AdaControl Programmer Manual

Last edited: 4 October 2005

This is the AdaControl Programmer Manual. It is intended for those who want to add new rules to AdaControl. Reading this manual is not necessary to use AdaControl. On the other hand, it is assumed that the reader knows how to use AdaControl before thinking of adding new rules.

Commercial support is available for AdaControl. 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.

Table of Contents

1 General

This programmer manual describes how to add new rules to AdaControl. Since AdaControl is based on ASIS, this manual assumes that the reader has some familiarity with ASIS programming.

The AdaControl tool includes several main components. Those that are relevant for writing new rules are:

- A general *framework* that provides services that are necessary to write rules. This includes a special module, `Framework.Plugs`, where rules are plugged-in;
- A set of *utilities* providing useful functionalities, but not specific to the writing of rules. Actually, the utilities packages are shared with other programs from Adalog's "Semtools" family of tools.
- The *rules* themselves.

This clear distinction makes it easy to add new rules. Actually, the framework relieves the programmer from all the "dirty work", and adding a new rule requires nothing else than caring about the rule itself.

2 The framework and utilities packages

The framework includes the package `Framework` itself and its public child packages. There are also some private child packages, but they are of course not relevant to the users of the framework.

In each package, services (declarations, subprograms) that are relevant for writing rules appear at the beginning of the package specification. Other services that are used by the rest of the framework, but not intended to be called from a rule, appear below the following comment lines:

```
--
--  Declarations below this line are for the use of the framework
--
```

This section provides an overview of the services that are made available by the framework and other utilities packages. It is not the purpose of this section to describe the syntax of every service provided : please refer to the comments in the specification of each package. Existing rules are also typical examples of how to use these functionnalities.

2.1 The package Framework

The package Framework includes general services, needed by most rules. These include:

- The notion of *location*, with associated subprograms. A location is a place within a source file where some construct happens.
- The notion of *rule context*. A rule context is some information that a rule associates to entities. For example, given the following rules:

  ```
  search Entities (Blah);
  Strictly_Forbidden: check entities (Ada.Unchecked_Conversion)
  ```

  the rule Entities must associate that `Blah` is the target of a search, and that `Ada.Unchecked_Deallocation` is the target of a check with label `Strictly_Forbidden`.

2.2 The package Framework.Rules_Manager

The package `Framework.Rules_Manager` is used to register and manage rules.

The procedure `Register` declares the name of the rule and the associated `Help`, `Add_Use`, `Command`, `Prepare`, and `Finalize` procedures.

Note that there is nothing else to do to make a rule known to the system: once it is registered, it will be recognized on the command line, help command will work, etc.

The procedure `Enter` is used to let the system know which rule is currently active.

2.3 The package Framework.Reports

The package `Framework.Reports` is used to report error or found messages when a rule matches. It deals automatically with things like rules being temporarily disabled, therefore the rule does not have to care.

The only procedure of interest for writing rules is `Report`. The specification of the Report procedure is:

```
procedure Report (Rule_Id   : in Wide_String;
                  Rule_Type : in Rules.Rule_Types;
                  Loc       : in Rules.Location;
                  Msg       : in Wide_String);
```

Note that there is only one string for the message. Please do not try to “improve” the presentation by introducing line breaks in the report message: the output of AdaControl should

remain parseable by rather naive tools, therefore it is necessary to ensure that one output line = one message.

2.4 The package Framework.Language

The package `Framework.Language` deals with the language used to specify which rules are used in a rules file. Only the subprograms used to parse parameters are relevant to the writing of rules.

There is a `Parameter_Exists` function that returns `True` if there are parameters left to parse. The corresponding parameter value can be retrieved with the `Get_Integer_Parameter`, `Get_String_Parameter`, or `Get_Entity_Parameter` functions. The two former functions return the parameter as an `Integer` or `Wide_String`, respectively. The latter one returns an entity specification, i.e. a descriptor for something which is expected to be a general specification for an Ada entity (including overloading information, for example). Such an entity can be used as a key for a context. There is also a generic procedure `Get_Flag_Parameter` that can be instantiated for the case of flags (keywords) parameters See Section 2.1 [The package Framework], page 3. Finally, there is a `Get_Modifier` to process modifiers (things like "not" or "case_sensitive" in front of a parameter).

2.5 The package Framework.Scope_Manager

The package `Framework.Scope_Manager` provides facilities for rules that need to follow scoping rules (i.e. which identifiers are visible at a given place). It provides subprograms to query currently active scopes, and a generic package that allows associating any kind of information to a scope. Scopes are automatically managed: the information will disappear when the corresponding scope is exited, except for information associated to package specifications that will be restored when the corresponding body is entered.

