# AdaControl User Guide

Last edited: 4 October 2005

# 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 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.

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).

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.

# 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 you 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.

## 2.2 Building AdaControl

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:

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

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.

## 2.3 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, here are some hints:

- Some UNIX shell emulators add CR's to the end of each line of a text file. Since the reference for the tests is in UNIX (LF only) text format, this may explain the difference. Try running `dos2unix` on the content of the `res/` directory, then do a `diff` on the content of the `res/` and `ref/` directories.
- If you compiled with GNAT 3.15p, there are known bugs and unimplemented features that will not allow AdaControl to run correctly in some cases. Upgrade to a more 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.

## 2.5 Integrating AdaControl into GPS

It is possible to integrate AdaControl into GPS and make it directly available from GPS' menus. Simply copy the file `src/adacontrol.xml` into the `<GPS_dir>/share/gps/customize` directory.

GPS now features an "AdaControl" entry in the "Tools" menu, which can be used to run AdaControl on the currently edited file. All parameters (including the rule or rule file to use) can be set from the "Project/Edit Project Properties" menu. If you check "AdaControl rules files" in the "Languages" tab, GPS will recognize files with extension `.aru` as AdaControl rules files, and provide appropriate colorization. AdaControl options can be set from the "Switches/AdaControl" tab.

# 3 Program Usage

## 3.1 Running AdaControl

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 2.5 [Integrating AdaControl into GPS], page 4) or emacs (see Section 3.2.1 [Types and report messages], page 5). 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
```

## 3.2 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 3.6 [Disabling rules], page 16. If no label is present, the rule name is printed instead. 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. A syntax error in a rule causes the execution to stop.

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 defaul bracket encoding would not conveniently allow brackets for regular expressions, like those required for the rule "Naming_Convention" (see Section 4.11 [Naming_Convention], page 25).

### 3.2.1 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") is made of (separated by ':'):

- the file name (where the rule matches)
- the line number (where the rule matches)
- the column number (where the rule matches)
- the rule id (the rule that matches) or the rule label if there is one.
- 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).

Note that the format used for report messages is the same 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.

### 3.2.2 Parameters

Some rules work with parameters. Parameters can be:

- an Ada entity name
- an Ada keyword
- a keyword for the rule
- a numerical value

A numerical value is given with the syntax of an Ada integer litteral (underscores are allowed as in Ada). Based litterals are not currently supported; if somebody can justify a need for them, we'll be happy to add this feature later...

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.

### 3.2.3 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 3.7.1 [pfni], page 17.

#### 3.2.3.1 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}`).

#### 3.2.3.2 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}`

#### 3.2.3.3 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}`.

#### 3.2.3.4 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).

#### 3.2.3.5 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`.

#### 3.2.3.6 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`.

#### 3.2.3.7 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`.

### 3.2.4 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: ouputs, 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.

## 3.3 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 3.4.5 [Interactive mode], page 14), they can be used in command files as well.

### 3.3.1 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 3.3.5 [Clear command], page 10 to reset rules.

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

### 3.3.2 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 3.4.5 [Interactive mode], page 14.

### 3.3.3 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).

### 3.3.4 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.

### 3.3.5 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`.

### 3.3.6 Set command

Syntax:

```
set Output <output file>;
set Verbose | Debug | Ignore  On | Off
```

In the first 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 second form, this command allows to activate ("on") or deactivate ("off") options. "Verbose" corresponds to the "-v" option, "Debug" to the "-d" option, and "Ignore" to the "-i" option. See Section 3.4.7 [Verbose and debug mode], page 14 and Section 3.4.6 [Local deactivation ignoring], page 14 for details.

### 3.3.7 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.

### 3.3.8 Inhibit command

Syntax:

```
Inhibit <rule name> (<unit> {,<unit>});
```

This command will inhibit execution of the rule for the indicated unit(s). 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.

### 3.3.9 Example of commands

Below is an example of a file with multiple commands:

```
message "Searching Unchecked_Conversion";
search entitities (ada.unchecked_conversion);
output uc_usage.txt;
go;
clear all;
message "Searching 'Address";
search attribute (address);
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.

