GtkAda: Design and Implementation of a High Level Binding in Ada

Emmanuel Briot[1], Joël Brobecker[2], and Arnaud Charlet[1]

[1] ACT Europe, 8 rue de Milan, 75009 Paris, France
[2] 218-1741 West 10^{th} avenue, V6J 2A5 Vancouver B.C., Canada

Abstract. The purpose of this paper is to describe the design and implementation choices that were made during the development of an Ada binding to the popular Gtk+ graphical toolkit. We concentrate on the methods used to interface between C and Ada, but many topics described in this paper are not tied to Ada and can be applied to other high level languages that need to interface with existing libraries. We also describe the various mechanisms developed to provide a powerful GUI builder with GtkAda.

This paper emphasizes the added value that Ada brings to the task of writing a high level binding over an existing library.

1 Introduction to Gtk+

Gtk+ is a powerful graphical toolkit originally developed as part of the GIMP [1] (the *GNU Image Manipulation Program*). Since Gtk+ had been designed to be cleanly separated from the rest of the GIMP code, and given the success of the toolkit, it became eventually obvious that the Gtk+ library should be made a stand-alone capability so that other applications could take advantage of it.

Thanks to its elegant Object-Oriented design, its efficiency, and its portability, this toolkit has become very popular and is being used in many applications. In particular, the GNOME [2] project (the *GNU Network Object Model Environment*) which intends to build both a complete, easy-to-use desktop environment for the user, and a powerful application framework for the software developer, has chosen Gtk+ as its graphical library.

Another important advantage of Gtk+ is that it was designed to make it easy to create bindings to it for various languages. Any construct that could make the binding task more complex, such as procedures with variable arguments for example, were mostly avoided. This explains why there are currently more than twenty different listed bindings based on Gtk+.

Gtk+ has thus become the graphical toolkit of choice. Along with its associated tools, such as code generators like GtkGlade [3], it offers a complete suite to develop graphical user interfaces in a portable way, and with a common look-and-feel.

2 The GtkAda Project

The GtkAda project started with the observation that there was no complete high level graphical library freely available for Ada. From this simple idea, we had to decide which library our toolkit would be based on. The obvious choices were Motif and Gtk+. Motif because it was a widely used and known standard; Gtk+ because of its free software status, its modern design, and its open architecture, which makes it easy to interface to it from other languages. The main drawbacks of Motif are the requirement to have an X server to run it, and its "old" interface that suffers some disadvantages that Gtk+ avoids, for example the absence of built-in support for multi-threaded applications.

2.1 Thin and Thick Bindings

A *thin binding* is a simple line-for-line translation that is usually done automatically or semi-automatically, to transform a foreign language interface into Ada specifications. Little attempt, if any, is made to introduce additional type safety and "real" Ada types, such as strings instead of null terminated character pointers, high level constructs such as tagged types, controlled types, etc.

The term *thick binding* usually relates to a library interface that goes beyond – sometimes far beyond – the simple task of mapping the foreign language types and routines, providing an API that is closer to what a similar library written directly in Ada would provide. Ideally, a good *thick binding* should "hide" the fact that it is using an external library internally, turning this into an "implementation detail".

This is where the term *thick binding* seems inadequate, because by providing a language-friendly interface and by adding extra functionalities, we are really making available a complete toolkit on its own. Because of this we like to refer to GtkAda as an *Ada Graphical Toolkit*.

3 How to Add Safety to a C Library

The first level of safety is brought by the Ada language itself. As opposed to C, Ada provides type safety via a strong typing policy which prevents the user from mixing inadvertently 2 different types. *Enumerated types* are also used to our advantage to replace C enums. In contrast to C, where it is possible to use an out-of-range value, Ada enforces at compile-time the usage of valid enumerations values only.

A central difficulty in binding to Gtk+ is the systematic usage of *C strings* (*char**), which leads to memory management problems, such as memory leaks, dangling pointers, etc. Instead, GtkAda uses the more natural type String, which allows the user to avoid all these problems. Converting an Ada String to C is achieved by simply appending ASCII.Nul to the Ada strings. Converting a C string to Ada presents additional problems. Although the conversion itself is

straightforward with the services provided in Interfaces.C.Strings, the memory management part has to be handled by GtkAda to avoid memory leaks.

