AdaControl Programmer Manual

Last edited: 10 November 2016

This is the AdaControl Programmer Manual. It is intended for those who want to add new rules to AdaControl, or more generally modify (and presumably improve!) AdaControl. Reading this manual is not necessary to use AdaControl. On the other hand, it is assumed that the reader is familiar with how to use AdaControl.

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.

Modifying AdaControl needs of course a source distribution. It is OK to work with the regular source distribution, but if you intend to submit your patches, it is appropriate to get the latest "bleeding edge" version from our GIT repository on SourceForge. Instructions on how to get AdaControl from GIT are here

1.1 vocabulary

Some terms have a precise definition in AdaControl, and will be used with that signification in the rest of this manual.

A *rule* is an AdaControl package whose purpose is to recognize occurrences of certain constructs in Ada programs. All rules are children of the "Rules" package. By extension, the term rule is also used to designate the check that is performed by the package. A rule has a name, and may have parameters.

A *control* defines a check to be performed on some Ada text. A control is defined by a rule, and the value of the parameters given to the rule.

A *command* is a statement in the command language interpreted by AdaControl.

A *control command* is a kind of command that describes a check to be performed. A control commmand includes a *kind* ("check", "search" or "count", see user's guide), and a control (rule name and parameters).

A *context* is a set of values used by a rule to keep the characteristics associated with a control. Those values can, but need not necessarily, be the parameters of the control.

1.2 General organization

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 Adactl_Constants

AdaControl has some fixed size structures that limit the complexity of the programs it can handle, like for example the maximum numbers of parameters that subprograms can have, the maximum nesting of loops, etc.

These limits are set as constants in the package `Adactl_Constants`. These values are large enough to accomodate any reasonable program, but should you hit one of these limits, you can safely change them here. No other change is required.

If a rule needs to set a fixed dimension to some tables for example, it should use the constants defined in this package. If no existing constant is appropriate, add a new one to the package, don't define dimensioning constants in the rule itself.

2.2 The package Framework

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

- The definition of some constants that are used to fix a bound to the number of allowable constructs. Use these constants to dimension tables for example.
- 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`.

2.3 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_Control`, `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.

2.4 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 main service provided by this package is the `Report` procedure, which comes in two flavors. This is the only allowed way for a rule to report its findings, never use `Wide_Text_IO` or any other mean. The specifications of the Report procedures are:

```
procedure Report (Rule_Id    : in Wide_String;
                  Rule_Label : in Wide_String;
                  Ctl_Type   : in Control_Kinds;
                  Loc        : in Location;
                  Msg        : in Wide_String);

procedure Report (Rule_Id    : in Wide_String;
                  Context    : in Root_Context'class;
                  Loc        : in Location;
                  Msg        : in Wide_String);
```

The first procedure expects the label and type to be given explicitely, while the second one gets them from a `Context` object (see comments in the package).

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.

In addition, there is an `Uncheckable` procedure, with the following profile:

```
procedure Uncheckable (Rule_Id : in Wide_String;
                       Risk    : in Uncheckable_Consequence;
                       Loc     : in Location;
                       Msg     : in Wide_String);
```

This procedure is called each time a rule encounters some dynamic construct that prevents normal checking. The parameter `Risk` is `False_Positive` if the consequence of not being able to analyze the construct would result in wrong error messages, and `False_Negative` if it would result in not detecting something that could be an error. It is important to call this procedure for any non-checkable construct, since it is what allows the rule "Uncheckable" to work.

2.5 The package Framework.Language

The package `Framework.Language` provides functionalities for parsing parameters, as well as the rest of the command language; but of course only the subprograms used to parse parameters are relevant to the writing of rules.

The functionalities provided here allow a good deal of freedom for defining the syntax of a rule's parameters (although it is a good idea to stay as close as possible to the syntax of other rules). Look at the syntax of various rules to see what can be accomplished.

The package provides a `Parameter_Exists` function that returns `True` if there are parameters left to parse. The kind of the next parameter can be checked with the `Is_Integer_Parameter`, `Is_Float_Parameter`, or `Is_String_Parameter` functions. The corresponding parameter value can be retrieved with the `Get_Integer_Parameter`, `Get_Float_Parameter`, `Get_Name_Parameter`, `Get_File_Parameter`, `Get_String_Parameter`, or `Get_Entity_Parameter` functions. The latter function 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 a generic package `Flag_Utilities` to help manage flags (keywords) parameters defined by an enumerated type. An instantiation of this package provides a `Get_Flag_Parameter` procedure to parse the flags, an `Image` function to get a string representation of a flag, and a `Help_On_Flags` function to print the help message that enumerates all possible flag values.

