AdaControl User Guide

Last edited: 6 December 2006

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.

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, 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 on the Eurocontrol software. The software was developped by Arnaud Lecanu and Jean-Pierre Rosen (Adalog). Some rules were contributed by Richard Toy (Eurocontrol-Maastricht), Pierre-Louis Escouflaire (Adalog), and Alain Fontaine (ABF consulting).

Commercial support is available for AdaControl, see file `doc/support.txt`. If you plan to use AdaControl for industrial projects, or if you want it to be customized or extended to match your own needs, please contact Adalog at info@adalog.fr.

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 was 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 Chapter 6 [Non upward-compatible changes], page 75 for details.

2 Installation

AdaControl is distributed only as source. 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. Given the current proliferation of various versions of GNAT, it seems better to let the user compile AdaControl himself, thus making sure that there is no mismatch.

Another reason for distributing AdaControl as source 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 Section 2.4 [Customizing AdaControl], page 4.

2.1 Prerequisites

The following software must be installed in order to install AdaControl:

- A GNAT compiler, any version. Note that the compiler is also required to use 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.

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 3.8.4 [unrepr.sed], page 23.

2.2 Building AdaControl

2.2.1 Build with project file

Simply go to the `src` directory and type:

```
gnatmake -Pbuild.gpr
```

You're done!

2.2.2 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`.

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.

2.2.3 Installing support form GPS

To add AdaControl support to GPS, simply copy all the files from the `GPS` directory into the `<GPS_dir>/share/gps/plug-ins` directory. Copy also the files `doc/adacontrol_ug.html` and `doc/adacontrol_pm.html` into the `<GPS_dir>/share/doc/gps/html` to access AdaControl’s guides from the "Help" menu of GPS.

2.3 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 because you are using an old version of Gnat (and especially 3.15p). AdaControl runs without any known problem (and it has been checked against the whole ACATS) only with the latest GnatPro version; 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.

2.4 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.

3 Program Usage

3.1 Running AdaControl from the command line

AdaControl is a command-line program, i.e. it's callable directly by a system shell, and can be integrated in GUIs such as GPS (see Section 3.2 [Running AdaControl from GPS], page 6) or emacs (see Section 3.3.1 [Rule types and report messages], page 9). It is very simple to use. It takes, as parameters, a list of units to process and a set of rules to apply. AdaControl produces error and/or found messages to the standard output. The type of message (i.e. error or found) depends on the type of the rule (i.e. check or search). It is also possible to locally disable rules 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
```

Caveat:

If your project includes source files located in several directories, the ADA_INCLUDE_PATH environment variable is not always considered by ASIS, resulting in error messages that tell you that the bodies of some units have not been found (and hence not 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 3.5.13 [ASIS options], page 21), or generate the tree files manually (see Section 3.9.2 [Generating tree files manually], page 24). Note that this problem does not happen if you are using Emacs project files (see Section 3.5.12 [Project files], page 21), nor if you are running AdaControl from GPS.

AdaControl can process only Ada-95, not Ada-2005, since there no ASIS for Ada-2005 yet. If 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 must force Ada-95 mode by having a "gnat.adc" file that contains a **pragma** `Ada_95`, since the corresponding option cannot be passed to the compiler in "compile on the fly" mode. Alternatively, you can generate the tree files manually (see Section 3.9.2 [Generating tree files manually], page 24) with the "-gnat95" option.

3.2 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 [Building AdaControl], page 3.

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.

3.2.1 The AdaControl menu and button

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.
- “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).
- “Create .adp project” create 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 3.5.12 [Project files], page 21.

There is also a button representing Lady Ada in a magnifier glass in the toolbar; clicking this button is the same as selecting “Control Current File (rules file)”.

Here are some tips about using the “interactive” menus:

- 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.

3.2.2 AdaControl switches

The tab "switches" from the "Project/Edit Project Properties" menu includes a page for AdaControl, which allows you to set various parameters.

- "Recursive mode". This sets the "-r" option. See Section 3.5.3 [Input units], page 17.
- "Ignore local deactivation". This sets the "-r" option. See Section 3.5.8 [Local deactivation ignoring], page 20.
- "Process specs only". This sets the "-s" option. See Section 3.5.3 [Input units], page 17.
- "Compilation unit mode". This sets the "-u" option. See Section 3.5.3 [Input units], page 17.
- "Display only errors". This sets the "-E" option. See Section 3.5.10 [Treatment of warnings], page 20.
- "Warnings as errors". This sets the "-e" option. See Section 3.5.10 [Treatment of warnings], page 20.
- "Statistics". This sets the "-S" option from a pull-down menu. See Section 3.3.1 [Rule types and report messages], page 9.
- "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.
- "File name". This is the name of the file that will contain the results when the previous button is checked. If the file exists, AdaControl will ask for the permission to override it.
- "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 3.3.1 [Rule types and report messages], page 9.
- "Debug messages". This sets the "-d" option. See Section 3.5.9 [Verbose and debug mode], page 20.
- "Halt on error". This sets the "-x" option. See Section 3.5.11 [Exit on error], page 20.

Since the GPS interface analyzes the output of AdaControl, you should not set options directly in the bottom window of this page.

3.2.3 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, the browser does not find the right anchor; hopefully, this will be fixed in an upcomming release of GPS).
- “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.

3.2.4 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.

3.2.5 AdaControl help

The AdaControl User Manual (this manual) and the AdaControl Programmer Manual are available from the "Help/AdaControl" menu of GPS. In addition, there is a "Help on rule" entry in this menu. This entry displays the list of all rules; if you click on one of them, it displays the rule(s) purpose and the syntax of its parameters.

3.3 Rules syntax

AdaControl is about *checking rules*. Each rule has a name, and may require parameters. Which rules are to be checked is specified either on the command line or in a rules file; in either case, the syntax for specifying rules is as follows:

```
[<label> ":"] "check"|"search"|"count" <Name>
    ["(" [<modifiers>] <parameter> {"," [<modifiers>] <parameter>}")"] ";"
```

If present, the label gives a name to the rule; it will be printed whenever the rule is activated, and can be used to disable the rule. See Section 3.7 [Disabling rules], page 22. 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. Note that there is no problem in specifying the same label for several rules.

Each rule consists of a rule type 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. The case of the rule type, rule name, and parameters is not significant. If a syntax error is encountered in a rule, 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.

Since wide characters are allowed in Ada programs, AdaControl accepts wide characters in rules 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 Chapter 7 [Syntax of regular expressions], page 78.

3.3.1 Rule types and report messages

There are three rule types:

- check

- search
- count

"Check" is intended to search for rules that must be obeyed in your programs. Normally, if a "Check" rule 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 rule is an error, and "search" when you consider it as a warning. AdaControl will exit with a status of 1 if any "Check" rule is triggered, and a status of 0 if only "Search" rule were triggered (or no rule was triggered at all).

"Count" works like "search", but instead of printing a message for each rule which is triggered, it simply counts occurrences and prints a summary at the end of the run. There is a separate count for each rule label (or if no label is given, the rule name is taken instead); if you give the same label to different rules, 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 rule matches)
- the line number (where the rule matches)
- the column number (where the rule matches)
- the rule label (if there is one) and/or the rule id (the rule that matches).
- a message (why the rule matches). A rule whose type is "check" will produce an error report message (i.e. containing the keyword ERROR) and a rule use whose type 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 rule label appears, unless there is none, in which case it is replaced with the rule id.

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), which uses the semicolon and not the comma to separate fields. Therefore, the formats "CSVX" and "CSVX_Short" do the same thing, but using semicolons (';') instead of commas. Both the rule label (replaced by an empty column if there is none) and the rule id appear.

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.

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 "go" command, 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 rule *not* triggered are output
- 3: The rule name and label (if any) of every rule is output, together with a count of each triggering type (check, search, count), or "not triggered" if the rule was not triggered.

3.3.2 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 are allowed as in Ada). Based literals are not currently supported; if somebody can justify a need for them, we'll be happy to add this feature later...

A character string (actually, any parameter whose value does not follow the rules of Ada identifiers or numeric literals) is given within double quotes "`"`". The tilde character ("`~`") can also be used as a replacement, 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.

An Ada entity name can be followed by overloading information (see below), 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.

3.3.3 Specifying an Ada entity name

The syntax of the <Ada_Entity_Name> is as follows:

```
<Ada_Entity_Name> ::= <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.

