AdaSubst User Guide

Last edited: 19 April 2019

Table of Contents

1 Introduction

AdaSubst is a tool for making semantic substitutions in Ada programs. This means that it does not replace *text*, but performs logical transformations of Ada programs. The transformations consider all Ada rules, including visibility rules, renamings, etc. to make sure that all the intended elements, and only those, are replaced. For example, transformations that affect a generic will be applied to all uses in instantiations; similarly, a transformation that affects the name of a primitive operation of a type will be applied to the corresponding operation of all derived types...

AdaSubst offers several functions:

- Identifier translations, where Ada elements (identifiers) are replaced by other identifiers, possibly located in other places (packages).
- Removal of all representation clauses from a program.
- Removal of all **use** clauses, or only of **use package** clauses (keeping other forms - **use type** and **use all type** clauses). Identifiers in the scopes of these clauses are changed to full dotted notation where necessary.

Other forms of substitutions may be added in the future; we are looking forward to user suggestions about what could be useful.

AdaSubst has already been used in industrial contexts, and has processed succesfully many thousands of lines. It is another example of the nice things that can be made using ASIS.

In the design of AdaSubst, we chose the most rigorous approach, hence the choice of ASIS. This means that we do not believe that AdaSubst would generate a code that compiles and is wrong. If in doubt, it will produce code that does *not* compile, so you can fix it manually.

However, using ASIS has its drawback: the code must be compilable, and running AdaSubst on it can take as long as it would need to fully compile it. Since AdaSubst is used normally only once on any source file, it seems a sensible compromise.

Please send bug reports, remarks, notes, good ideas, improvements to the documentation, etc. to J-P. Rosen.

2 Installation

Like any ASIS application, AdaSubst can be run only if the compiler available on the system has exactly the same version as the one used to compile AdaSubst itself. The executable distribution of AdaSubst will work only with GNAT version GPL 2016, as distributed by AdaCore. If you are using any other version, please use the source distribution of AdaSubst and compile it as indicated below.

2.1 Building AdaSubst from source

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

2.1.1 Prerequisites

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

- A GNAT compiler, any version. Note that the compiler must also be available on the machine in order to run AdaSubst (all ASIS application need the compiler).
- ASIS for GNAT
- The GNATColl component

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

2.1.2 Build with installer (Windows)

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

2.1.3 Build with project file

Simply go to the `src` directory and type:

```
gnatmake -Pbuild.gpr
```

You're done!

2.2 Installing AdaSubst

All you need to run AdaSubst is the executable named `adasubst` under Linux or `adasubst.exe` under Windows.

If you downloaded the Windows installer executable version of AdaSubst, simply run `adasubst_exe-setup.exe`. This will install all the files in the recommended locations (as has been done with the Windows installer source version).

If you built AdaSubst from source without an installer, the executables are in the `src` directory of the distribution. If you downloaded an executable distribution, they are in the root directory of the distribution. AdaSubst needs no setup, no special rights. Simply copy the executables to any convenient directory on your path; a good place, for example, is in the `bin` directory of your GNAT installation.

This version has been tested under Windows up to version 10 and Linux. There is nothing system-dependent in AdaSubst, and it should be easily portable to any operating system.

3 Program usage

AdaSubst features several functions that fall in two categories:

- Main functions that perform substitutions on Ada source files: Translate, Unuse, Unrepresent
- Auxiliary functions that perform secondary tasks: Help, Dependents, Prepare

The syntax for invoking AdaSubst is:

```
adasubst <function> [<options>] [<dictionary>] [<sources>]
         [-- <ASIS-options>]
<function> ::= dependents  | help | prepare | translate |
               unrepresent | unuse
<sources>  ::= <unit-name>{+|-<unit-name>}|@<list-file>
```

<function> is the function to be performed. Functions are detailed below.

<dictionary> is present only for the Translate function, see below.

<sources> gives the units to be translated. <unit-name> is the name of the Ada unit to translate; note that it is a unit name, not a file name: case is not significant, and there should be no extension! Of course, child units and subunits 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. If a unit is preceded by "-", it means that the unit (and possible child units) is *not* to be processed; this is especially useful in conjunction with the "-r" (recursive) option in order not to analyze some units, for example:

```
adasubst unuse -r Asis_Application-asis-a4g
```

will process the unit "Asis_Application" and all the units it depends on, except those in the "asis" and "a4g" hierarchies.