There is a `Get_Modifier` to process modifiers (things like "not" or "case_sensitive" in front of a parameter). For more sophisticated modifiers, you can instantiate the generic package `Modifier_Utilities`, which works like `Flag_Utilities`, but also provides the notion of sets of modifiers.

Note that if you instantiate `Flag_Utilities` or `Modifier_Utilities` in a library package (as will be the case most of the time), you *must* put a `pragma Elaborate (Framework.Language);` on top of the package. Failing to do so will result in circular elaboration problems; (`pragma Elaborate_All`, as implicitely provided by GNAT, does *not* work).

2.6 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.

The scope manager follows strictly the visibility rules for child units: when entering a public child unit, the scope from the visible part of the parent is restored, and when entering the private part of the child, the scope of the private part of the parent is restored. In the case of a private child, the full scope of the parent is restored upon entering.

See the package specification for more details.

2.7 The package Framework.Variables

This package manages rule variables (the ones that can be set with the `set` command of the language).

A rule may use any number of rule variables for adjusting its behaviour. A rule variable holds a value, and various subprograms are provided for setting and getting the associated value.

Rule variables are extensions of `Framework.Variables.Object`. Typically, they extend this type with a value that can be of any type. Abstract subprograms that need to be overriden include `Set` and `Value_Image` for setting and getting a value from/to a string and `All_Values` to give possible values for help messages.

Generic packages are provided to make it easier to define variable types associated to enumerated types and integer types. In addition, the package `Framework.Variables.Shared_Types` provides variable types for commonly used associated types.

When a rule needs a rule variable, it declares an object of the appropriate type, and registers it to the framework by calling `Framework.Variables.Register`, passing a pointer to the object and the name used to refer to it. This name must be given in upper case, and start with the rule's name, like "`EXPRESSIONS.CALLED_INFO`" for example. Of course, it is better to use `Rule_Id` for building the name, like in:

```
Framework.Variables.Register (Called_Info'Access,
                              Variable_Name=> Rule_Id & ".CALLED_INFO");
```

This call is typically put in the initalization part of the rule's package, after the registration of the rule itself.

The rule `Expressions` (in file `rules-expressions.adb` includes a typical usage of a rule variable.

2.8 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 15.

2.9 The package Rules

The package `Rules` is (almost) empty. It's purpose is to serve as the parent package of all rules.

It simply provides an empty state type (`Null_State`), and a null procedure that can be used for instantiating `Traverse_Element` in simple cases.

2.10 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 procedures 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. In most cases, parameters are checked in the `Add_Control` procedure of the rule (see Chapter 3 [Writing a new rule], page 9), and therefore errors are reported during the parsing of the commands. In some cases, incorrect parameters are discovered while traversing the code. It is acceptable to call `Framework.Language.Parameter_Error` at any time, but be aware that this will immediately stop all analysis. See the `Rules.Unsafe_Paired_Calls` for an example of this.

  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 a boolean value, or the context of an ASIS element or 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.

2.11 The packages Thick_Queries and Framework.Queries

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

The difference between the packages is that `Thick_Queries` does not depend in any way on the other parts of AdaControl (and notably on the framework); it is therefore directly reusable for any ASIS application. On the other hand, `Framework.Queries` requires facilities provided by the framework, and is therefore not directly reusable outside of AdaControl.

2.12 The packages Linear_Queue and Binary_Map

These packages provide simple generic containers that are needed by several rules.

The generic package `Linear_Queue` can be instantiated with any (non-limited) `Component` type, and provides a simple queue of elements. Note that this queue has *value* semantics: when a queue is assigned, its content is duplicated. Queues are controlled, and therefore all internal storage allocations are internally managed. The package `Framework.Element_Queue` is an instantiation of `Linear_Queue` with the type `Asis.Element`.

The generic package `Binary_Map` 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 12.

See existing rules for examples of using this package.

2.13 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

There are two kinds of rules: *semantic* rules, which operate on Ada elements, and *textual* rules, which operate on the source text. In some rare cases, a rule can be of both kinds at the same time; see the rule "Style" for an example of this. Note that a semantic rule can still access the text of an Ada construct with the facilities provided by the package `Asis.Text`, this does not require the rule to be `Semantic_Textual`.

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`. For an example of a textual rule, see `Rules.Max_Line_Length`. Note that `Rules.Entity` can be used as a template for writing new semantic 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 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 must be given in 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.

3.2.2 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 15.

Process procedures of a semantic rule 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.