3.3.3.1 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 <Ada_Entity_Name>, 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 <Ada_Entity_Name> 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 <Ada_Entity_Name>. 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 <Ada_Entity_Name>, 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}`).

3.3.3.2 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}`

3.3.3.3 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}`.

3.3.3.4 Attributes

It is also possible to designate attributes, 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).

3.3.3.5 Anonymous constructs

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_`. So 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`.

3.3.3.6 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`.

3.3.3.7 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`.

3.3.4 Multiple rules

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

The following configuration 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 rules, but a disabling comment that mentions `No_Elaborate` will disable only the second one.

3.4 Commands

In addition to rules specification, AdaControl recognizes a number of commands. Although these commands are especially useful when using the interactive mode (see Section 3.5.7 [Interactive mode], page 19), they can be used in command files as well.

3.4.1 Go command

Syntax:

```
go;
```

This command starts processing of the rules that have been specified. Rules 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 3.4.5 [Clear command], page 15 to reset rules.

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

3.4.2 Quit command

Syntax:

```
quit;
```

This command terminates AdaControl. If given in a file, all subsequent commands will be ignored. This command is really useful only in interactive mode. See Section 3.5.7 [Interactive mode], page 19.

3.4.3 Message command

Syntax:

```
message <any string>;
```

This command prints the given message on the output file. The length of the message is limited to 250 characters.

Note that the message is terminated by the first ";" encountered. If a message needs to include a ";", the hole message must be quoted (double quotes).

3.4.4 Help command

Syntax:

```
Help [ all | <rule name>{,<rule name>} ];
```

Without any argument, this command prints a summary of all commands and rule names. If given one or more rule names, it prints the detailed help for the given rules. If given the keyword `all`, it prints the detailed help for all rules.

3.4.5 Clear command

Syntax:

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

This command clears all "count", "search", and "check" commands given for the indicated rules, of for all rules if the `all` keyword is given. 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`.

3.4.6 Set command

Syntax:

```
set Format Gnat | Gnat_Short | CSV | CSV_Short | source | source_short
set Output <output file>;
set Statistics <level>
set Trace <trace file>;
set Verbose | Debug | Ignore | Warning  On | Off
```

In the first form, this commands selects the output format for the messages, like the "-F" option; see Section 3.3.1 [Rule types and report messages], page 9 for details.

In the second form, this command redirects the output of subsequent checks to the indicated file. If the string `console` (case irrelevant) is given as the <output file>, output is redirected to the console.

As with the "-o" option, if the file exists, output is appended to it, 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.

In the third form, this 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.

In the fourth form, this command allows to set the statistics level, like the "-S" option; see Section 3.3.1 [Rule types and report messages], page 9 for details.

In the fifth form, this command allows to activate ("on") or deactivate ("off") options. "Verbose" corresponds to the "-v" option, "Debug" to the "-d" option, "Ignore" to the "-i" option, and "Warning" to the "-E" option. See Section 3.5.9 [Verbose and debug mode], page 20, Section 3.5.10 [Treatment of warnings], page 20, and Section 3.5.8 [Local deactivation ignoring], page 20 for details.

3.4.7 Source command

Syntax:

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

This command redirects the input of commands from the indicated file. Commands and rules 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 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.

3.4.8 Inhibit command

Syntax:

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

This command will inhibit execution of the rule (or all rules if “all” is specified in place of a rule name) for the indicated unit(s). In addition, if “all” is given 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 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, 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 is intended for the last case, to prevent ASIS bugs from spoiling any unit that calls something from an offending unit.

3.4.9 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 attribute (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.

3.5 Command line options and parameters

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 complete syntax for invoking AdaControl is:

```
adactl [-deEiIrsuvw] [-f <rules file>] [-l <rules list>] [-o <output file>]
       [-F <format>] [-p <project file>] [-S <statistics level>]
       {<unit>[+|-<unit>]|[@]<file>} [-- <ASIS options>]
```

or

```
adactl -h [<rule id>... | all]
```

or

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

or

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

Using AdaControl with the “-D” option is described later. See Section 3.8 [Helpful utilities], page 22.

3.5.1 Getting help

The “-h” option alone displays a help message about usage of the AdaControl program, the various options, and the rule names. If the “-h” is followed by one or several rule names (case irrelevant), it displays the help message for the rule(s). If the “-h” option is followed by the keyword “all”, it displays the help message for all rules. If the “-h” option is followed by the keyword “list”, it simply lists the names of all rules (note that “-h” without parameters also displays the list of rules, in a prettier format; this option is mainly useful for the integration of AdaControl into GPS).

Ex:

```
adactl -h
adactl -h pragmas Unnecessary_Use_Clause
adactl -h all
```

Note that if the “-h” option is given, no other option is analyzed and no further processing happens.

3.5.2 Checking rules syntax

If the “-C” option is given, AdaControl will simply check the syntax of the rule provided with the “-l” option, or of the rules 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 Input units

Units to be processed are simply 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! Of course, 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. All 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" 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 (the "-r" (recursive) option would find them otherwise).

Alternatively, it is possible to give a parameter as an "`@`" followed by the name of a file. This file must contain a list of unit names (not files), one on each line. 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
```

3.5.4 Specifying rules

Rules list can be passed on the command line using the "-l" option. Rules list must be quoted with "`"`".

Ex:

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

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

Rules list can also be passed from a file, whose name must be given after the "-f" option. As a special case, if the file name is "-", rules 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 rules directly to AdaControl, the interactive mode is more appropriate. See Section 3.5.7 [Interactive mode], page 19.

Ex:

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

A rule file must contain at least one rule. The layout of rules is free (i.e. a rule can extend over several lines, and spaces are allowed between syntactic elements). A rule file may also contain comment lines. Comments begin with a "#" or a "--", and extend to the end of the line. Comments can be placed anywhere in the file.

Ex:

```
# My rules file
# generated by myself 2004.09.27.14.12.36
search rule1 (param1, param2, param3);   -- This is Rule 1
My_Label: check rule2 (param1);
search rule3 (param1,
-- Comment in the middle
              param2,
              param3, param4);
search rule4;   -- A rule without parameters
```

Note that the "-l" and "-f" options are *not* exclusive: if both are specified, the rules to be checked include those in the file and those given on the command line.

3.5.5 Output file

By default, the standard output is used for output. The default output can be changed by specifying an output file with the "-o" option.

Ex:

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

Error and found rule messages are output to the output file. Syntax error messages for rules and possible internal errors from AdaControl itself are output to the standard error file.

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.

Ex:

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

3.5.6 Output format

The "-F" option selects the output format. It must be followed by "Gnat", "Gnat_Short", "CSV", or "CSV_Short" (case insensitive). By default, the output is in "Gnat" format. See Section 3.3.1 [Rule types and report messages], page 9 for details.

The "-S" option selects which statistics are output. It must be followed by a value in the range 0..3. See Section 3.3.1 [Rule types and report messages], page 9 for details on the various statistics levels.

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

3.5.7 Interactive mode

The "-I" option tells AdaControl to operate interactively. In this mode, commands and rules specified with "-l" or "-f" options are first processed, then AdaControl prompts for commands on the terminal. Note that the "quit" command (see Section 3.4.2 [Quit command], page 14) is used to terminate AdaControl.

The syntax for rules and commands is exactly the same as the one used for files; especially, each rule or 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. Note that if a command contains syntax errors, all “go” commands 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 check. 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.

3.5.8 Local deactivation ignoring

The “-i” option tells AdaControl to ignore deactivation tags in Ada source code (see Section 3.7 [Disabling rules], page 22).

Ex:

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

3.5.9 Verbose and debug mode

In the default mode, AdaControl displays only rule messages. It is possible to get more information with the verbose option (“-v”). In this mode, AdaControl displays unit names as they are processed, and prints the number of errors, the number of warnings, and its global execution time when it finishes.

Ex:

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

It is also possible to get more information in case of a program error by using the debug mode. Debug mode is enabled by using the “-d” option.

Note that in this mode, AdaControl may, 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 handled by AdaControl.

Ex:

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

In addition, output of the messages printed by the “-d” option can be directed to a file (instead of being printed on 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.

3.5.10 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” rules were triggered. See Section 3.6 [Return codes], page 21. 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 3.3.1 [Rule types and report messages], page 9.

3.5.11 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 25.

Ex:

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

3.5.12 Project files

3.5.12.1 Emacs style 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
```