## 3.4 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 [-deiIrsuvw] [-f <rules file>] [-l <rules list>] [-o <output file>]
       [-p <project file>] {<unit>[+|-<unit>]|[@]<file>} [-- <ASIS options>]
```

or

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

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 3.7 [Helpful utilities], page 17.

### 3.4.1 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.

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.

### 3.4.2 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`. Note that when a unit is processed, all its subunits are processed at the same time; therefore, there is no need to specify subunits.

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 a "#" character, 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
```

### 3.4.3 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 3.4.5 [Interactive mode], page 14.

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.

### 3.4.4 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, AdaContol overwrites the output file if it exists.

Ex:

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

### 3.4.5 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 3.3.2 [Quit command], page 10) 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 ";".

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.

### 3.4.6 Local deactivation ignoring

The "-i" option tells AdaControl to ignore deactivation tags in Ada source code (see Section 3.6 [Disabling rules], page 16).

Ex:

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

### 3.4.7 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 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.

Ex:

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

### 3.4.8 Treat warnings as errors

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 3.5 [Return codes], page 16. It does not change the messages however.

### 3.4.9 Exit on error

If an error is encountered during processing 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 3.9 [In case of trouble], page 19.

Ex:

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

### 3.4.10 Project files

#### 3.4.10.1 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
```

#### 3.4.10.2 GPS project files

Currently, ASIS does not accept the "-P" option for GPS style project files. Should this change in the future, a "-P" option could be passed as described for the "-I" option. See Section 3.4.11 [ASIS options], page 15.

In the meantime, it is possible to use GPS project files by first compiling the modules to be checked with the "-gnatct" option (and of course the "-P" option for the project); this will save the so-called "tree files", which will appear with an ".adt" extension. AdaControl will use the tree files if they are available (and up to date), thus saving the recompilation and the need to specify any "-I" or "-P" option.

After running the tool, the tree files can be deleted.

### 3.4.11 ASIS options

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

Casual users don't need to care about ASIS options, except in one case: if 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 using an Emacs project file (the "-p" option), or by passing a "-I" option to ASIS, or by using ADA_INCLUDE_PATH.

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 3.8 [Optimizing AdaControl], page 17.

## 3.5 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.

## 3.6 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.

### 3.6.1 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
```

### 3.6.2 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
```

## 3.7 Helpful utilities

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

### 3.7.1 pfni

The convention used to refer to entities (as described in Section 3.2.3 [Specifying an Ada entity name], page 6) 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.

### 3.7.2 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 3.4.2 [Input units], page 12. 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 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 "--".

### 3.8.1 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. Such tree files are obtained by compiling the units with the "-gnatc -gnatt" options. Alternatively, the tree files can be generated automatically when needed.

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".

### 3.8.2 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.
- 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.

## 3.9 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 3.3.8 [Inhibit command], page 11.

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 Allocators

### 4.1.1 Syntax

```
<check|search|count> allocators [(<type name list>)];
```

### 4.1.2 Action

This rule controls usage of allocators. If type names are given, only allocators whose allocated type is mentioned 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);
```

### 4.1.3 Tips

If the allocated type is `T'Base` or `T'Class`, it will currently be found as T. This can be improved in the future.

## 4.2 Declarations

### 4.2.1 Syntax

```
<check|search|count> declarations (Declaration_kw {, Declaration_kw};
Declaration_kw ::= access | access_subprogram | aliased |
                   exception | task | tagged
```

### 4.2.2 action

This rule controls usage of certain Ada declarations. Declaration keywords that are Ada keywords match the corresponding Ada declarations. Note that `task` will match task type declarations as well as single tasks declarations; `access` will match all access type declarations, while `access_subprogram` will match only access to procedure or function declarations.

Ex:

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

## 4.3 Default_Parameter

### 4.3.1 Syntax

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

### 4.3.2 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 names is <formal name>, then:

- If the string `used` (case irrelevant) is given as the third parameter, the rule is fired if 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 is fired if 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.

Ex:

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

### 4.3.3 Limitations

Due to an unimplemented feature under some versions of ASIS-for-Gnat, this rule may give an ASIS error (ASIS_FAILED, with a diagnosis of "Not Implemented Query" if used for subprogram calls. It allways works OK with instantiations.