Process procedures of a textual rule take two parameters: an input line, and the corresponding location.

Ex:

```
-- Semantic rule:
procedure Process_Pragma (Pragma_Element : in Asis.Pragma_Element);

-- Textual rule:
procedure Process_Line (Line : in Asis.Program_Text;
                        Loc  : in Framework.Location);
```

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. To be honest, not all current rules do provide this information, and some crucial information may be missing in the rules that do... You are more than welcome to improve these comments, especially if you think that some fundamental information should have been provided here.

The body must contain a `Help`, an `Add_Control`, 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. Note that there is a parameter to this procedure that tells whether the rule is semantic, textual, or both. 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.
- A short description of the purpose of the rule. This description can span several lines, but don't make it too long.
- A blank line (call `Utilities.User_Message` without parameter).
- The word "Parameter(s):" followed by a description of parameters. Note that if you have a parameter of an enumerated type, the package `Flag_Utilities` features a `Help_On_Flag` procedure that formats automatically the values.

  If the different parameters have different meanings, you can use the form "Parameter(1):", "Parameter(2):" etc. instead. If the rule has no parameters, just say "Parameters: none".

Note that the command:

```
adactl -h all
```

outputs the help messages for all rules, providing examples of how you should write your own help messages.

Ex:

```
procedure Help is
  use Utilities;
begin
  User_Message ("Rule: " & Rule_Id);
  User_Message ("Control usage of specific pragmas");
  User_Message;
  User_Message ("Parameter(s): [multiple] all | nonstandard | <pragma>");
end Help;
```

3.3.2 Add_Control

`Add_Control` is a procedure which is called by the rules parser when it finds a control command that refers to the corresponding rule. It is passed the corresponding label (an empty string if there is no label), and the control's kind (`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_Control` procedure should not handle it; the exception will be processed by the framework.