3.5.12.2 GPS project files

When run from GPS, AdaControl will automatically use the source directories from the current (root) project. However, if you run it from the command line, it will not accept “.gpr” project files, because ASIS does not currently accept the “-P” option like other Gnat commands do. Should this change in the future, a “-P” option could be passed as described for the “-I” option. See Section 3.5.13 [ASIS options], page 21.

In the mean time, 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 3.2 [Running AdaControl from GPS], page 6. 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 24 for details.

3.5.13 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 (the “-p” option), by passing a “-I” option to ASIS, or by putting the appropriate directories into the ADA_INCLUDE_PATH environment variable.

It is possible to include one or several “-I” options to reference 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.

3.6 Return codes

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

- 0: At most “search” rules were triggered (no rule at all with “-e” option)
- 1: At least one “check” rule was triggered (or at least one “search” or “check” rule 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.

3.7 Disabling rules

It is possible to disable rules on parts of the source code by placing a tag (special Ada comment) in the source code. This can be done in two ways: block disabling or line disabling. The disabling tag is "--##". Both ways take a list of rules to disable as parameters. A list of rules is a list of rule names or rule labels, separated by spaces. Alternatively, the list of rules can be the word "all" to disable all rules.

In a "–##" line, everything appearing after a second occurrence of "##" is ignored. This allows the insertion of a comment explaining why the rule is disabled at that point.

3.7.1 Block disabling

A rule is disabled from the "rule off" tag until the "rule on" tag. If there is no "rule on" tag, the rule is disabled up to the end of file.

Syntax:

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

Ex:

```
--## rule off rule1 rule2
I := I + 1;
Proc (I);
--## rule on rule2
```

3.7.2 Line disabling

The rule is disabled only for the line where the tag 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 rule for just the current line in a block where rules are disabled:

Syntax:

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

Ex:

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

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 Section 3.3.3 [Specifying an Ada entity name], page 11) 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 [-sofd] [-p <project-file>] <unit>[:<line_number>[:<column_number>]]
     [-- <ASIS options>]
```

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 given 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). If a <line_number> is given, only identifiers on that line are printed. If both <line_number> and <column_number> are given, only the identifier (if any) at the given line and column is printed. The image is printed without overloading information, unless the "-o" option is given.

If the "-s" option is given, 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.

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 Adactl -D

When run with the "-D" option, AdaControl simply outputs the list of units that would be processed.

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 3.5.3 [Input units], page 17. Note that if you use the recursive ("-r") option, it is more efficient to create the list of units once and then use the indirect file than to specify all applicable units each time AdaControl is run.

3.8.3 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 4.15 [Header_Comments], page 40.

3.8.4 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 rules, 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;
- 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 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).

Note also that the "-r" option (recursive mode) of AdaControl implies an extra pass over the whole program tree to determine the necessary units.

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.
- 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

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 3.4.8 [Inhibit command], page 16.

If you encounter a problem while using AdaControl, you are very welcome to report it to rosen@adalog.fr. Please include the exact rule and the unit that caused the problem, as well as the captured output of the program (with “-d” option).

4 Rules Usage

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.

4.1 Abnormal_Function_Return

4.1.1 Syntax

```
<check|search|count> abnormal_function_return;
```

4.1.2 Action

This rule controls that the sequence of statements of each function body, as well as each exception handler, ends either with a **return** statement or a **raise** statement (or equivalently, a call to `Ada.Exceptions.Raise_Exception` or `Ada.Exceptions.Reraise_Occurrence`). Note that this last statement can be embedded in blocks (i.e., it can be followed by any number of **end** for block statements, but nothing else).

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
```

4.1.3 tip

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 4.43 [Statements], page 58.

4.2 Allocators

4.2.1 Syntax

```
<check|search|count> allocators
   [(task|protected|<type name> {, task|protected|<type name>})];
```

4.2.2 Action

This rule controls usage of allocators. If type names are given, only allocators whose allocated type is mentioned are controlled; if "task" or "protected" is given, allocators for task types or protected types (respectively) are controlled; otherwise all allocators are controlled. This rule is especially useful for finding memory leaks, since it tells all the places where dynamic allocation occurs.

Ex:

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

4.2.3 Tips

The type given in the rule is the first named subtype, and the rule will also find allocators that use a subtype of this type; especially, if the allocated type is `T'Base`, it will be found as T.

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`.

4.3 Array_Declarations

4.3.1 Syntax

```
<check|search|count> Array_Declarations (First, <value>);
<check|search|count> Array_Declarations (Max_Length, <maximum_length>);
```

4.3.2 Action

This rule controls various properties of array types and array objects declarations, depending on the keyword given as the first parameter:

- "First" controls the lower bound of each dimension of arrays (even unconstrained array types) whose value is not the given value. If this subrule is given both for "search" and for "check", the value for "search" is interpreted as the prefered one, and the value 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, 0);
  ```

  will issue a warning if the lower bound of an array is 0, and an error if it is neither 0 or 1.
- "Max_Length" controls arrays that have a dimension whose number of elements is greater than the given value, except for unconstrained array types.

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 array_declarations (first, 1);
check array_declarations (max_length, 100);
```

4.4 Barrier_Expressions

4.4.1 Syntax

```
<check|search|count> 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
```

4.4.2 Action

This rule controls expressions used in barriers of protected entries. 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.
- "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 membship 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);
```

4.4.3 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 subset of those allowed for "check". This way, the use of certain features can be interpreted only as a warning.

4.5 Case_Statement

4.5.1 Syntax

```
<check|search|count> Case_Statement (max_range_span, <maximum_span>);
<check|search|count> Case_Statement (max_values, <maximum_span>);
<check|search|count> Case_Statement (min_others_span, <minimum_span>);
<check|search|count> Case_Statement (min_paths, <minimum_span>);
```

4.5.2 Action

This rule controls various sizings in case statement, depending on the keyword given as the first parameter:

- "max_range_span" controls that ranges used as choices in **case** statements cover at most the specified number of values. Especially, a value of 0 disallows all ranges as choices.
- "max_values" controls **case** statements where the subtype of the case selector covers more values than the specified number of values.
- "min_others_span" controls **when others** case alternatives that cover less than the specified number of values. The <minimum_span> must be at least 1 (i.e., if 1 is specified, the rule will signal "when others" that cover no value at all).
- "min_paths" controls **case** statements with less paths (i.e. **when** branches) than the specified number of values.

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 (min_others_range, 1);
search Case_Statement (min_others_range, 5);

check  Case_Statement (max_values, 10);
check  Case_Statement (min_paths, 5);
```

4.5.3 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 4.46 [Uncheckable], page 63.