A similar problem arises for arrays. Whereas passing an Ada array to a C function in a portable fashion is easy (give the address of the first element), getting an array back is trickier. For instance, it is in general not possible to ask the user to allocate an array, and pass it as an "out" parameter, because this would require knowledge of the size of the returned array. Likewise, unless one knows the internals of the implementation of arrays and their bounds in Ada, i.e. the way the structures are laid out, it is not possible to recreate a complete Ada array without copying the data element by element, which of course is inefficient.

The solution we have used is to provide "flat arrays", that have no real bounds (effectively they have the maximum possible bounds), as seen in figure 1. With the declarations given therein, a function that interfaces to the C library should return a System.Address, that can then be safely converted into a valid Ada array. Of course, the bounds information has to be read from elsewhere, generally as an out parameter of a C function, since it is not available within the new array. A `Points_Array` structure is created to closely associate the flat array with its real bounds.

```
type Gdouble_Array is array (Natural) of Gdouble;
type Gdouble_Array_Access is access all Gdouble_Array;
type Points_Array is record
   Points     : Gdouble_Array_Access;
   Num_Points : Gint := 0;
end record;

function To_Double_Array is new Unchecked_Conversion
    (System.Address, Gdouble_Array_Access);
```

Fig. 1. Converting from C arrays to Ada arrays

Tagged types provide another level of type safety, in the widget hierarchy of GtkAda. Fig. 2 shows the pattern followed for the declaration of each widget: each widget is declared as an access to a class-wide type. Most of the operations associated with each widget are primitive, thus making them dispatching operations. As a result, the library provides the user with all the advantages of class-wide programming.

Although Gtk+ follows a design that is fully Object-Oriented, the C language imposed some limitations on the implementation itself. Only type inheritance is provided, and the "methods" are not inherited. Therefore, there is no real polymorphism. As a consequence, the user is forced into the frequent use of *type-casts*, as shown in Fig. 3(a) where a `Gtk_Window` is converted into a `Gtk_Container` by calling the `GTK_CONTAINER()` macro. Experience shows that these type-casts are ubiquitous, increasing development costs and impairing the readability of

```
with Gtk.Object; use Gtk.Object;
package Gtk.Widget is
  type Gtk_Widget_Record is new Gtk_Object_Record with private
  type Gtk_Widget is access all Gtk_Widget_Record'Class;
  --  Followed by all the widget associated operations...
end Gtk.Widget;
```

Fig. 2. All widget declarations follow the same pattern

the resulting software. Moreover, as type-casting is a potentially unsafe operation, programming errors are likely to occur and lead to obscure software faults, which explains the introduction of these casting macros: besides doing a type cast, they also perform a run time type check.

On the other hand, Ada has been designed to provide such type safety in a totally transparent way: these problems were naturally solved in GtkAda with the usage of tagged types. Figure 3, where the same program is successively written in C and Ada, shows that using tagged types makes the GtkAda toolkit lighter to use and easier to read. The complexity of developing and maintaining the Ada program is therefore decreased.

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title (GTK_WINDOW (window), "reparent");
gtk_container_set_border_width (GTK_CONTAINER (window), 0);
```

(a) C implementation

```
Gtk_New (Window => Window, The_Type => Window_Toplevel);
Set_Title (Window => Window, Title => "reparent");
Set_Border_Width (Container => Window, Border_Width => 0);
```

(b) Equivalent Ada implementation

Fig. 3. The same code extract written both in C and Ada

Furthermore, through a careful use of generics, GtkAda provides at compile-time a type safety which was only available as run-time checks in the C library. As all other GUI toolkits do, GtkAda provides an callback mechanism (functions called whenever an event happens). But this mechanism is much more powerful than most, since its callbacks can have any number of arguments and can optionally return a value, depending on the type of event.

This multipurpose *callback mechanism*, which gives access to functions with variable argument lists, was probably the area where the type safety issue was the most critical. In Ada, this callback mechanism has been translated in a set of collaborating *generic packages* defined in Gtk.Handlers. The user needs to instantiate at least one of these packages for each of the possible profiles of the

callbacks, and then use the subprograms provided to connect the widget and the callback.

The idea behind these packages is that by default the argument list is transformed into an array, and passed as a single argument to the callback. However, for convenience, another set of packages, called marshallers, is provided to break this array into its components and then call the user handler.

Figure 4 shows how these generic packages can be used to connect a handler to an event generated when the user tries to close the window using the window decorations. The function `To_Marshaller` creates the marshaller transparently, as explained above, so that the user callback is a standard Ada function.