See the package specification for more details.

2.6 The package Framework.Plugs

Procedures in the package `Framework.Plugs` are called during the traversal of the Ada source code. Unlike the rest of the framework, this package does not provide services to rules, but instead *calls* processing procedures defined in the rules packages. Therefore, it is necessary to *plug* the corresponding calls in this package. This is described in details in Chapter 4 [Plugging-in a new rule into the framework], page 10.

2.7 The package Rules

The package `Rules` contains only the definition of some constants that can be used to dimension data structures. It's main purpose is to serve as the parent package of all rules.

2.8 The package Utilities

This package provides various general facilities that are not specific to AdaControl. The main elements provided are:

- `User_Message` and `User_Log`. Both procedure output a message, the difference being that `User_Log` outputs its message only in verbose mode. `User_Message` is used to print help messages. `User_Log` could be used if some rule wanted to print some extra information in verbose mode. Note that these procedures should *not* be used to report the result of a check or search (use `Framework.Reports.Report` instead).
- String handling services, see package specification

- Error management. The `Error` procedure is not to be called directly, use `Framework.Language.Parameter_Error` instead to report errors in user provided parameters. The `Failure` procedure is used to report internal failures. It is frequent in ASIS programming to have a big case statement over the various kinds of elements, of which only a few values are interesting or possible given the context. We strongly encourage to call `Failure` in the **when others** part of the case statement to trap unexpected cases. Note that the procedure is overloaded with a version that allows to print information about the failing element.
- Debugging facilities. Several `Trace` procedures allow you to output a message, possibly with the context of an ASIS element of element list. There is also an `Assert` procedure that calls `Failure` if its condition is false; well placed `Assert` calls are very helpfull in debugging. Note that traces are output only in debug mode.
- Other facilities for managing the output that are called by the framework, but not useful for writing rules.

2.9 The package Thick_Queries

This package contains high level services that are built on top of Asis queries, and can therefore be quite useful to the writing of rules. These queries are documented in the specification of the package.

2.10 The package Binary_Map

This generic package can be instantiated with a `Key_Type` and a `Value_Type`, and associates values of `Value_Type` to values of the `Key_Type`. The mapping uses a binary tree; if you use it to keep user information, it is appropriate to rebalance the tree before starting the actual processing. See [Prepare], page 8.

See existing rules for examples of using this package.

2.11 The package A4G_Bugs

AdaControl is quite demanding on the ASIS implementation, and we found some bugs in ASIS-for-GNAT during its development. These have been reported to ACT, and are fixed in the wavefront version of GNAT, or should be fixed very soon.

However, many people do not have access to the wavefront version, or prefer to stay with the stable version. This package provides replacements for some ASIS subprograms that do not behave as expected. Subprograms in this package have specifications identical to the corresponding ASIS subprograms, and are designed in such a way that there is no harm in using them with a version of ASIS that does not exhibit the bug. Therefore, it is strongly recommended to use the subprograms in this package rather than their ASIS equivalent.

Note that if you run the rules file `src/verif.aru` on your code, it will spot any use of an ASIS function for which there is a replacement in `A4G_Bugs`.

3 Writing a new rule

All rules currently provided follow a common pattern, described below; it is recommended that new rules do the same, in order to make maintenance easier.

The first thing to do before adding a new rule is to read the source for existing rules, as they provide good examples of how a rule is implemented. For an example of a simple rule, see `Rules.Entity`; for an example of a sophisticated one, see `Rules.Unnecessary_Use`. Note that `Rules.Entity` can be used as a template for writing new rules, as most rules will follow the same general structure, just making more elaborated processing of relevant entities.

3.1 General considerations

A rule is implemented as a child package of package `Rules`. The following sections describe the structure of the specification and body of a rule package.

It is good practice to use only one string type all over a program, and since ASIS is based on `Wide_String`, a rule should not use the type `String`, but rather use `Wide_String` instead.

3.2 Specification

The specification of a rule package must contain the following elements:

3.2.1 Header comment

It is recommended to start the specification with a comment describing what the rule does. This can be roughly the same as the content of the help message.

3.2.2 Rule_ID

Rule_ID is a constant of type `Wide_String`. It is the unique rule identifier of a rule. It is used by the package `Framework.Rules_Manager` as the key in the rules list to dispatch to the corresponding registered operation, and as the rule name used by the user on the command line to parameterize and use the rule. The name of the rule should be all upper-case (to allow for case-independant recognition).