4.6 Control_Characters

4.6.1 Syntax

```
<check|search|count> control_characters;
```

4.6.2 Action

This rule controls the occurrence in the source file of the control characters that are allowed by the language (ASCII HT, ASCII VT and ASCII FF). Since it has no parameters, this rule can be given only once.

Ex:

```
check control_characters;
```

4.7 Declarations

4.7.1 Syntax

```
<check|search|count> declarations (<declaration_kw> {, <declaration_kw>});

declaration_kw ::=
   access_protected_type          | access_subprogram_type         |
   access_task_type               | access_type                    |
   aliased                        | array                          |
   array_type                     | child_unit                     |
   constant                       | constrained_array_type         |
   decimal_fixed_type             | defaulted_discriminant         |
   defaulted_generic_parameter    | defaulted_parameter            |
   derived_type                   | discriminant                   |
   enumeration_type               | entry                          |
   exception                      | extension                      |
   fixed_type                     | float_type                     |
   formal_function                | formal_package                 |
   formal_procedure               | generic                        |
   handlers                       | in_out_generic_parameter       |
   in_out_parameter               | integer_type                   |
   initialized_protected_field    | initialized_record_field       |
   limited_private_type           | modular_type                   |
   multiple_names                 | named_number                   |
   nested_function_instantiation  | nested_generic_function        |
   nested_generic_package         | nested_generic_procedure       |
   nested_package                 | nested_package_instantiation   |
   nested_procedure_instantiation | non_limited_private_type       |
   non_identical_renaming         | not_operator_renaming          |
   null_extension                 | null_ordinary_record_type      |
   null_tagged_type               | operator                       |
   operator_renaming              | ordinary_fixed_type            |
   ordinary_record_type           | out_parameter                  |
   package_statements             | private_extension              |
   protected                      | protected_entry                |
   protected_type                 | record_type                    |
   renaming                       | separate                       |
   signed_type                    | single_array                   |
   single_protected               | single_task                    |
   subtype                        | tagged_type                    |
   task                           | task_entry                     |
   task_type                      | type                           |
   unconstrained_array_type       | uninitialized_protected_field |
   uninitialized_record_field
```

4.7.2 action

This rule controls usage of certain Ada declarations. The rule can be specified at most once for each declaration keyword.

- Declaration keywords that are Ada keywords match the corresponding Ada declarations.
- `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.

- 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.
- child_unit controls the declaration of all child units.
- 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 defaulted_discriminants controls only those where defaults are provided for the discriminants.
- exception controls exception declarations.
- fixed_type controls all declarations of fixed point types while ordinary_fixed_type controls only ordinary (binary) fixed point types, and decimal_fixed_type controls only decimal fixed point types.
- float_type controls declarations of floating point types.
- formal_function, formal_package, and formal_procedure control generic formal functions, packages, and procedures, respectively.
- 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
- integer_type controls all declarations of integer types, while signed_type controls only signed integer types, and modular_type controls only modular types.
- initialized_record_field and initialized_protected_field control the declaration of record (respectively protected) component that include a default initialization, while uninitialized_record_field and uninitialized_protected_field control the declaration of record (respectively protected) component that do not include a default initialization
- limited_private_type controls limited private type declarations, while non_limited_private_type controls regular (non limited) private type declarations.
- multiple_names controls declarations where more than one defining identifier is given in the same declaration.
- named_number controls declarations of named numbers, i.e. untyped constants.
- nested_package controls package declarations that are not compilation units (i.e. nested in some other unit).
- nested_generic_function, nested_generic_package, nested_generic_procedure control generic function (respectively package, procedure) declarations that are not compilation units (i.e. nested in some other unit).
- nested_function_instantiation, nested_package_instantiation, nested_procedure_instantiation control function (respectively package, procedure) instantiations that are not compilation units (i.e. nested in some other unit).
- 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.

- `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.
- `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.
- `renaming` controls all renaming declarations, while `operator_renaming` controls only those that are renamings of an operator, `not_operator_renaming` controls only those that are not renamings of an operator, and `non_identical_renaming` controls only those where the new name and the old name are not the same.
- `subtype` control all explicit subtype declarations (i.e. not all anonymous subtypes that appear at various places in the language).
- `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`).
- `type` controls all type (but not subtype) declarations.

Ex:

```
search declarations (task, exception);
```

4.7.3 Tips

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

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

regular record types will be reported as "record_type", while tagged types will be reported as "tagged_type" (but not both). However, if several keywords apply for *different* rule types, like:

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

then both are reported (for a tagged type declaration).

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 AdaControl can be used not only for checking, but also for searching; finding all type declarations in a set of units can make sense.

4.7.4 Limitation

It is currently not possible to specify different rule types for the same declaration keyword; especially, it is not possible to specify both `search` (or `check`) and `count` for the same declaration keyword. However, it is possible to specify different rule types for *different* declaration keywords, even if they overlap. For example, the following will report all task entries, and count all entries (whether task entries or protected entries):

```
search declarations (task_entry);
count  declarations (entry);
```

This limitation is expected to be removed in the next version of AdaControl.

4.8 Default_Parameter

4.8.1 Syntax

```
<check|search|count> default_parameter
   (<entity> | all, <formal name> | all, [not] used);
```

4.8.2 Action

This rule controls subprogram calls or generic instantiations that use (or conversely don't use) the default value for the indicated parameter. 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 `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).
- If the string given as the third parameter is anything else, it is an error.

Alternatively, the <entity> and/or the <formal name> can be replaced by the keyword `all`, in which case any entity (respectively formal) will match.

Ex:

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

4.8.3 Limitations

This rule does not (yet) consider the use of default formal procedures and functions in generic instantiations.

4.9 Directly_Accessed_Globals

4.9.1 Syntax

```
<check|search|count> Directly_Accessed_Globals [(<kind_kw> {,<kind_kw>})];
kind_kw ::= plain | accept | protected
```

4.9.2 Action

This 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, this 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
```

4.9.3 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 4.50 [Usage], page 67 for details.

4.9.4 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 4.46 [Uncheckable], page 63.

4.10 Entities

4.10.1 Syntax

```
<check|search|count> entities (<name> {, <name>});
```

4.10.2 Action

This rule controls all uses of the indicated entities. It 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.

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 or part of a generic, all corresponding uses in 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`.

4.10.3 Tips

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);
```

In certain contexts, only a limited set of the Ada predefined units is allowed. For example, it can be useful to forbid entities from `Standard`, `System`, or entities defined in special needs annexes. The `rules` directory of Adacontrol contains files with Entity 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 5.1 [Rules files provided with AdaControl], page 71.

4.10.4 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.

4.11 Entity_Inside_Exception

4.11.1 Syntax

```
<check|search|count> entity_inside_exception (<spec> {, <spec>});
<spec> ::= [not] <entity> | calls
```

4.11.2 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 an <entity> (or "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 handler). 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:

```
check entity_inside_exception (ada.text_io.put_line);

-- Control all calls, except to the Report_Exception procedure:
check entity_inside_exception (calls, not Reports.Report_Exception);

-- Control all Put, except the one on Strings:
check entity_inside_exception (all Put,
                               not Ada.Text_IO.Put{Standard.String});
```

4.12 Exception_Propagation

4.12.1 Syntax

```
<check|search|count> exception_propagation
   ([<level>,] interface, <convention> {, <convention> });
<check|search|count> exception_propagation
   ([<level>,] parameter, <parameter name> {, <parameter name>});
<check|search|count> exception_propagation
   ([<level>,] task);
<check|search|count> exception_propagation
   (<level>, declaration);
```

4.12.2 Action

This rule controls 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 handlers 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`.

A declaration is considered propagating if it includes elements that could propagate exceptions. The strength of the check depends on the given <level>. The possible values and their effect are:

- 0: expressions in declarative parts are not considered (anything allowed, default behaviour for "interface", "parameter" and "task". Not allowed for "declaration").
- 1: no function calls (including operators) are allowed in expressions.
- 2: same as 1, plus no use of variables in expressions allowed.
- 3: same as 2, plus no declaration of objects (constants or variables) allowed (not very useful for "declaration").