```
function My_Cb (Widget : access Gtk_Widget_Record'Class;
                Event : Gdk.Event.Gdk_Event)
                return Gint;
--  The function that needs to be called back

package Return_Widget_Cb is new Gtk.Handlers.Return_Callback
  (Widget_Type => Gtk.Widget.Gtk_Widget_Record,
   Return_Type => Gint);
--  Instantiation of a generic callback package

Return_Widget_Cb.Connect
  (Widget => W,
   Name => "delete_event",
   Marsh => Return_Widget_Cb.To_Marshaller (My_Cb'Access));
--  The compiler checks that My_Cb has the right signature
```

Fig. 4. How to use the generic packages provided by Gtk.Handlers

In combination with generics, *access to subprogram* types are also used systematically to provide additional strong typing. This feature is used, for example, to allow the user to pass in a simple way a self-defined action routine to an active iterator in Gtk.Ctree, as demonstrated in Fig. 5.

4 Benefits of the Object Oriented Features of Ada

As seen in the previous section, GtkAda is based on a tagged type hierarchy, as are most modern graphical libraries. This has a number of advantages, which will be explained in this section. We also show how this hierarchy is implemented over a non-object oriented C library.

4.1 Why objects are a natural way of representing GUI components

This section is not specific to GtkAda, but rather tries to explain briefly why having an object oriented design helps the user to better understand and use a li-

```
with Gtk.Ctree;
with Gtk.Style; use Gtk.Style;

package Access_To_Subprogram_Example is
   package Style_Row_Data is new Gtk.Ctree.Row_Data (Gtk_Style);

   procedure Set_Background
     (Ctree : access Gtk.Ctree.Gtk_Ctree_Record'Class;
      Node : in Gtk.Ctree.Gtk_Ctree_Node;
      Dummy : in Style_Row_Data.Data_Type_Access) is ... ;
   --  The procedure that will be passed to the active iterator