Ex:

```
Rule_Id : constant Wide_String := "PRAGMAS";
```

Note that from a language point of view, this declaration could be in the body of the package; however, for identification purposes, it is more convenient to put it in the specification.

3.2.3 Process

One (or more) procedure(s) may be necessary to process the rule (collectively named the `Process` procedures in this document). These procedures are called from `Framework.Plugs` at appropriate places, and therefore must be declared in the specification of the rule. See Chapter 4 [Plugging-in a new rule into the framework], page 10.

These procedures take one parameter of type `Asis.Element`. Although all element kinds are equivalent from the point of view of Ada's type checking, it is recommended to follow general ASIS practice, and to define the parameter with the ASIS element kind expected by the procedure.

Ex:

```
procedure Process_Pragma (Pragma_Element : in Asis.Pragma_Element);
-- Expected Element_Kinds:
--    A_Pragma
```

3.3 Body

It is a good habit to start the body of a rule by giving a comment explaining the general principles of the algorithm used, especially if the algorithm is not trivial.

The body must contain a `Help`, an `Add_Use`, and a `Command` procedure. It may also optionnally contain a `Prepare` and a `Finalize` procedure. These procedures are call-backs that are registered to the framework by calling `Framework.Rules_Manager.Register` in the statements part of the body. This procedure has `null` defaults for the optional subprograms.

3.3.1 Help

Help is a procedure that displays a short help message to the standard output for the rule. It takes no parameter.

The procedure Help is called when the user specifies a "-h" option for the rule. It must display a useful message by calling `Utilities.User_Message`. In order to have a uniform presentation for all rules, the message must be structured as follows:

- The word "Rule:" followed by the rule ID
- The word "Parameters:" followed by a description of parameters
- A helpful message describing the purpose of the rule

Ex:

```
procedure Help is
   use Utilities;
begin
   User_Message ("Rule: " & Rule_Id);
   User_Message ("Parameter(s): pragma name (e.g. Elaborate_Body)");
   User_Message ("This rule can be used to check/search for the usage "
                 & "of a specific pragma.");
end Help;
```

3.3.2 Add_Use

`Add_Use` is a procedure which is called by the rules parser when it finds a use of the corresponding rule. It is passed the corresponding label (an empty string if there is no label), and the rule's type (`Check`, `Search` or `Count`). It will typically loop over the parameters with the various `Get_XXX_Parameters` from package `Rules.Language` to process the parameters.

If for some reason a parameter is not appropriate to the rule, the rule should call `Rules.Language.Parameter_Error` with an appropriate message. This procedure will raise the exception `User_Error`, and the `Add_Use` procedure should not handle it; the exception will be processed by the framework.

Note that `Add_Use` may be called several times if the same rule is activated with different parameters in a rules file. If a rule can be specified only once, it is up to the rule to check this and call `Parameter_Error` in case it is given more than once.

Ex:

```
procedure Add_Use (Label     : in Label;
                   Rule_Type : in Rule_Types) is
begin
   while Parameter_Exists loop
      -- process parameter
   end loop;
end Add_Use;
```

There is no special requirement on the implementation of the `Add` procedure. The programmer is free to interpret the parameters as necessary and do whatever initialisation processing they

imply. Typically, for a rule that searches for the occurrence of an identifier, this procedure would add the identifier to some internal context table.

3.3.3 Command

`Command` is a procedure used by the framework to send "commands" to the rule in order to change its state. It has a paraemeter of an enumeration type that can take the values `Clear`, `Suspend`, and `Resume`.

- `Clear`: `Command` is called with this value whenever a "clear" command is given. The rule must reset the rule to the "not used" state, and free any allocated data structure.
- `Suspend`: `Command` is called with this value whenever the rule is inhibited. The rule must preserve its current "used" state, and enter the "not used" state.
- `Resume`: `Command` is called with this value whenever the rule is no more inhibited. The rule must restore its state from the copy saved by the previous `Suspend`

This procedure is required, since it must at least deal with the `Rule_Used` flag (see [Process], page 8). Note that it is guaranteed that `Suspend`/`Resume` are properly paired, and that `Suspend` is not called on an already suspended rule. Therefore, a simple variable can be used to save the current state.