It is dangerous to call an Ada subprogram that can propagate exceptions from a language that has no exception (and especially C). Therefore any such subprogram should have a "catch-all" exception handler. In its first form, the rule analyzes all subprograms to which an `Interface` or `Export` pragma applies (with the given convention(s)), and reports on those that can propagate exceptions.

Moreover, 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. In its second form, the rule is given 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 Section 3.3.3 [Specifying an Ada entity name], page 11). The rule will report on 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.

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. In its third form, the rule will report on any task that can propagate exceptions.

For these three forms, ensuring that a handler is present protects against exceptions raised in the sequence of statements, but not against exceptions raised by declarations. In addition, the (optional) `<level>` parameter can be used to control the use of certain constructs in the declarative part of subprograms or tasks, in order to minimize the possibility of exceptions being raised.

Finally, it is sometimes desirable to make sure that no declaration raises an exception, ever. In its fourth form, the rule will report on any declaration that can propagate exceptions, irrespectively of where it appears. In this case, the specification of <level> is required and cannot be 0.

Ex:

```
check exception_propagation (interface, C);
check exception_propagation (parameter, Pack.Register.CB);
check exception_propagation (task);
check exception_propagation (2, declaration);
```

The first line 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 line will report on every call like:

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

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

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

4.12.3 Tips

Note that the registration procedure 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.

4.12.4 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 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 4.46 [Uncheckable], page 63.

4.13 Expressions

4.13.1 Syntax

```
<check|search|count> expressions (<expression_kw> {, <expression_kw>});

expression_kw ::= and     | and_then      | array_others  | or    |
                  or_else | real_equality | record_others | slice |
                  xor
```

4.13.2 Action

This rule controls usage of certain forms of expressions. The rule can be specified at most once for each expression keyword.

- `real_equality` controls usage of exact equality or inequality ("=" or "/=") between real (floating point or fixed point) values.
- `slice` controls usage of array slices.
- `and`, `or`, `xor`, `and_then`, and `or_else` control usage of the corresponding logical operator (or short circuit form).
- `array_others` and `record_others` control the occurrence of a **when others =>** association in array and record aggregates, respectively.

Ex:

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

4.14 Global_References

4.14.1 Syntax

```
<check|search|count> global_references
   (all|multiple|multiple_non_atomic,
    task|protected|<Entity_name> {, task|protected|<Entity_name>});
```

4.14.2 Action

This rule controls access to global variables from several entities. The `<Entity_name>` must be subprograms, task types, single task objects, protected types, or single protected objects. The special keywords `task` and `protected` are used to refer to all tasks and to all protected entities, respectively.

If the first parameter is `all`, all references to global elements from the indicated entities are reported. If the first parameter 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 the first parameter 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 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.

4.14.3 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`.

4.15 Header_Comments

4.15.1 Syntax

```
<check|search|count> header_comments (minimum, <comment lines>);
<check|search|count> header_comments (model, "<file name>");
```

4.15.2 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 interpreted as a file name (and must be given within quotes, since usually file names contain special characters like "." and "/"). 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 the 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 Chapter 7 [Syntax of regular expressions], page 78. However, if a line contains only a single "*" character, it means that the next line is a pattern that can be matched any number of times (including 0).

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:

```
^--$
^-- Author: .+$
^-- Date: \d{2}/\d{2}/\\d{4}$
```

4.15.3 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 "\".

4.16 If_For_Case

4.16.1 Syntax

```
<check|search|count> if_for_case;
```

4.16.2 Action

This rule 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 rule 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;
```

Ex:

```
check if_for_case;
```

4.17 Instantiations

4.17.1 Syntax

```
<check|search|count> instantiations (<generic name> {, <entity name> | <>});
```

4.17.2 Action

This rule controls all instantiations of a generic, or only instantiations that are made with specific values of the parameters.

An instantiation matches if either:

1. No entity name is given in the rule
2. The entity names given are the same as the first parameters of the instantiation (i.e. there can be more actual parameters in the instantiation than specified in the rule). A box `<>` can be given instead of an entity name, in which case it will match any actual parameter.

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

Ex:

```
search instantiations (ada.unchecked_deallocation);
check instantiations (ada.unchecked_conversion, standard.string);
check instantiations (ada.unchecked_conversion, <>, standard.string);
```

The first example searches for all instantiations of `Ada.Unchecked_Deallocation`; the second one checks instantiations of `Ada.Unchecked_Conversion` where the first parameter is `String` (ignoring the second parameter), while the third example checks instantiations of `Ada.Unchecked_Conversion` where the second parameter is `String` (ignoring the first parameter).

4.17.3 Tips

It is often useful to check that a generic is instantiated only once (at least for a given type) in a project. For example, a project may have a special service in charge of releasing pointers to strings; it may be useful to check that `Unchecked_Deallocation` is not instantiated for `String` anywhere else.

Note that the report message for this rule counts how many matches are found; a first solution is to search for instantiations of `Unchecked_Deallocation` and verify manually that the count is 1.

Another solution is to disable the check for the rule at the place where it is allowed, and then do a check; if there are other instantiations, they will come out as errors.

4.18 Insufficient_Parameters

4.18.1 Syntax

```
<check|search|count> insufficient_parameters
                       (<Max_Allowed> {, <Type_Name>});
```

4.18.2 Action

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. <Max_Allowed> is the maximum number of allowed "insufficient" parameters (can be 0). <Type_Name> designates enumeration types whose values should be included in the check.

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);
```

4.18.3 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 `style (positional_association)`. See Section 4.44 [Style], page 60.

Note also that this rules applies only to calls, while `style (positional_association)` applies to all forms of associations.

4.19 Local_Hiding

4.19.1 Syntax

```
<check|search|count> local_hiding;
```

4.19.2 Action

This rule controls declarations that hide an outer declaration with the same name (and parameter and result type profile, if both are overloadable constructs). Since this rule has no parameters, it can be given only once (otherwise, it is an error).

Ex:

```
search local_hiding;
```

4.20 Local_Instantiation

4.20.1 Syntax

```
<check|search|count> local_instantiation
   [(<generic name> {, <generic name>})];
```

4.20.2 Action

This rule controls instantiations that are done in a local scope (i.e. not at library level in a library package, or a subpackage of a library package). Instantiations that appear in a generic package are not flagged (unless the generic package is itself in a local scope).

Without parameter, the rule controls all local instantiations, otherwise it controls only instantiations of the indicated generics.

Ex:

```
check local_instantiation (ada.unchecked_deallocation);
search local_instantiation;
```

4.21 Max_Blank_Lines

4.21.1 Syntax

```
<check|search|count> max_blank_lines (<max allowed blank lines>);
```

4.21.2 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);
```

4.22 Max_Call_Depth

4.22.1 Syntax

```
<check|search|count> Max_Call_Depth (<allowed depth> | finite);
```

4.22.2 Action

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

4.22.3 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.

4.22.4 Limitations

Calls to attributes, predefined operators, etc. are assumed to have a depth of 1.

Calls through pointers to subprograms and dispatching calls are unknown statically; they are assumed to have a depth of 1. Such calls are detected by the rule "uncheckable". See Section 4.46 [Uncheckable], page 63.

4.23 Max_Line_Length

4.23.1 Syntax

```
<check|search|count> max_line_length (<max allowed length>);
```

4.23.2 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);
```

4.24 Max_Nesting

4.24.1 Syntax

```
<check|search|count> max_nesting (<max allowed depth>);
```

4.24.2 Action

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. 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). Of course, this makes sense only if the level for search is less than the one for check.