   procedure Run is
   begin
      Style_Row_Data.Pre_Recursive
        (Ctree => Ctree,
         Node  => Gtk.Ctree.Null_Ctree_Node,
         Func  => Set_Background'Access,
         Data  => null);
      --  Changes the background of all nodes of Ctree
   end Run;
end Access_To_Subprogram_Example;
```

Fig. 5. An access to subprogram used with an active iterator

brary. The arguments below are language-independent and domain-independent, although we have tried to explain their specific use in the case of GtkAda.

Tight integration between an object and its operations The concept of object-oriented programming is to provide both a structure that contains data, and some subprograms to manipulate that structure (called its primitive subprograms in Ada). Such a structure inherits its ancestors' attributes automatically, which makes it easier for the user to locate (and if necessary modify) the relevant subprograms.

For instance, a `Gtk_Button` is a special kind of widget that reacts to mouse clicks. Its parent is a `Gtk_Container`, which has the property of being able to have one child, and thus a button can also have one child which is displayed in the button (like a label, a pixmap, ...). It inherits all the behavior from a container, including the capability to add or remove a child, etc. But a `Gtk_Container` is also a `Gtk_Widget`, that can be displayed on the screen. Thus a `Gtk_Button` also inherits that capability.

Avoiding type casting Since the Ada runtime always knows what exact type an object has, there is no need to dynamically test it in explicit code. The appropriate subprogram is automatically called (this is called dynamic dispatching, or polymorphism).

For instance, this is useful when a widget is returned by one of the functions in GtkAda (for example when you extract a child from a container). You do not need to test what type the child has, since the runtime will automatically know what subprograms apply to it.

In the few cases where an explicit cast (view conversion) is required, Ada checks at run-time that the conversion is valid.

Easy extension and creation of new widgets The main advantage of object oriented programming is that you can use any type derived from a parent type wherever the library expects the parent type. This makes it almost trivial to extend existing widgets, either by adding your own internal data to them, or by modifying their behavior.

The whole GtkAda library is based on this principle: for instance, a toggle button is a simple button with two possible states: (active/inactive). Instead of rewriting everything to create a toggle button, we just specialize a few subprograms, and inherit the general behavior of a button.

Likewise, it is easy for the user to create his own widgets, either from scratch, or more simply by modifying an existing one. Instead of handling all the low-level details, the user just has to concentrate on the high level differences between the new widget and its parent.

4.2 Providing a tagged type hierarchy over a C library

Designing the hierarchy We have seen all the advantages that an object-oriented design provides. However, it is not always possible to create such a hierarchy over every C library. Fortunately for us, the design of Gtk+, the C library on which GtkAda is based, was already object-oriented, even if not programmed in a strict object-oriented language. This of course made the choice of objects and the design of the hierarchy much easier to do, whereas it is generally one of the hard things when designing from scratch. We will not discuss in this paper the different approaches (UML, HOOD, etc.) available for this task.

A well-designed binding must be easy to maintain and be kept up-to-date vis-a-vis of the underlying C library. It was thus very important for GtkAda to be as independent as possible from the C structures. We did not want to make visible the internals of these structures in the GtkAda interface.

All our objects contain a single private field that is a pointer to the C object (System.Address), and everything is done through that pointer. A couple of subprograms (Set and Get) must be provided for all the public fields of the C interface, and it is in fact sometimes a difficult task to guess whether a given field should be public or private. At this level, the Ada interface is thus much cleaner than the C one, in that the user does not have to read the source to understand whether he is allowed to use a field or not.

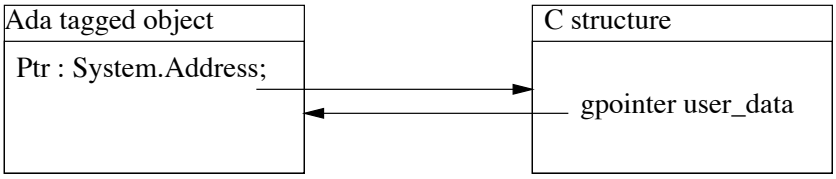
This method also makes the interface highly stable: if one field of the C structure is renamed, or removed in a future version of Gtk+, it is still possible to keep the same interface at the Ada level, but change the body of the subprograms.

As we said, accessing the fields is done through a functional interface. This requires an additional small C layer. Using that layer means that we do not have to map all the C structures to Ada records, thus solving the big portability issue of representation clauses.

Links between C and Ada objects One implementation problem is worth noticing. It is closely related to memory management (see the next section of this paper). Although it is easy to get the matching C structure given an Ada tagged object (just use the appropriate field of the record), it is sometimes more difficult to convert from a C object to the matching Ada record. This feature is required every time a subprogram (implemented in C in the underlying library) needs to return a widget.

Simply creating a new Ada structure is not possible, since we have to create it with the appropriate type (a C `GtkButton` is mapped to a `Gtk_Button`, not to the more general `Gtk_Widget`), and because the user might in fact have created his own tagged object by inheritance, we need to restore the exact value of the new fields.