Note that `Add_Control` 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_Control (Label    : in Label;
                       Ctl_Type : in Control_Kinds) is
begin
   while Parameter_Exists loop
      -- process parameter
   end loop;
end Add_Control;
```

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 parameter 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 12). 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 all controls refering to the rule have been parsed, and before processing the units. It is optional (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_Control` 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_Control`, and checked at the beginning of `Process`. For efficiency reasons, avoid doing any call to the ASIS library before this check. This means that if you need objects initialized with such calls, they should be declared in a block *after* the test, rather than in the declarative part of the `Process` procedure.

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.

A special case arises for rules that follow the call graph. Such rules may traverse elements outside the current unit, but should avoid analyzing units to which an `inhibit all` applies (so-called *banned* units). The framework features an `Is_Banned` function that tells if an element should not be traversed due to it being declared in a banned unit. See `Rules.Global_References` for an example of this.

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_Control`, `Command`, and optionally `Prepare` and `Finalize`, procedures. Note that the second parameter of `Register` tells whether it is a semantic, textual, or semantic_textual rule.

Ex:

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

3.4 Programming rules and tips

3.4.1 style

We try to maintain a consistent style in AdaControl, therefore the code you write should match the style of the rest of the program. Have a look at other rules, and run `src/verif.aru` on your code. In addition, please note the following:

- The **use** clause is allowed, but its scope should be restricted to the innermost declarative region where it is useful. Use a **use** clause for ASIS units, and another one for other units. Sort units alphabetically in the clause.

- The only output of a rule should be by calling `Report`. Especially, no rule should use `Wide_Text_IO` directly.
- If your rule encounters a dynamic construct that prevents normal checking, call `Framework.Reports.Uncheckable` to warn the user.
- The framework should be sufficient for all your needs. If you have a special need that you think cannot be satisfied by the current framework, get in touch with us and we'll discuss it.

3.4.2 Things to care about

Each time you want the name of something, remember that the name may be given in selected notation. In most cases, you should call `Thick_Queries.Simple_Name` on the result of any query that returns a name to get rid of the possible selectors. Otherwise, you should inspect the returned expression to see if its `Expression_Kind` is `A_Selected_Component`, and take the `Selector` if it is.

When designing a rule, consider the implications of renamings and generics.

If you want to output the Element_Image of some element, beware that it will be preceded by spaces. Do not use `Ada.Strings.Wide_Fixed.Trim` to eliminate them, since it wont remove tab characters. Use `Utilities.trim_all`, which will do the right thing.

3.4.3 Using ASIS efficiently.

Remember that ASIS queries can be costly. Declare local variables (or constants) rather than evaluating several times the same query.

There are issues with some ASIS queries. The rule whose label is "Avoid_Query" in `verif.aru` will remind you if you use one of these queries.

Asis.Definitions.Subtype_Mark
: There might be confusion with `Asis.Subtype_Mark`; moreover, you normally want to get rid of selected components (see above). Use `Thick_Queries.Subtype_Simple_Name` instead.

Asis.Definitions.Corresponding_Root_Type
: This query return a `Nil_Element` if any type in the derivation chain is a `'Base` attribute (to be honnest, versions of ASIS before the latest 5.05 will loop indefinitely in this case). Use `Thick_Queries.Corresponding_Root_Type_Name` instead, and consider what you want to do if there is a `'Base` in the derivation chain.

Asis.Definitions.Corresponding_Parent_Subtype
: This query suffers from the same problem as `Corresponding_Root_Type`. Don't use it, rather take the `Subtype_Simple_name` of the `Parent_Subtype_Indication`, and do your own analysis, depending on whether the returned `Expression_Kind` is `An_Attribute_Reference` or not.

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.
- `Text_Analysis`: Called on every source line of the code.

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. For textual rules, `Text_Analysis` is the only appropriate place.

   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.

If you have specific rules plugged into `Framework.Specific_Plugs`, change the constant `Specific_Version` in the specification of the package to something that identifies the specific version (like your company's name for example). This way, the version number of AdaControl will show that it is a specific version.

4.3 User documentation

Of course, you should update the user's guide with the information about your rules. This guide is written in Texinfo, see `http://www.gnu.org/software/texinfo/`. Note however that you don't need to understand all the possibilities of Texinfo to update the manual; look at the description of other rules, the few commands you need will be quite straightforward to understand.

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.

Note that if your rule encountered some uncheckable cases, you should add a child for your rule to the test `t_uncheckable`, and also modify the `t_uncheckable.aru` file accordingly. Look at how it is done currently, and do the same.

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 2.10 [The package Utilities], page 6. 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 the case where AdaControls enters an endless loop, the first thing to do is to determine the place where the loop is happening. To ease this, AdaControl may be compiled in "interruptible" mode. In normal mode, the package `Framework.Interrupt` is a renaming of `Framework.Interrupt_Std`, a dummy package that does nothing. Edit `Framework.Interrupt` to make it a renaming of `Framework.Interrupt_Dbg` (instructions provided in the package) and recompile.

Now, when you hit Ctrl-C while AdaControl is running with the "-d" option, execution of the current "go" command is interrupted with a message telling which rule is active, and on which compilation unit. If the "-x" option is also given, the whole execution is stopped.

Of course, when you are done, reestablish the normal package by doing the inverse manipulation. The reason we didn't put the debug version in the regular version is that it drags in the whole tasking run-time, with a measurable impact on efficiency (we measured 18% extra time for running AdaControl on the ACATS).

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>[:<span>] -- <ASIS_Options>
<span> ::=   <line_number>
           | [<first_line>]-[<last_line>]
           | <line_number>:<column_number>
```

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 has the same meaning as in AdaControl itself. ASIS options can be passed, like for AdaControl, after a "--" (but -FS is the default).

The <unit> is given either as an Ada unit, or as a file name, provided the extension is ".ads" or ".adb" (as in AdaControl). If a span is mentionned behind the unit name, only the constructs that cover the indicated span are output. The syntax of the span is the same used by pfni. This is useful if you are interested in just one particular structure in a big unit.

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 2.13 [The package A4G_Bugs], page 8.

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` (or in some rare cases, `ts_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, normally using `t_`, unless the test requires some weird parameters that prevent it from being run normally, in which case it should use `ts_` ("s" stands for special). 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` and `tfw_check` tests. Your test will not be reported, because its expected output is not yet in the directory `test/ref`; test `tfw_help` will report FAILED because this test prints all help messages, and that the help message for your rule has been added; test `tfw_check` will report FAILED because there is now one extra message (from the extra rule file) saying "No error found".

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.

Do the following command:

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

and check that the only difference is the addition of one "No error found" message.

Then copy `test/res/tfw_help.txt` and `test/res/tfw_check.txt` to the directory `test/ref/`. Run `test.sh` again: it should print PASSED for all tests, including yours. Pay special attention to the last test, called `tfw_stress`. This test runs all rules against all test units. If it fails, the file `res/tfw_stress.txt` contains the whole listing of the run (with `-dv` options), so you'll find all the context of the problem.

In case of problems, note that options can be passed to `test.sh`; check the comments at the top of the file for details.

When everything is OK, all you have to do is send your modifications (including the tests) to rosen@adalog.fr, for inclusion in the next release of AdaControl!