Ex:

```
search max_nesting (5);
check max_nesting (7);
```

4.25 Max_Parameters

4.25.1 Syntax

```
<check|search|count> max_parameters (<max_allowed>, {,<entity_kw>});
entity_kw ::= function           | procedure           | protected_entry |
              protected_function | protected_procedure |task_entry
```

4.25.2 Action

This rule controls declarations of callable entities that have more parameters than the specified allowed value. If one or more <entity_kw> is specified, the rule applies only to the corresponding declaration(s), otherwise it applies to all callable entities.

This rule can be given once for each of check, search, and count for each kind of entity. 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_parameters (10, procedure, function);
search max_parameters (5, procedure, function);
count max_parameters (5);
```

4.25.3 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.

4.26 Max_Statement_Nesting

4.26.1 Syntax

```
<check|search|count> Max_Statement_Nesting (<stmt_kw>, <max allowed depth>);
<stmt_kw> ::= block | case | if | loop | all
```

4.26.2 Action

This rule controls the nesting of compound statements. 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);
```

4.27 Movable_Accept_Statements

4.27.1 Syntax

```
<check|search|count> movable_accept_statements
   (certain|possible [, <entity_list>])
<entity_list> ::= <entity name> {, <entity name>}
```

4.27.2 Action

This rule controls statements that are inside accept statements and could safely be moved outside. 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 entities whose names are given as parameters to the rule are never considered movable. Note that if a generic entity name is given, or the name of an entity declared in a generic package, 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);
```

4.27.3 Tips

The entity names 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.

4.28 Naming_Convention

4.28.1 Syntax

```
<check|search|count> naming_convention
   ([root] {<Location>} <Filter_Kind>,
    [case_sensitive|case_insensitive] [not] "<Pattern>"
    {, ...});
<Location> ::= global | local | unit
```

```
<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 |
                          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 |
                       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
```

4.28.2 Action

This rule controls the declaration of identifiers that do not follow the project's naming conventions. The first parameter defines the kind of declaration to which the rule is applicable, and other parameters are regular expressions that define the patterns that must be matched. See Chapter 7 [Syntax of regular expressions], page 78. Note that the pattern needs not include any wildcard, but if it does, it must be enclosed in quotes.

If one or more <Location> keyword is specified, the pattern applies only to identifiers. 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 program 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.

If "case_sensitive" is specified, pattern matching considers casing. Otherwise (default or "case_insensitive"), casing is irrelevant. 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 “root”, rules above it in the hierarchy are not considered (neither for itself not its children). This is useful to make exceptions to a more general rule.

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.

For renamings, the applicable rule is the one for the renamed entity. Similarly, 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).

Ex:

```
-- All identifiers must have at least 3 characters:
check naming_convention (all, "...");

-- 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$");

-- Exception to the rule for "all":
-- No minimum length for "for loop" identifiers
check naming_convention (root loop_control, ".");

-- "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_");
```

4.28.3 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 name, or both.

“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.

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".

4.28.4 Limitations

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

4.29 No_Safe_Initialization

4.29.1 Syntax

```
<check|search|count> no_safe_initialization [(<check_kind> [,<check_kind>])]
check_kind ::= out_parameter | variable
```

4.29.2 Action

This rule controls variables and/or **out** parameters that are not "safely" initialized. 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, until anything other than assignments, **if** or **case** statements, or procedure calls is encountered. Variables assigned in **if** or **case** statements must receive a value in all paths. The value can be given either through assignment or by having the variable as an **out** (but not **in out**) parameter of a procedure call. This rule can be given only once for each value of <check_kind>. Without parameters, it is equivalent to giving both.

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.

Ex:

```
check no_safe_initialization (out_parameter);
```

4.29.3 Limitation

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 4.46 [Uncheckable], page 63. It is then easy to disable the rule for this variable. See Section 3.7 [Disabling rules], page 22.

4.30 Non_Static

4.30.1 Syntax

```
<check|search|count> non_static [(context_kw {, context_kw})];
context_kw ::= index_constraint | discriminant_constraint | instantiation
```

4.30.2 Action

This rule controls that expressions used in certain contexts are static. These are index constraints if the keyword `index_constraint` is given, discriminant constraints if the keyword `discriminant_constraint` is given, or instantiations if the keyword `instantiation` is given. If no keyword is given, all contexts are controlled.

This rule is useful in contexts where the space occupied by data structures must be computable from the program text.

Ex:

```
check non_static (index_constraint);
```

4.31 Not_Elaboration_Calls

4.31.1 Syntax

```
<check|search|count> not_elaboration_calls
   (<subprogram name> {, <subprogram name>});
```

4.31.2 Action

This rule controls subprogram calls (procedure, function or entry calls) that are performed at any time except during the elaboration of library packages.

Ex:

```
search not_elaboration_calls (Data.Initialize);
```

4.31.3 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.

4.32 Parameter_Aliasing

4.32.1 Syntax

```
<check|search|count> parameter_aliasing [(Certain|Possible|Unlikely)];
```

4.32.2 Action

This rule controls aliased use of variables in subprogram calls. Specifically, this rule will identify 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);
```

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 integer or enumeration literals, the rule will be able to eliminate the diagnosis of aliasing (if the values are different). This does not cover all cases of static expressions, but will avoid 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;
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.

4.32.3 Limitation

Due to a weakness of the ASIS standard, 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!

4.33 Other_Dependencies

4.33.1 Syntax

```
<check|search|count> other_dependencies (<unit> {,<unit>});
```

4.33.2 Action

This rule controls semantic dependencies (i.e. **with** clauses) to units other than those indicated. This rule can be specified only once.

Ex:

```
check other_dependencies (Ada.Text_IO);
```

4.34 Potentially_Blocking_Operations

4.34.1 Syntax

```
<check|search|count> potentially_blocking_operations;
```

4.34.2 Action

This rule controls usage of potentially blocking operations (as defined in LRM 9.5.1 (8..16)) from within protected operations. It does follow the call graph, therefore identifying indirect potentially blocking operations. 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;
```

4.34.3 Limitation

There is one case defined in LRM E.4(17) which is not recognized: remote subprograms calls.

4.34.4 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.

4.35 Pragmas

4.35.1 Syntax

```
<check|search|count> pragmas
    (all|nonstandard|<pragma name> {, <pragma name>});
```

4.35.2 Action

This rule controls usage of one or several specific pragmas. 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. Ex:

```
check pragmas (elaborate_all, elaborate_body);
```

4.35.3 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`).

4.36 Reduceable_Scope

4.36.1 Syntax

```
<check|search|count> reduceable_scope [no_blocks]
```

4.36.2 Action

This rule controls declarations that could be moved to some inner scope. More precisely, it will report on any declaration that is referenced only from a single, inner scope. However, entities that are used in a 'Access or 'Address attribute are never reported, since moving them would change their accessibility level.

If `no_blocks` is specified, the rule will not consider blocks as possible targets for a reduced scope.

As a side effect, the rule will report about entities that are declared but not used.

Ex:

```
check reduceable_scope;
```

4.37 Representation_Clauses

4.37.1 Syntax

```
<check|search|count> representation_clauses
   [(<repr_kw>|<attribute> {, <repr_kw>|<attribute>}, ... )];

repr_kw ::= at | at_mod | enumeration | record
```

4.37.2 Action

This rule controls usage of representation clause. Without parameter, it will control all representation clauses, otherwise it will control the representation clauses given as parameter.

"at" checks for address clauses given in Ada 83 style ("for XXX use at"). "at_mod" checks for alignment clauses given in Ada 83 style ("for T use record at mod XX;"). "enumeration" checks for enumeration representation clauses. "record" checks for record representation clauses. In addition to these keyword, any specifiable attribute can be given (including the initial "'"); the rule will check for a specification of this attribute. 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);
count representation_clauses ('SIZE);
```

4.38 Return_Type

4.38.1 Syntax

```
<check|search|count> return_type [(<type_kind> {, <type_kind>})];
type_kind ::= class_wide          | unconstrained_discriminated |
              unconstrained_array | task | protected
```

4.38.2 Action

This rule controls functions whose return type belongs to one of the indicated type kinds:

- `class_wide` controls class-wide 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.