Alternatively, you can give an "@" followed by the name of a file instead of unit names. This file should contain a list of unit names, 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 the units that you are processing reference other units whose source is not in the same directory, AdaSubst needs to know how to access these units (as GNAT would). You can do this in two ways (not exclusive):

1. You can specify a project file (".gpr") with the "-p" option. AdaSubst will automatically consider all the directories mentionned in all "source_dirs" specified in the project file or one of the projects it depends on (directly or indirectly).

   Alternatively, an old emacs project file (the file with a ".adp" extension used by the Ada mode of Emacs and older versions of AdaSubst) can also be specified with the "-p" option. AdaSubst will consider all the directories mentioned in "src_dir" lines from the project file.
2. you can 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. Note that since "-I" is an ASIS option, it must appear after a "--".

Everything that appears after "--" will be treated as an ASIS option, as described in the ASIS user manual. Normally, you don't have to care about this, except for the previous issue.

Options can be grouped or provided separately (i.e. "-b -s" is the same as "-bs"). Options can appear at any place on the command line. Note that some options may not apply to some functions.

```
<options> ::= -bcdrsvw
              -oO <output-prefix>
              -p <project-file>
              -l <line-length>
```

-b	process body of unit
-c	Comment modified lines (rather than remove). Every changed line is kept in the output file, as a comment line starting with `"--CHANGED: "`. This makes it handy to retrieve changed lines with an editor, and to check the substitutions.
-d	debug. Internal debugging switch, adds extra messages for debugging purposes. Casual users have no reason to use it, unless you encounter a bug in AdaSubst.
-l	limits output line length to the given value. See below.
-o	specifies an output file, or a prefix for output files. See below.
-O	same as -o, but keep output files that have no changes.
-p	use source directory indications from provided project file. See above.
-r	recursive. Process the unit and (recursively) all user units it depends on (including parent units if the unit is a child or a subunit). Predefined Ada units and units belonging to the compiler's run-time library are never processed.
-s	process specification of unit
-v	verbose mode: print extra messages, including the name of each unit as it is being processed.
-w	overwrite output file if already existing

It may happen that a modified line is longer than the original, and therefore exceeds the maximum line length accepted by the compiler. The "-l" option sets an upper bound to the length of the output lines. If a line becomes longer than that, it is folded at an appropriate point. By convention, if the given <line-length> is zero (or if the "-l" option appears without a value), it means that lines should never be folded. If no "-l" option is given, the maximum line length is 200 (the minimum required by the Ada standard). A line length less than 80 is not accepted.

For the main functions, AdaSubst will output the result of processing each unit to a file whose name is obtained by prefixing the <output-prefix> of the "-o" option to the original name of the corresponding source file. The <output-prefix> can be any string, and is not analyzed by AdaSubst. A prefix like "result/" will result in all the output going to the directory "result", with the same name as the original. Alternatively, a prefix like "new-" will result in all output files being in the same directory, with a "new-" prepended to the name. AdaSubst will refuse to overwrite an existing file on output, unless the "-w" option is given. If no substitution were performed on a unit, the output file is not kept, unless you specified the <output-prefix> with the "-O" (capital O) option (i.e. with "-o" only modified files are generated, while with "-O" a complete copy of indicated units is generated, whether there are substitutions or not). If no `"-o <output-prefix>"` option is given, the result of the substitutions is simply written to the standard output. This is useful to check the substitutions, or if you want to have all the output in a single file. With Gnat, this file could be gnatchopped to rebuild the correct source file names.

For auxiliary functions, the “-o” option simply gives the name of the generated file. Without this option, the generated file is produced on standard output.

If neither -s or -b is given, -sb is assumed (i.e. both the specification and body of the given units are processed).

4 Instantiate Function

This function replaces all instantiations of generics by equivalent, explicit code.

This can be useful if, for example, your safety constraints ban the use of any "hidden" code. Another use case is when you use an external tool that misbehaves on generic instantiations.

4.1 Syntax

```
adasubst instantiate [<options>] <sources>}
         [-- <ASIS-options>]
```

4.2 Action

This function performs "in place" instantiation of generics. It makes its best efforts to preserve the exact semantics of generics, including for visibility rules to make sure that everything that should be visible stays visible, and that nothing becomes visible that shouldn't. There are however a few cases where this is not 100% possible; see "Caveat" below.

Instantiations of generics from the standard Ada library (i.e. those defined in the ARM) are *not* replaced. The reason is that their source may not be available, and any tool should be able to handle these.

