing the new version.

C.10.2 Declarations

The parameters "access" and "access_subprogram" have been changed to "access_type" and "access_subprogram_type", for consistency with the new parameters.

C.10.3 Header_Comments

A keyword has been added to specify the required number of comment lines. Change:

```
check Header_Comments (10);
```

to:

```
check Header_Comments (minimum, 10);
```

C.10.4 No_Closing_Name

This rule is now part of the "style" rule. Change:

```
check|search|count No_Closing_Name;
```

to:

```
check|search|count Style (No_Closing_Name);
```

C.10.5 Specification_Objects

This rule is now part of the "usage" rule. Change:

```
check|search|count Specification_Objects (<parameters>);
```

to:

```
check|search|count Usage (Object, From_Spec, <parameters>);
```

C.10.6 Statement

Name changed from "statement" to "statements" (added an 's'), to be consistent with other rules.

C.10.7 When_Others_Null

This rule is now part of the "statements" rule. Change:

```
check|search|count When_Others_Null (case);
check|search|count When_Others_Null (exception);
```

to:

```
check|search|count Statements (case_others_null);
check|search|count Statements (exception_others_null);
```