Ex:

```
procedure Command (Action : Framework.Rules_Manager.Rule_Action) is
   use Framework.Rules_Manager;
begin
   case Action is
      when Clear =>
         Rule_Used := False;
         -- Free internal data structures if necessary
      when Suspend =>
         Save_Used := Rule_Used;
         Rule_Used := False;
      when Resume =>
         Rule_Used := Save_Used;
   end case;
end Command;
```

3.3.4 Prepare

`Prepare` is a procedure that performs some initialisations that must be done after adding uses of the rule and before processing the units. It is optionnal (i.e. a **null** pointer can be passed for it to the `Register` procedure, or simply not mentionned since **null** is the default).

A typical use of `Prepare` is to balance the tree from a binary map to improve efficiency.

3.3.5 Process

There is no special requirement on the implementation of the `Process` procedure(s). The programmer is free to do whatever is necessary to the rule. It is possible to use ASIS query functions, or any other service deemed appropriate.

It is also possible to have several `Process` procedures (e.g. if the programmer wants to do some processing when going down the ASIS tree, and some other processing when going up).

A `Process` procedure should return immediately if no corresponding `Add_Use` has ever been called. In most cases, this is conveniently done by having a `Rule_Used` global boolean variable which is set to `True` in `Add_Use`, and checked at the beginning of `Process`.

After this test, the rule should immediately call `Rules_Manager.Enter` (with the rule name as the parameter). In case of a problem, this allows the system to report which rule failed.

3.3.6 Finalize

`Finalize` is called at the end of a "Go" command, after all units have been processed. It is useful for rules that report on global usage of entities, and therefore can report findings only at the end. It is optionnal (i.e. a **null** pointer can be passed for it to the `Register` procedure, or simply not mentionned since **null** is the default).

Ex:

```
procedure Finalize is
begin
   -- Report findings
end Finalize;
```

3.3.7 Package statements

The package body statements part should include a call to `Framework.Rules_Manager.Register` in order to register the rule and its associated `Help`, `Add_Use`, `Command`, and optionally `Prepare` and `Finalize`, procedures.

Ex:

```
begin
   Framework.Rules_Manager.Register (Rule_Id,
                                     Help    => Help'Access,
                                     Add_Use => Add_Use'Access,
                                     Command => Command'Access,
                                     Prepare => Prepare'Access);
end Rules.Pragmas;
```

4 Plugging-in a new rule into the framework

4.1 Normal case

Adding a new rule to the tool requires only simple modifications to the package `Framework.Plugs`.

The package `Framework.Plugs` contains several procedures that are called during the traversal of the code under the following circumstances:

- `Enter_Unit`: Called when entering a compilation unit, before any other processing.
- `Exit_Unit`: Called when leaving a compilation unit, after any other processing.
- `Enter_Scope`: Called when entering a new scope (i.e. a construct that can contain declarations).
- `Exit_Scope`: Called when leaving a scope.
- `Pre_Procedure`: Called when entering a syntax node (this is like the usual `Pre_Procedure` used in the instantiation of `ASIS.Iterator.Traverse_Element`, except that there is no `State_Information` and no `Control`).
- `Post_Procedure`: Called when leaving a syntax node.
- `True_Identifier`: Called when entering an `An_Identifier`, `An_Operator_Symbol`, or `An_Enumeration_Literal` node that corresponds to a real identifier, i.e. not to a pragma name or other forms of irrelevant names. This avoids special cases in rules dealing with identifiers.

These procedures have the usual "big case" structure of an ASIS application (i.e. a first level case statement on `Element_Kind`, with each case alternative containing other case statements to further refine the kind of node that is being dealt with).

The following modifications must be done to the body of this package:

1. Add a with clause naming the rule package:

   Ex:

   ```
   with Rules.Pragmas;
   ```

2. Add calls to the rule's `Process` procedure(s) at the appropriate place(s) in the body of the provided procedures.

   Ex:

   ```
   procedure Pre_Procedure (Element : in Asis.Element) is
      use Asis;
      use Asis.Elements;
   begin
      case Element_Kind (Element) is
         when A_Pragma =>
            Rules.Pragmas.Process_Pragma (Element);
          ...
   end Pre_Procedure;
   ```

Many alternatives of the big case statement cover a number of values. It may happen that a new rule requires calling its `Process` procedure for some, but not all of these values. In this case, the case alternative must be split. This is not a problem, but do not forget to duplicate the statements from the original alternative before adding the new calls, to make sure that the split does not break existing rules.

It is always possible to plug a `Process` procedure in `Pre_Procedure` or in `Post_Procedure`. However, some "natural" places for plugging rules correspond to many branches of the big case statement. For example, there are many places where you enter a scope. That's why the

package `Framework.Plugs` includes other procedures that are called in "interesting" contexts. If appropriate, it is better practice to plug calls to `Process` procedures here, rather than all over the place in various alternatives of the big case statement.