## 4.4 Entities

### 4.4.1 Syntax

```
<check|search|count> entities (<name list>);
```

### 4.4.2 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.

Ex:

```
search entities (Debug.Trace);
```

### 4.4.3 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. The `rules` directory of Adacontrol contains a file named `no_standard_unit.aru`. This file contains an Entity rule that forbids the use of any predefined Ada unit. Comment out the lines for the units that you want to allow. You can then simply "source" this files from your own rule file (or copy the content) if you want to disallow other units.

## 4.5 Entity_Inside_Exception

### 4.5.1 Syntax

```
<check|search|count> entity_inside_exception (<name list>);
```

### 4.5.2 Action

This rule controls exception handlers that contain references to one or several Ada entities specified as parameters.

Ex:

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

## 4.6 Exception_Propagation

### 4.6.1 Syntax

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

### 4.6.2 Action

This rule controls subprograms or tasks 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`.

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 3.2.3 [Specifying an Ada entity name], page 6). 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.

Finally, 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.

Ex:

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

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.

### 4.6.3 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.

### 4.6.4 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 exampe:

```
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!

## 4.7 Instantiations

### 4.7.1 Syntax

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

### 4.7.2 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).

### 4.7.3 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.

## 4.8 Local_Hiding

### 4.8.1 Syntax

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

### 4.8.2 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;
```

## 4.9 Local_Instantiation

### 4.9.1 Syntax

```
<check|search|count> local_instantiation (<generic name list>);
```

### 4.9.2 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).

Ex:

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

## 4.10 Max_Nesting

### 4.10.1 Syntax

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

### 4.10.2 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 for 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);
```

## 4.11 Naming_Convention

### 4.11.1 Syntax

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

### 4.11.2 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 define patterns, using the full Regexp syntax. Please refer to the `regexp` reference manual, or to the comments in file `gnat-regpat.ads` for details. If a pattern is preceded by "not", then the pattern must *not* be matched (i.e. the rule is fired if there is a match). Note that the pattern needs not include any wildcard, but if it does, it must be enclosed in quotes. 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.

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_parameter, ".");

-- "Upper_Initials" naming convention:
check naming_convention
   (all, case_sensitive "^[A-Z][a-z0-9]*(_[A-Z0-9][a-z0-9]*)*$");
```

### 4.11.3 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".

### 4.11.4 Limitations

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

If you compiled with the `Portable_String_Matching` package, only basic ("*" and "?") wildcards are available.

## 4.12 No_Closing_Name

### 4.12.1 Syntax

```
<check|search|count> no_closing_name [(<acceptable length>)];
```

### 4.12.2 Action

This rule 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.

Ex:

```
search no_closing_name;
check no_closing_name (5);
```

## 4.13 Not_Elaboration_Calls

### 4.13.1 Syntax

```
<check|search|count> not_elaboration_calls (<subprogram name list>);
```

### 4.13.2 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);
```

### 4.13.3 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.

## 4.14 Parameter_Aliasing

### 4.14.1 Syntax

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

### 4.14.2 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 litterals, 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.

### 4.14.3 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!

## 4.15 Pragmas

### 4.15.1 Syntax

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

### 4.15.2 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.

Ex:

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

## 4.16 Real_Operators

### 4.16.1 Syntax

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

### 4.16.2 Action

This controls usage of exact equality or inequality ("=" or "/=") between real (floating point or fixed point) values.

Ex:

```
check real_operators;
```

## 4.17 Representation_Clauses

### 4.17.1 Syntax

```
<check|search|count> representation_clauses
   [( at | at_mod | enumeration | record | <attribute>, ... )];
```

### 4.17.2 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);
```

## 4.18 Side_Effect_Parameters

### 4.18.1 Syntax

```
<check|search|count> Side_Effect_Parameters (<function name list>);
```

### 4.18.2 Action

This 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.

### 4.18.3 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.

## 4.19 Silent_Exceptions

### 4.19.1 Syntax

```
<check|search|count> Silent_Exceptions (<procedure name list>);
```

### 4.19.2 Action

This rule controls exception handlers that can cause exceptions to silently disappear, i.e. handlers that do *not* call one of the given procedures (for example a reporting procedure) nor re-raise an exception. Entry calls are accepted as well as procedure calls.