If no type kind 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);
```

4.38.3 Limitations

There is a (very rare) case where AdaControl does not properly recognize that a function returns a class-wide type. This is due to an ASIS bug fixed in version 5.05, and therefore appears only with earlier versions of the compiler. This happens when a generic unit contains functions whose return type is a generic indefinite formal type, and this generic is instantiated with a class-wide type.

4.39 Side_Effect_Parameters

4.39.1 Syntax

```
<check|search|count> Side_Effect_Parameters
   (<function name> {, <function name>});
```

4.39.2 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 functions mentionned as parameters in the rule 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.

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.

4.39.3 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.

4.40 Silent_Exceptions

4.40.1 Syntax

```
<check|search|count> Silent_Exceptions (<element> {, <element>});
element ::= <subprogram name> | raise | return | requeue
```

4.40.2 Action

This rule controls exception handlers that can cause exceptions to silently disappear, i.e. 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.

The parameters are the Ada callable constructs considered "reporting". In addition to subprogram and entry names, the special names "raise", "return" and "requeue" mark raise statements, return statements, and requeue statements (respectively) as reporting. If a generic procedure or function is given to the rule, 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.

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:

```
check silent_exceptions (raise, reports.trace);
```

If the **raise** statements or subprogram calls 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.

If "raise" is given as a parameter, the procedures `Ada.Exceptions.Raise_Exception` and `Ada.Exceptions.Reraise_Occurrence` are automatically added to the list of procedures for both Check and Search, unless they are explicitely specified as a parameter in a rule. This way, it is possible to consider them as reporting procedures for Check (for example) and not for Search.

4.40.3 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 the next 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...

4.41 Simplifiable_Expressions

4.41.1 Syntax

```
<check|search|count> Simplifiable_Expressions
   [(<Expression_kw> {, <Expression_kw>})];

Expression_kw ::= range         | logical     | logical_true |
                  logical_false | parentheses
```

4.41.2 Action

This rule controls expressions that can be simplified. The "range" parameter controls expressions of the form `T'First .. T'Last` that should be `T'range` (or even simply `T`). "logical_true" controls redundant boolean expressions of the form `<expr> = True` (or `/=`), and "logical_false" does the same for comparisons with `false`. "logical" is the same as specifying both "logical_true" and "logical_false". "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.

Ex:

```
search simplifiable_expressions (parentheses);
check  simplifiable_expressions (range, logical);
```

4.41.3 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.

4.42 Special_Comments

4.42.1 Syntax

```
<check|search|count> Special_Comments ("<pattern>" {, "<pattern>"});
```

4.42.2 Action

This rule controls comments that match one of the given patterns. 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 Chapter 7 [Syntax of regular expressions], page 78 for details. Note that the pattern needs not include any wildcard, but if it does, it must be enclosed in quotes. Pattern matching is always case insensitive.

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

Ex:

```
check special_comments ("TBSL");

-- Report places where rules are disabled:
search special_comments ("##.* off");
```

4.42.3 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.

4.42.4 Limitations

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

4.43 Statements

4.43.1 Syntax

```
<check|search|count> statements (<statement_kw> {, <statement_kw>};

statement_kw ::=
   abort                   | accept_return          | asynchronous_select |
   block                   | case_others            | case_others_null    |
   conditional_entry_call  | delay                  | delay_until         |
   dispatching_call        | entry_return           | exception_others    |
   exception_others_null   | exit                   | exit_for_loop       |
   exit_while_loop         | for_loop               | function_return     |
   goto                    | labelled               | loop_return         |
   multiple_exits          | no_else                | null                |
   procedure_return        | raise                  | raise_standard      |
   requeue                 | reraise                | selective_accept    |
   simple_loop             | terminate              | timed_entry_call    |
   unconditional_exit      | unnamed_block          | unnamed_exit        |
   unnamed_loop_exited     | unnamed_multiple_loop  | unnecessary_null    |
   untyped_for             | while_loop             | while_true
```

4.43.2 Action

This rule controls usage of certain Ada statements.

- Statement keywords that are Ada keywords control the corresponding Ada statements; note that `delay` will control only relative **delay** statements (i.e. it will not control the **delay until** statement).
- `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 that appear inside a **loop** statement.

- `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.
- `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.
- `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.
- `exit` controls all exit statements, while `exit_for_loop` and `exit_while_loop` control **exit** statements that terminate **for** and **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.
- `exception_others` controls any **when others** exception handler, while `exception_others_null` controls only **when others** exception handlers that contain only **null** statements.
- `for_loop` controls all `for` loops.
- `function_return` controls 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.
- `labelled` controls statements with a label (true statement labels, not block and loop names).
- `no_else` controls `if` statements that have no `else` path.
- `null` controls all **null** statements, while `unnecessary_null` controls only **null** statements that serve no purpose and can be removed. Note that if a **null** statement carries a label, it is not considered unnecessary.
- `raise` controls all **raise** statements, while `raise_standard` controls `raise` statements that raise one of the predefined exceptions (those declared in package `Standard`) and `reraise` controls only **raise** statements in exception handlers that reraise the same exception. Note that `raise_standard` and `reraise` take precedence over `raise` if they are mentionned together, but that `raise` will control all form of **raise** statements if no more specific subrule is given.
- `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_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 whose that uses a range without an explicitely named type (i.e. `for I in 1..10 loop`)
- `while_loop` controls all `while` loops, while `while_true` controls `while` loop statements where the condition is a plain `True`.

Ex:

```
search statements (delay);
check  statements (goto, abort);
check  statements (case_others_null, exception_others_null);
```

4.43.3 Tips

`while_true` 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...

4.44 Style

4.44.1 Syntax

```
<check|search|count> style;
<check|search|count> style (casing_identifier, <casing_kw>);
<check|search|count> style (casing_attribute,  <casing_kw>);
<check|search|count> style (casing_pragma,     <casing_kw>);
<check|search|count> style (compound_statement);
<check|search|count> style (default_in);
<check|search|count> style (exposed_literal, <type_kw>, {, <value_place>});
<check|search|count> style (multiple_elements {,<element_kw>});
<check|search|count> style (negative_condition);
<check|search|count> style (no_closing_name [, <max_lines>]);
<check|search|count> style (numeric_literal, [not] <base> [, <block_size>]);
<check|search|count> style (positional_association
                             {,<context_kw> [,<max_allowed>]}
                             | [, <max_allowed>]);
<check|search|count> style (renamed_entity);

casing_kw   ::= uppercase | lowercase | titlecase | original
context_kw  ::= pragma          | discriminant     | call | instantiation |
                array_aggregate | record_aggregate
element_kw  ::= clause | declaration | statement
type_kw     ::= integer | real | character | string
value_place ::= <value> | <place>
value       ::= <integer number> | <real number> | <pattern>
place       ::= number | constant | var_init | repr_clause
```

4.44.2 Action

This rules controls usage of various Ada coding style. The first parameter specifies which style aspect is to be checked:

- "casing_identifier", "casing_attribute", and "casing_pragma" control that identifiers (respectively attributes 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.
- "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.
- "exposed_literal" controls the usage of literals (aka "magic values"), that appear outside of constants or named numbers declarations. The second parameter tells to which kind of literals the rule applies. The (optional) indicated values that follow are allowed at any place; for strings, they are regular expressions. See Chapter 7 [Syntax of regular expressions], page 78. 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: "number" stands for named number declarations, "constant" for constant declarations, "var_init" for the initialization expression of variable declarations, and "repr_clause" for representation clauses. 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, and statements that do not start on a line of their own (i.e. when there are more than one of these on the same line). Extra parameters specify which kind of element to check; if not specified, all kind of elements are controlled.
- "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.
- "positional_association" controls pragmas, discriminants, calls, aggregates, or instantiations that use positional associations. Extra parameters specify which kind of construct to check; if not specified, all constructs are controlled. Each of the construct keywords is optionally followed by an integer value; if it is specified, it gives the maximum number of associations that are allowed to be positional, i.e. the rule will trigger only if there are more than the indicated number of associations. See examples below.

  Note that for calls, positional association is *not* reported for operators that use infix notation nor for calls to subprograms that are attributes, since named notation is not allowed in these cases. For calls, another rule controls positional associations according to the value of parameters rather than their number: See Section 4.18 [Insufficient_Parameters], page 42.
- "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 (literal, 10, 3);
check style (exposed_literal, integer, 0, 1);
check style (exposed_literal, real, 0.0, 1.0);
```

```
 -- All positional associations:
check style (positional_association);

-- All positional associations in calls and aggregates:
check style (positional_association, aggregate, call);

 -- All positional associations with more than 3 elements:
search style (positional_association, 3);

-- Positional associations in calls with more than 3 elements,
-- and positional associations in aggregates with more than 4 elements:
search style (positional_association, call, 3, aggregate, 4);
```