The generics themselves are removed from the produced code, and any **with** clause that references a generic is also removed. On the other hand, **with** clauses from the generic itself are added to the unit containing the instantiation, since the replacement code will need them. In short, generics completely disappear from the generated code, but the resulting code should compile normally (if it doesn't, please tell us!)

4.2.1 Replacement technique

Generic instantiations are replaced by a number of renaming declarations, subtype declarations, and constant declarations for binding formal parameters to corresponding actual parameters. The generic unit is then copied as-is. To ensure proper visibility, this is achieved inside a wrapper package. A renaming declaration, outside of this package, provides visibility to the instantiated unit. This process is pictured by the following example. Given the generic unit:

```
generic
   type T is private;
   Init : T;
   with function "+" (L, R : T) return T;
procedure Proc (A, B : in out T);

procedure Proc (A, B : in out T) is
begin
   A := A + B;
   B := Init;
end Proc;
```

and the following program:

```
with Proc;
```

```
procedure Main is
   procedure Inst is new Proc (Integer, 0, "*");
   V1, V2 : Integer;
begin
   V1 := 5;
   V2 := 10;
   Inst (V1, V2);
end Main;
```

After substitution, the program becomes:

```
procedure Main is
   package Inst_Instantiation_Wrapper is
      subtype T is Integer;
      Init : constant T := 0;
      function "+" (L, R : T) return T renames "*";
      procedure Proc (A, B : in out T);
   end Inst_Instantiation_Wrapper;
   package body Inst_Instantiation_Wrapper is
      procedure Proc (A, B : in out T) is
      begin
         A := A + B;
         B := Init;
      end ;
   end Inst_Instantiation_Wrapper;

   procedure Inst (A, B : in out Inst_Instantiation_Wrapper.T) renames Inst_Instantiat

   V1, V2 : Integer;
begin
   V1 := 5;
   V2 := 10;
   Inst (V1, V2);
end Main;
```

This transformation is very close to what is used by the compiler to process generic instantiations, except for the extra wrapper package. Of course, this can create conflicts if the program contained an entity whose name would be the same as the one of the wrapper package; however, this name is constructed by appending “`_Instantiation_Wrapper`” to the name of the instantiated unit. It is highly unlikely that such a name be already used in the rest of the program.

Of course, the same process is recursively applied to the copy of the generic, to make sure that instantiations nested inside generics are handled properly.

4.2.2 Caveat

If a unit is a library level instantiation, the transformation will create three units in the same file (the wrapper specification, the wrapper body, and the library level renaming). If you intend to use the GNAT compiler, running Gnatchop on the output file is necessary.

As of this version, the presentation of the generated code is not well polished; using a pretty-printer on the output is recommended.

There is one case where the exact semantics of instantiation is not preserved: when an instantiation occurs within a package specification. In this case, the Ada instantiation creates the specification *and body* of the instantiated unit at the place of instantiation, inside the enclosing package specification. But in regular Ada code, you cannot have a body inside of a package specification. The wrapper body is therefore moved to the beginning of the body of the enclosing package (and the body is automatically created if there was none). Regular code should not show any difference in behaviour, but it is definitely possible to construct a nasty generic whose body elaboration heavily depends on side effects, and whose expansion does not behave the same as the original instantiation. We suggest you reconsider how your generic is written if this ever happens to you...

If a generic is declared inside a package and uses global elements (like a global variable) from the body of the package, these elements won't be visible from the outside, and especially from the instantiated unit. Therefore, the generated code won't compile (hence there is no risk). If this happens, move the accessed elements to the specification, or provide an exported subprogram to access the element.

Finally, there might be cases where some easy manual adjustment are necessary. For example, if a library package contains only declarations of generic units, it will become an empty package, and its body will become empty too. But an empty package specification does not allow a body (even an empty one)! You will have therefore to remove the unnecessary body yourself.

5 Translate Function

This function translates identifiers according to a *dictionary*. The dictionary specifies new names and/or new locations where the identifier is declared. Names and visibility control clauses (like **use** clauses) are modified accordingly.

5.1 Syntax

```
adasubst translate <options> <dictionary> <sources>
         [-- <ASIS-options>]
<dictionary> ::=  <dictionary-file>|<substitution>
```

5.2 Action

<dictionary-file> is the name of the file to be used as the dictionary. If it is given as the single character `"-"`, the dictionary is read from the standard input. Alternatively, you can give one substitution as a quoted string instead of a dictionary file; this avoids having to write a dictionary file for simple substitutions. For example:

```
adasubst translate "pack.x => A_Better_Name" My_Package
```

5.3 Dictionary structure

The *dictionary* file defines the substitutions to be performed. Note that source comments, character literals, character strings, and attribute designators are never substituted.