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 (reports.trace);
```

If the `raise` statements or procedure 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.

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.

### 4.19.3 Limitations

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.

## 4.20 Simplifiable_Expressions

### 4.20.1 Syntax

```
<check|search|count> Simplifiable_Expressions [(keyword)];
keyword ::= range | logical | logical_true | logical_false | parentheses
```

### 4.20.2 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 surrounding the expression of an "if" or "case" statement.

Ex:

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

## 4.21 Specification_Objects

### 4.21.1 Syntax

```
<check|search|count> specification_objects
   [( [not] constant|read|written|initialized {, ...})];
```

### 4.21.2 action

This rule controls usage of objects (variables and constants) declared in (generic) package specifications. Variables are often discouraged in package specifications, or need at least some extra control. Constants that are never used (not even in the package itself) are also suspicious. Moreover, some useful compiler warnings (like those about variables that should be declared constants) are not output for variables declared in library packages (at least with GNAT). This rule can do the same thing, project wide.

By default (without any parameter), this rule will report about usage of each object declared in a package specification which is part of the processed units; usage of objects whose declaration is not processed (like, typically, elements declared in standard packages like `Ada.Text_IO`, are not reported). The report includes the kind of package that declares the object (normal package, instantiation, or generic) and whether it is known to be initialized, read, and/or written. 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. 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).

In the case of objects declared in generic packages, the rule will report on usage of the objects 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 33.

It is possible to specify as parameter(s) one or several of the keywords (case irrelevant) `constant` `read`, `written`, or `initialized`, possibly preceded by `not`. The rule will output the information only for objects that match all the conditions given. The rule can be given once and only once for each combination of the parameters.

Ex:

```
search specification_objects (not read);
check specification_objects  (not initialized, not written, read);
```

### 4.21.3 Tips

An unspecified parameter in a rule stands for two rules (positive and negative form of the missing parameter). I.e.:

```
search specification_objects (read, written);
```

is the same as:

```
search specification_objects (read, written, initialized);
search specification_objects (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 specification_objects (read, written);
search specification_objects (read, written, not initialized);
```

Constants will be reported only for rules that apply (implicitely or explicitely) to "initialized, not written" since this is guaranteed by the language. The message tells it is a constant, but does not report "initialized, not written". Use "not constant" if you don't want reports about constants. Note that the notion of constants for this rule includes named numbers.

### 4.21.4 Limitation

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.

Due to a weakness of the ASIS standard, specification variables that appear as [`in`] `out` parameters in dispatching calls are not marked as "written". This limitation will be removed as soon as we find a way to work around this problem, but the issue is quite difficult!

## 4.22 Statements

### 4.22.1 Syntax

```
<check|search|count> statements (statement_kw {, statement_kw};
Statement_kw ::= abort | asynchronous_select | case_others | delay |
                 delay_until | exit | goto | raise | requeue
```

### 4.22.2 action

This rule controls usage of certain Ada statements. Statement keywords that are Ada keywords match the corresponding Ada statements; note that `delay` will match only relative `delay` statements (i.e. it will not match the `delay until` statement). `asynchronous_select` matches the

`select ... then abort` statement. `case_others` matches a `when others` path in a `case` statement. `unnamed_exit` matches an `exit` statement without a loop name that exits from a named loop.

Ex:

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

## 4.23 Unnecessary_Use_Clause

### 4.23.1 Syntax

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

### 4.23.2 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;
```

### 4.23.3 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 3.6 [Disabling rules], page 16. 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.

## 4.24 Use_Clauses

### 4.24.1 Syntax

```
<check|search|count> use_clauses (<package name list>);
```

### 4.24.2 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);
```

## 4.25 When_Others_Null

### 4.25.1 Syntax

```
<check|search|count> when_others_null [(exception | case, ...)]
```

### 4.25.2 Action

This rule controls "when others" case alternatives or exception handlers that contain only null statements. If no parameter is specified, both exception handlers and case statements are searched. Otherwise, it is possible to specify "exception" to search only exception handlers, or "case" to search only case statements.

This rule can be specified at most twice, once for "case" and once for "exception".

Ex:

```
check  when_others_null (exception);
search when_others_null (case);
```

# 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 Automatically checkable rules

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

**Goto statement shall not be used**

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

**All type names must start with "T_"**

```
check naming_convention (type, "^T_");
```

**All program units must repeat their name after the "end"**

```
check 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 specification_objects (not constant);
```

**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.

**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);
```

## 5.2 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.