4.2 Specific rules

In some cases, you may want to keep your rules separate from the general purpose ones. This may happen if you have developped some very specific rules that take the structure of your project into account, and hence would not be of interest to anybody else. Or it may be that your local lawyer does not allow you to publish your rules as free software.

This should not prevent you from using AdaControl. Just write the rules as usual, but instead of plugging them in `Framework.Plugs`, use the package `Framework.Specific_Plugs` instead. This package has subprograms identical to those described above for plugging-in rules, and they are called in the same contexts. But it is guaranteed that no rule from the public release of AdaControl will ever be plugged-in into this package. This way, you can keep your rules separate from the public ones, and you can upgrade to a new version of AdaControl without needing to merge the modifications for your rules.

5 Testing and debugging a rule

5.1 Testing

Once the rule is written, you will test it. Of course, you'll first write a small test case to make sure that it works as expected. But that's not enough.

Our experience with existing rules has shown that getting the rule 90% right is quite easy, but the last 10% can be tricky. Ada offers constructs that you often didn't think about when writing the rule; for example, if you are expecting a name at some place, did you take care of selected names (we got trapped by this one several times)? Therefore, it is extremely important that you check your rule against as much code as you can, the minimum being the code of AdaControl itself.

5.2 Debugging aids

As mentionned above, it is often the case when writing a new rule, as well as with any kind of ASIS programming, that one comes across unexpected contexts. This is due to the rich features of Ada, but it is sometimes difficult to understand what is happenning.

The framework provides some facilities that help in debugging. Don't hesitate to use the `Trace` and `Assert` utilities. See Section 2.8 [The package Utilities], page 4. Note that the `Trace` procedures may be given an element (or an element list) whose basic characteritics are printed. If the `With_Source` parameter is `True`, the source correponding to the element is also printed.

In addition, a small stand-alone utility called `ptree` is provided. It prints the logical nesting of ASIS elements for a unit. The syntax of Ptree is:

```
ptree [-sS] [-p <project_file>] <unit> -- <ASIS_Options>
```

If the "-s" option is given, `ptree` processes the specification of the unit, otherwise it processes the body. If the -S option is given, the span of each element is also printed. The -p option and <ASIS_Options> have the same meaning as in AdaControl itself.

If you come across a situation where you don't understand the logical nesting of elements, try to reduce it to a very simple example, then run `ptree` on it. It can be quite instructive!

Of course, a more elaborated, but less convenient solution is to use Asistant. Please refer to your ASIS documentation to learn how to use Asistant.

Finally, if you come to suspect that you get a strange result from an ASIS provided operation, check whether there is an equivalent operation in the package `A4G_Bugs`, and if yes, use it instead. See Section 2.11 [The package A4G_Bugs], page 5.

5.3 Integrating the test in the test suite

When your rule has been carefully tested and is ready for integration, run the rule file `src/verif.aru` on every unit that you have written or changed. This will control that you match the programming rules for AdaControl. There can be some "found" messages (try to minimize them if possible), but there should be no "Error" message. Then, the last thing you have to do is to write a test for non-regression verification purpose. Don't forget to include examples of the tricky cases in the test.

Go to the `test` directory. You'll notice that all test programs have a name of the form `t_name.adb`. The `name` is the rule name. You'll notice also that some units have a name like `tfw_name.adb`; these are tests for the framework, you should normally ignore them. Name your test file according to this convention. It is OK for your test to have child units (whose names will be dictated by the Gnat naming convention). If your test requires other units, name them like `x_name` or `x_name_complement`. Then, go to the `test/conf` directory, and put your rule file under the name `t_name.aru` (with the same `name` of course).

Go back to the `test` directory, and run `test.sh`. All tests should report PASSED, except the `tfw_help` test. Your test will not be reported, because its expected output is not yet in the directory `test/ref`, and test `tfw_help` will report FAILED because this test prints all help messages, and that the help message for your rule has been added.

Check that the result of your test is OK (in the file `test/res/t_name.txt`), and copy this file to the directory `test/ref/`. Do the following command:

```
diff test/ref/tfw_help.txt test/res/tfw_help.txt
```

and check that the only difference is the addition of the help message from your rule. Then copy `test/res/tfw_help.txt` to the directory `test/ref/`.

Run `test.sh` again: it should print PASSED for all tests, including yours. All you have to do then is to send your modifications (including the tests) to rosen@adalog.fr, for inclusion in the next release of AdaControl!