The dictionary file is a regular text file defining substitutions, one per line. The logic behind the format of the dictionary is:

- if you change the name of en entity, you just give the new (simple) name for the entity in the dictionary.
- if you move an entity to another place, you specify the new location of the entity.

Each line has one of the following formats:(elements between quotes must be typed as is):

```
<original> "=>" <substitution>
"not" <original>
```

<original> defines what kind of element is to be substituted (the *original*). Since <original> is an Ada element, case is irrelevant, and you can even put spaces where they are allowed by Ada syntax. Only one substitution is allowed for a given <original>.

In the first form, <substitution> describes how you want <original> to be modified. Note that the replacement is always taken "as is", i.e. the replacement uses the same casing as given in the substitution string.

The second form is an indication that the corresponding <original> is *not* to be substituted. This is useful to make exceptions to a more general substitution rule. See [Substitution with exceptions], page 17, for examples of this.

In addition, empty lines and lines starting with “`#`” or “`--`” (comment lines) are ignored.

5.3.1 Specifying the original name

The syntax of <original> is as follows:

```
<original> ::= <Full_Name> | "all" <Simple_Name>
```

<Full_Name> is the *full* name of the element to be substituted, using normal Ada dot notation (with some extension, see [Overloaded names], page 12). *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. 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, see [Overloaded names], page 12. No qualification is allowed.

The first form designates a single element from an Ada program, and is therefore called "element substitution". Only this element will be substituted, i.e. if you want to replace the variable "V" declared in package "P", the original should be "P.V" , and no other "V" in the program will be substituted. The second form designates all identifiers with the given name in the program, irrespectively of where they appear, i.e. if you use the form "all V", all elements named "V" will be substituted. This form is called "identifier substitution".

5.3.1.1 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}`).

5.3.1.2 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}`

5.3.1.3 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`.

5.3.1.4 Record and protected types components

You can designate the name of a record or protected type component (a "component" 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`.

5.3.1.5 Operators

AdaSubst 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}`.

There is no problem in changing a regular function name into an operator, but the substitution will continue to use the prefixed call notation. If you do the opposite, i.e. changing an operator into a regular function, AdaSubst will automatically transform any infixed call into a prefixed call. In short, if your dictionary says:

```
Plus => "+"
"-"  => Minus
```

the following sequence:

```
X := Plus (X, Y);
X := X - Y;
X := "-" (X, Y);
```

will become:

```
X := "+" (X, Y);
X := Minus (X, Y);
X := Minus (X, Y);
```

Note finally that `"and then"`, `"or else"`, `"in"` and `"not in"` are not operators (although they are operations) and cannot therefore be substituted. Similarly, operators between universal types have specifications that cannot be expressed in Ada, and therefore cannot be substituted.

5.3.2 Specifying the substitution

The *substitution* can take one of the following forms:

```
["abs"] <Name>
[<Name>] "in" <Package_Name>
<Name> { [","] <Name> }
```

<Name> is the new name(s) of the entity, <Package_Name> is the place where it is now (if it has been moved). The idea is that the substitution describes how the element has been changed. If you simply changed the name of an entity, just give the new name. If you moved an entity into another package, just give "in" followed by the new package name. Of course, if you moved an entity and renamed it, you can give both.

If the substitution starts with an "abs", the element is replaced exactly by the <Name> (we call this an *absolute* substitution). Note that the element is replaced, irrespectively of how it was originally named (i.e. using a short name or a qualified name). This allows you to change short names into long names, i.e. if the element is "Pack.V" and the substitution is "abs Pack.V", then every occurrence of the former (even in short notation) will be changed into the full notation.

As a special case, you may provide several substitions for an element in the dictionary, separated by spaces or a comma (the third form). Such substitutions are always absolute. This is intended for the case when one element has been replaced by several ones, typically when a package has been split into several packages. For the purpose of the explanation, assume that package `Pack` has been split into `Pack1` and `Pack2`. The dictionary will contain a line like "`Pack => Pack1, Pack2`". Normally, you should specify explicitely in the dictionary where each element of Pack has been moved (see Section 8.3 [Prepare], page 21, for a mode that helps you do that). When AdaSubst encounters a reference to `Pack` for which there is no explicit substitution, two things can happen:

1. If Pack appears by itself in a with or use clause, it is replaced by the list of substitutions (i.e., "`with Pack`" becomes "**with** `Pack1, Pack2`"). This may create unnecessary references, but causes no harm.
2. If Pack appears as the prefix of some elements, it is replaced by the list of substitutions, with "?" as the separator (i.e. "`Pack.V`" becomes "`Pack1? Pack2.V`"), and a warning message is issued. Since there is no case in Ada where a "?" is legal, you will be noticed at the first compilation of the problem, and it will be easy for you to fix it manually, or to adjust the dictionary as appropriate.