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_identifier, original);
style (casing_attribute, titlecase);
style (casing_pragma, titlecase);
style (positional_association);
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);
```

4.44.3 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 notation any more), and calls to subprograms that are attributes (since named notation is not allowed for these).

In many cases, badly laid-out compound statements will trigger both the "multiple_elements, statement" and the "compound_statement" subrules. For example:

```
if C then I := 1; end if;
```

will complain that the assignment is on the same line as the **if**, and that the **if** statement spans less than 3 lines. However, the subrules are not equivalent. For example,

```
if C then I := 1;
end
if;
```

will only find that the assignment is on the same line as the **if**, while

```
if C then
   I := 1; end if;
```

will only find that the **if** statement spans less than 3 lines. In most cases, you'll want to specify both subrules to ensure proper lay-out.

4.44.4 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 4.46 [Uncheckable], page 63.

4.45 Terminating_Tasks

4.45.1 Syntax

```
<check|search|count> terminating_tasks
```

4.45.2 Action

This rule controls tasks that can terminate. 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
```

4.45.3 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 4.43 [Statements], page 58.

4.46 Uncheckable

4.46.1 Syntax

```
<check|search|count> Uncheckable [(<risk_kw> [,<risk_kw>])];
<risk_kw> ::= false_positive | false_negative | missing_unit
```

4.46.2 Action

If the keyword “missing_unit” is given, this rule controls missing units, i.e. units 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.

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);
```

4.46.3 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.

4.46.4 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”!

4.47 Unnecessary_Use_Clause

4.47.1 Syntax

```
<check|search|count> unnecessary_use_clause;
```

4.47.2 Action

This rule controls **use** clauses that do not serve any purpose and can safely be removed. This happens in two cases:

- A **use** clause is given, but no element from the corresponding package is mentionned in its scope.
- A **use** clause is given within the scope of an enclosing **use** clause for the same package.

In the first case, just remove the **use** clause. In the second case, the rule will signal the location of the enclosing **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.

This rule will also signal **use** clauses given in a package specification that can safely be moved to the body. Since this rule has no parameters, it can be given only once (otherwise, it is an error).

Ex:

```
search unnecessary_use_clause;
```

4.47.3 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 3.7 [Disabling rules], page 22. 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.

4.48 Unsafe_Paired_Calls

4.48.1 Syntax

```
<check|search|count> unsafe_paired_calls
   (<Opening procedure>, <Closing procedure> [, <Lock type>]);
```

4.48.2 Action

This rule controls usage of calls to operations that are normally paired (like P/V operations) and do not follow the "safe" pattern defined below. 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).

The "safe" 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);
```

4.48.3 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.

4.48.4 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 4.46 [Uncheckable], page 63.

Due to limitations of internal date structures, this rule can be specified at most 32 times.

4.49 Unsafe_Unchecked_Conversion

4.49.1 Syntax

```
<check|search|count> unsafe_unchecked_conversion
```

4.49.2 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
```

4.49.3 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 4.46 [Uncheckable], page 63.

4.50 Usage

4.50.1 Syntax

```
<check|search|count> usage
   (variable|constant|object {,[not] from_spec|read|written|initialized});
<check|search|count> usage
   (exception {,[not] from_spec|raised|handled});
<check|search|count> usage
   (task {,[not] from_spec|called|aborted});
<check|search|count> usage
   (protected {,[not] from_spec|called});
<check|search|count> usage
   (all [,[not] from_spec]);
```

4.50.2 action

This rule controls how certain entitities (variables, constants, exceptions, tasks and protected objects) are used. The first parameter defines the class of entities to be controlled (“object” stands for both “constant” and “variable”, 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_spec” restrict entities being checked to those that appear in (generic) package specifications. 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) 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. Exceptions raised by calling `Raise_Exception` and tasks aborted by calling `Abort_Task` are properly recognized as exceptions begin 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 69.

Note that usage of entities whose declaration is not processed (like, typically, elements declared in standard packages like `Ada.Text_IO`), is not reported.

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 usage (exception, not raised);
check usage (task,      aborted);
check usage (protected, not called);
count usage (task);
```

4.50.3 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.

4.50.4 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);
```

Due to a weakness of the ASIS standard, usages of variables used as parameters to dispatching calls are ignored. This limitation will be removed as soon as we find a way to work around this problem, but the issue is quite difficult!

4.51 Use_Clauses

4.51.1 Syntax

```
<check|search|count> use_clauses [(<package name> {, <package name>})];
```

4.51.2 Action

This rule controls usage of use clauses, *except* for the ones that name one of the mentioned packages. It is therefore possible to allow use clauses just for certain packages.

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:

```
check use_clauses (Ada.Text_IO, Ada.Wide_Text_IO);
```

4.52 With_Clauses

4.52.1 Syntax

```
<check|search|count> with_clauses [(<with_kw> [, <with_kw>])];
with_kw ::= multiple_names|reduceable|inherited
```

4.52.2 Action

This rule controls **with** clauses that should be removed or moved to a better place. Each of the keywords can be given at most once. If no keyword is given, both `reduceable` and `inherited` are assumed.

If the keyword `multiple_names` is given, the rule will report on any **with** clause that mentions more than one unit name.

If the keyword `reduceable` is given, the rule will report:

- 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.

If the keyword `inherited` is given, the rule will report when a child unit or a subunit uses 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.

Ex:

```
check with_clauses (multiple_names, reduceable);
search with_clauses (inherited);
```

4.52.3 Tips

A **with** clause can safely be removed if it is unused, and no child unit (or subunit) reports that the unit is inherited.

5 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 checks, 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 checks can be written.

5.1 Rules files provided with AdaControl

The `rules` directory provides 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.

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.

5.2 Automatically checkable rules

Below are examples of rules that can be directly checked by AdaControl.

Goto statement shall not be used

```
check statements (goto);
```

Short circuit forms should be preferred over corresponding logical operators

```
Use_Short_Circuit: search expressions (and, or);
```

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 real_operators;
```

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_range, 10);
```

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$
```

5.3 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.

6 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 with scripts.

6.1 Migrating from 1.5r24

6.1.1 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".

6.1.2 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_");
```

6.1.3 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);
```

6.1.4 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);
```

6.1.5 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);
```

6.1.6 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);
```

6.2 Migrating from 1.4r20

6.2.1 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.

6.2.2 Declarations

The parameters “access” and “access_subprogram” have been changed to “access_type” and “access_subprogram_type”, for consistency with the new parameters.

6.2.3 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);
```

6.2.4 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);
```

6.2.5 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>);
```

6.2.6 Statement

Name changed from “statement” to “statements” (added an ’s’), to be consistent with other rules.

6.2.7 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);
```

7 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.