For example, given a dictionary file containing the following lines:

```
Pack      => Parent, Parent.Child
Pack.X    => New_X in Parent.Child
Pack.V    => abs Variable
Pack.Z    => Child.Z in Parent
```

Here is how the substitutions will happen:

Original

```
with Pack;
procedure Example is
   A : Integer := Pack.X
   B : Integer := Pack.Y;
   use Pack;
begin
   A := X + Y + V;
   B := Pack.V;
   B := Pack.Z + Z
end Example;
```

Substituted

```
with Parent, Parent.Child;
procedure Example is
   A : Integer := Parent.Child.New_X
   B : Integer := Parent? Parent.Child.Y;
   use Parent, Parent.Child;
begin
   A := New_X + Y + Variable;
   B := Variable;
   B := Parent.Child.Z + Child.Z;
end Example;
```

Notes:

- if you move an enumerated type to a different package, you must put in the dictionary the new place of the type (as usual), but also the new place for each enumeration literal, since the enumerated type also declares the values. Of course, the dictionary generated by the "prepare" function of AdaSubst includes the declaration of all enumerated litterals (see Section 8.3 [Prepare], page 21).
- There is no reason why you would want to have an `"in <Package_Name>"` in the substitution when the original is the name of a record (or protected type) *component*. If you do it anyway, it is ignored.

5.3.3 Preference rules

It may happen that several substitutions are possible for a given element. In this case, the rule is to use the most specific interpretation. This means that substitutions are considered in the following order:

- Overloaded elements substitution
- Non-overloaded elements substitution
- Overloaded identifier substitution
- Non-overloaded identifier substitution.

The exact rule for substitution is that if an element has a substitution, then that substitution is applied; otherwise, if the element's name was given using a qualified notation, a substitution is attempted on the prefix (i.e. everything before the last dot), using this same rule.

For example, you may specify to replace `"Pack"` with `"My_Package"`, and `"Pack.X"` with `"Other_Package.X"`. In this case, the most specific substitution is applied (the later), but if you have a `"Pack.Y"` (no specific substitution specified), it will be replaced by `"My_Package.Y"`.

5.4 Examples

The "translate" function can be used for various purposes. Here are some typical use cases.

5.4.1 Changing identifiers

You simply want to change every occurrence of an identifier, irrespectively of where it appears. For example, you misspelled an identifer. Just make a dictionary like this :

```
Wrong_Spelling => Right_Spelling
```

and run AdaSubst on your program.

If you don't want to create a dictionary file, you can also use the following command:

```
echo Wrong_Spelling => Right_Spelling | AdaSubst - main_program -r
```

Note that here, we run AdaSubst on the main program with the "-r" option to make sure that every occurrence of "Wrong_Spelling" in the whole program is replaced by "Right_Spelling". You can also avoid the "echo" command, and just type the dictionary line on the keyboard, don't forget to input the end-of-file in that case.

The following case illustrates another use of absolute substitutions. Imagine you had the following package, from an Ada83 project:

```
package Missing_Ada83_Functions is
   function Min (L, R : Integer) return Integer;
   function Max (L, R : Integer) return Integer;
   -- etc.
end Missing_Ada83_Functions;
```

Now, you'd rather use the Ada95 'Min and 'Max attributes. Make a dictionary like this:

```
Missing_Ada83_Functions.Min    =>  abs Integer'Min
Missing_Ada83_Functions.Min.L  =>  Left
Missing_Ada83_Functions.Min.R  =>  Right
Missing_Ada83_Functions.Max    =>  abs Integer'Max
Missing_Ada83_Functions.Max.L  =>  Left
Missing_Ada83_Functions.Max.R  =>  Right
```

Note the use of "`abs`". Therefore, even if the original was written as "Missing_Ada83_Functions.Min", it will be replaced by "Integer'Min", not "Missing_Ada83_Functions.Integer'Min". Note also the translation for the formal parameter names, to ensure correct translation of calls using named notation.

5.4.2 Substitution with exceptions

As another example, suppose you defined several "Put" operations for types defined in various packages, but someone decides that they should rather be called "Print". Of course, the substitution should *not* be applied to the "Put" defined in Ada.Text_IO. The following dictionary will do the trick:

```
all Put => Print
not Ada.Text_IO.Put
```

Overloaded names can be used with all forms of substitution. If you decide in addition that all "Put" on type My_Int should be called "Print_Int" and that all "Put" on type My_Float should stay the same, the dictionary would become (assuming that the types are defined in package "My_Data"):

```
all Put => Print
not Ada.Text_IO.Put
all Put {My_Data.My_Int} => Print_Int
not all Put {My_Data.My_Float}
```

5.4.3 Splitting a package into child units

Here you have a big package (possibly from an old Ada83 project) and you want to take advantage of the hierarchical units feature of Ada95 to split it into several child units.

1. run AdaSubst with the "prepare" function on the package. This will prepare an identity dictionnary file.
2. Rearrange the package at your convenience. Each time you move an element from the visible part of the old package, adjust the corresponding line of the dictionary to reflect the new name. Note that you can change the name of the element at this time. Imagine for example the following package:

   ```
   package Old is
      Ident1 : Integer;
      Ident2 : Integer;
   end Old;
   ```

   Running "adasubst prepare" on this package will produce:

   ```
   Old.Ident1   =>  Ident1
   Old.Ident2   =>  Ident2
   ```

   If you want to change this package into the following:

   ```
   package Parent is
      Good_Variable_Name : Integer;
   end Parent;

   package Parent.Child is
      Another_Good_Name : Integer;
   end Parent.Child;
   ```

   change the dictionary to:

   ```
   Old         =>  Parent, Parent.Child
   Old.Ident1  =>  Good_Variable_Name in Parent
   Old.Ident2  =>  Another_Good_Name in Parent.Child
   ```

   The first line expresses that package "Old" has been replaced by "Parent" and "Parent.Child". Any "with" or "use" for "Old" will be replaced with the corresponding clause for both packages. Alternatively, you could use the following dictionary:

   ```
   Old         =>  Parent
   Old.Ident1  =>  Good_Variable_Name in Parent
   Old.Ident2  =>  Child.Another_Good_Name in Parent
   ```

   Here, "Old" would be systematically replaced by "Parent" and any use of "Ident2" would be replaced by "Child.Another_Good_Name", therefore producing also a correct code.

   Run AdaSubst on all units that used the old package with this dictionary. You can also run AdaSubst with the same dictionary on the body of the packages you have modified, all required modifications will be performed automatically.

NB: if you plan to split a package into child units, it is often useful to know which parts of the package are used by various modules. Adadep, another utility provided with AdaSubst, can be instrumental for that.

6 Unuse Function

This function removes (or comments out) all **use** clauses from a set of units. Identifiers that were visible through **use** clauses are changed to full (dotted) notation.

This allows you to remove safely all **use** clauses, for example if you have a software component that must be used in a program whose coding standard forbids the use of **use** clauses.

6.1 Syntax

```
adasubst unuse [<options>] <sources>}
         [-- <ASIS-options>]
```

In addition to regular options, the -u option is used to keep **use type** and **use all type** clauses.

6.2 Action

This function produces output files where all **use** (and if no "-u" option, all **use type**, and **use all type**) clauses are removed (or commented out if the "-c" option is given).

If no "-u" option is given, all names that were visible because of the **use**, **use type**, and **use all type** clauses are changed to full name notation. Operator calls are changed to prefix calls, i.e.:

```
declare
   use Pack;
begin
   X := A+B;  -- "+" declared in Pack
end;
```

is changed to:

```
declare

begin
   X := Pack."+" (A, B);  -- "+" declared in Pack
 end;
```

If the "-u" option is given, operators (and in the case of **use all type** primitive operations and enumeration litterals) are not substituted.

7 Unrepresent Function

This function removes (or comments out) all representation clauses from a set of units. This can be useful if you want to assess the impact of representation clauses on your program, especially efficiency. Thanks to AdaSubst, you can easily compare two versions of your program, with or without representation clauses.

7.1 Syntax

```
adasubst unrepresent [<options>] <unit-name>{+|-<unit-name>} | @<list-file>
         [-- <ASIS-options>]
```

7.2 Action

This function produces output files where all representation clauses are removed (or commented out if the `"-c"` option is given).

8 Auxiliary Functions

In addition to the main substitution functions, AdaSubst offers some auxiliary functions to make its use more user-friendly.

8.1 Help

8.1.1 Syntax

```
adasubst help
```

8.1.2 Action

AdaSubst just prints a brief usage summary and exits. As a convenience to the user, this mode is also activated if the -h option is given on the command line.

Any parameter or option is ignored.

8.2 Dependents

This function helps you prepare the list of files you want to process by finding all the dependents of a set of units.

8.2.1 Syntax

```
adasubst dependents [<options>] <sources> [-- <ASIS-options>]
```

`"-c"` and `"-l"` options which would mean nothing for this function are not accepted. The `"-r"` option is assumed by default; it is accepted on the command line, but has no effect. The `"-o <output-file>"` option specifies the name of the file to which output is to be written (standard output if no `"-o"` option is given). Note that if an <output-file> is specified that already exists, new data is appended to it, unless the -w option is given (in which case it is overridden).

8.2.2 Action

AdaSubst will scan all units given by <unit-name> (or the names of packages given in <list-file>), and produce a list of the unit names, and (transitively) all units that are withed by these units, except predefined units. Especially, if you give it the name of your main program, this will build the list of all units that make up the program.

The resulting file is appropriate to use as an indirect file ("@<list-file>") for the main functions. Note that since you really need to perform this function only once (unless new units are added to your program), it is faster to generate the list once and use it than to specify the "-r" option each time adasubst is run.

8.3 Prepare

This function is intended to help you prepare the dictionary file of the "translate" function when you want to split elements exported by a package into several other packages. This happens quite often when you start writing a package, then later decide to split it into child units.

8.3.1 Syntax

```
adasubst prepare <options> <sources> [-- <ASIS-options>]
```

"-c" and "-l" options, which would mean nothing, and are not accepted. The "-o <output-file>" option specifies the name of the file to which output is to be written (standard output if there is no "-o" option). Note that if an <output-file> is specified that already exists, new data is appended to it, unless the "-w" option is given (in which case it is overridden).

8.3.2 Action

AdaSubst will analyze the visible part (not the private part) of the units given by <sources>, possibly recursively if a "-r" option is given), and produce a dictionary consisting of the full (overloaded) name of all elements declared within the visible parts of (generic) package units (others are ignored), followed by the simple identifier of the same element. Of course, if the visible part includes package, task or protected declarations, only their visible parts will be processed. In short, given the package:

```
package Pack is
   type T is range 0..100;
   E : exception;
   V : T;
   function F (X : Integer) return Float;
   task A_Task is
      entry Visible;
   private
      entry Not_Visible;
   end A_Task;
private
   Hidden : Integer;
end Pack;
```

AdaSubst will produce a file containing:

```
#
# Elements declared in package PACK
#
Pack.T                              => T
Pack.E                              => E
Pack.V                              => V
Pack.F{Standard.Integer return Standard.Float} => F
Pack.F{Standard.Integer return Standard.Float}.X => X
Pack.A_Task                         => A_Task
Pack.A_Task.Visible{}               => Visible
```

Assuming that you have moved elements declared in Pack to other packages, all you have to do is take your favorite text editor and add after each identifier from the second column an "in" plus the name of the package where it is now. You can then use this dictionary to update all units that depended on Pack.

Note that the generated dictionary is a "no-op", i.e. each element is substituted by itself. AdaSubst recognizes no-ops, so that lines that contain identical substitutions are not considered modifications (no "–CHANGED:" line is generated if you put the `"-c"` option). Therefore, if you just want to change some names of elements declared in a package, you can use the `"prepare"` function to generate the dictionary, then modify only the lines for the identifiers you want to change. You can then run AdaSubst in normal mode on the package itself as well as on any unit that uses the package to perform the substitutions.

Note also that if the original element does not use the same casing as the substitution, substitution does occur. This means that the `"raw"` dictionary can be used to change the casing of all identifiers to the one of the original declaration in the package specification.

9 AdaSubst and ASIS

AdaSubst has been tested only with ASIS-for-gnat; however, the only (known) dependency is for the implementation-dependent parameters used to open an ASIS context. These parameters are defined as strings in the package Implementation_Options (file implementation_options.ads), so all you have to do if you want to port AdaSubst to another implementation is adjust these definitions. If you ever do so, please keep us informed by sending a note to rosen@adalog.fr.

10 Troubleshooting

10.1 Message: Inconsistent tree file for ..., please remove .adt file

It is a consequence of the way ASIS works that tree files (with a .adt extension) are created as part of the process. Currently, these file are not erased by AdaSubst after execution. This speeds up execution if you run AdaSubst several times, as is often the case when you are designing the dictionary file. However, if you change your source files, the trees are no more consistent with the sources. Simply remove the adt files and rerun AdaSubst; new tree files will be created.

10.2 AdaSubst prints `"!! Internal error, continuing"`, or halts with an error

There are many subtile special cases when analysing Ada code, so it may happen that you find a context that we didn't think about. There are also some cases where the error comes from a bug in ASIS-for-Gnat. Normally, AdaSubst will print `"`!! internal error, continuing`"` and will try to proceed, without processing the failing element. It will output a `"` ???? `"` string in the output file (something that will certainly not compile), for you to check manually if a substitution has been missed. Since such a case is very rare in practice, a bug will not prevent you from using AdaSubst. Note however that if you run AdaSubst with the `"`-d`"` (debug) option, it will halt on the failing element and print the full ASIS context.

If you encounter a bug while using AdaSubst , please rerun it with the `"`-d`"` option and save the output to a file, then send this file together with the dictionary and the file you are processing to rosen@adalog.fr. We will do our best to fix it. Of course, you are welcome to try and fix it yourself; we made every attempt to make the source of AdaSubst quite readable, but be aware that you will need a good understanding of ASIS and Ada syntax to understand how it works.

10.3 AdaSubst is slow when run on many files

There is one slow operation in AdaSubst, known as `"`opening the context`"` in ASIS. This happens only once per run - but happens in every run. This means that if you use a shell script to run AdaSubst multiple times, with one file at a time, it will take you quite a while. On the other hand, if you provide a list of files, or provide several files at the same time on the command line, or just provide a main program with the `"`-r`"` option, AdaSubst will be much faster (some users have reported going from several hours to a couple of minutes!), since the context is opened only once.

10.4 AdaSubst Translate complains that all identifiers are doubly defined

Look at your dictionary: they are actually doubly defined. Remember that if you run the “prepare” function of AdaSubst and the output dictionary exists, new elements are appended to it (unless you use the `"`-w`"` option). Therefore, if you run by mistake twice the same `"`AdaSubst prepare`"` command, all definitions will appear twice in the dictionary.

11 Known bugs

11.1 Derived primitive operations

In the case of cascaded derivations with several substitutions, there is a case where AdaSubst does not behave as expected. Consider:

```
package Pack is
   type T is ...;
   procedure Primitive (X : T);

   type D1 is new T;

   type D2 is new D1;
end Pack;
```

with the following dictionary:

```
Pack.Primitive{Pack.T}  => T_Primitive
Pack.Primitive{Pack.D1} => D1_Primitive
```

We give different substitutions for "Primitive" on "T" and "D1", but we do not specify the substitution for "D2". Since D2 derives from D1, any occurrence of a call to "Primitive" with a "D2" parameter should be substituted as a "D1_Primitive", however it will be substituted as a "T_Primitive". In other words, the implicit substitution uses the ultimate ancestor, not the direct ancestor. This is due to a bug in Asis-for-Gnat, so there is not much we can do until the bug is fixed.

11.2 Terminating identifiers

There is one very special case of a terminating identifier which is not handled properly by AdaSubst:

- IF you process a child unit for which (part of) the name must be substituted
- AND IF the name is repeated at the end of the unit
- AND IF there are separators between the elements of the closing name

THEN the closing name will not be substituted at all. In short, if you write:

```
procedure Parent.Child is -- Never a problem here
   ...
end Parent  -- Crazy comment
    .       -- and crazy way of writing
    Child   -- an identifier
;
```

substitution will not occur on the final name. It is unlikely that we fix this bug in any foreseeable future since:

- (believe it or not) it is extremely difficult to handle.
- there is no risk associated with it: since the substitution is performed correctly on the beginning identifier (even if written the crazy way), the closing identifier will not match, and the unit will simply not compile. It is then easy to fix it by hand.

- this is such an unusual and strange programming style that the best advice is certainly to fix the source and write the identifier normally.

12 Future of AdaSubst

The structure of this programs allows it to be extended very easily. Here is a brief overview of our to-do list:

- More sophisticated processing of renamings
- Provide substitution of subparts of identifiers (i.e. change all "foo" into names of the form "*_foo_*" into "bar").
- Other kinds of substitutions
- and more...

Send good ideas, bug reports, suggestions of improvements to the program or to the documentation, etc. to J-P. Rosen).

Commercial support is available for this product; please refer to the file `support.txt` in the doc directory. For more information, or if you need any support or assistance for your Ada projects, please visit our Web site (`http://www.adalog.fr/en/`) or send us a mail at info@adalog.fr.