This problem can not be solved in a general way for all the C libraries. Fortunately, this is made quite easy with Gtk+, since it is possible to associate some user data with every widget, through a keyword, like a hash-table. Thus, every time an object is created at the Ada level, we also create the C object at the same time, and set some user data to point back to the Ada object.



Of course, some widgets are not created by the user, but directly in some C code. For these widgets the user data is absent. In such cases, we simply invoke a more elaborate procedure that tests the C type against all its possible values, and creates the corresponding Ada type. This procedure is time-consuming, but is only executed rarely, since once we have created the new Ada object, it is permanently associated with the C widget.

This mechanism is made easy with Gtk+. In fact, it can easily be done for other foreign languages bindings through a global hash table that the Ada code would maintain.

5 Avoiding Memory Leaks

Library designers are always confronted to the problem of memory management: should the memory allocation and deallocation be done automatically, or should it be left to the user ? Although the first solution is obviously more user-friendly, it generally has some speed tradeoffs, and is not always possible to implement.

Ada offers a very powerful memory management mechanism : controlled types. Initialization and cleanup functions can be called automatically on objects of a controlled type. Although very comprehensive and fine-grained, this mechanism is heavy, and imposes a substantial overhead of function calls.

With GtkAda, leaving the memory management to the user is not an option. This is a difficult issue for GUIs, since it is not always obvious when a widget is still needed and when it can be destroyed. Moreover, when a container is destroyed, all its children must be destroyed as well.

Of course, the same problem appears in Gtk+. The authors of Gtk+ have thus implemented a reference counting mechanism, close to a garbage collector, that takes care of all memory management. This is implemented in the ancestor of all widgets, so that creating new widgets does not require special care.

Whenever a C widget is destroyed, it also frees the memory occupied by all its user data (as seen in the section 4.2). Since we can register our own destruction callbacks, this also allows us to free the memory allocated at the Ada level. Thus, GtkAda does not add any overhead over Gtk+.

This system of hooks or callbacks (functions called whenever specific actions are performed by a library) is extremely useful, both for the users of the library and for the people who want to adapt the library to their needs.

In a few cases it wasn't possible to use this mechanism. This is most notably the case for the low-level part of the library, which is very close to the X11 or Win32 protocols. These specific parts are carefully documented, since GtkAda does not provide any memory management for them. Using controlled types is not possible here, since most of these structures have to live in the X server itself, and thus cannot always be freed automatically when the subprogram exits. To avoid potential hidden memory leaks, some of the simplest structures have been directly mapped to Ada record (like points, rectangles,etc.) But there are still a few cases (pixmaps and graphic contexts most notably) where the user must take care of freeing the structure when done with them. Convenient functions are always provided for that usage.

6 Integrating GtkAda with a GUI Builder

GUI builders are one of the most desirable and popular applications built on top of graphical libraries. They are popular because they can automatically generate a major part of a user interface, simply by mouse interactions, or by describing at a high level the desired widget hierarchy. GUI builders obviate the need to write long, repetitious and stereotyped code.

6.1 Finding a suitable GUI Builder

Up to now, outside of the Windows world, GUI builders were rare and expensive, forcing many people to develop GUIs by hand. With the availability of a popular and open toolkit like Gtk+, many projects can develop GUI builders of their own. The most popular of these projects, called GtkGlade, had reached a sufficient

level of functionality when we started to look at a GUI builder solution for GtkAda that we could consider adopting it for Ada needs.

6.2 Generation of Ada code

Following the design philosophy of Gtk+, GtkGlade had been designed to easily support code generation for any language, by saving the widget hierarchy and properties in an easy-to-read and easy-to-parse XML file. Therefore, we decided to write an *XML-to-Ada* translator, specifically tailored for GtkGlade and GtkAda. During the design of this translator (called *Gate*), we had two choices: either write a completely independent code generator that would contain all the information and properties required for each widget, or enhance our existing GtkAda library by giving each widget the ability to generate the Ada code associated with it.

The second solution looked much more promising for several reasons. First, we had complete control over GtkAda anyway, so making changes in the library itself was not an issue. Second, it would take advantage of the already existing widget hierarchy, thus avoiding the need to recreate many of the needed structures. Finally, giving each widget the capability to generate the code related to its own properties is very natural and fits the object oriented model well. It also makes it easy to add support for new widgets. For example, suppose the user needs to add support for an extended implementation of `Gtk_Drawing_Area` (for instance a double buffer area), instead of having to write a complete support for all the data and properties of the ancestors of this new widget, he only needs to add code generation for the extended features, which is usually a simple task.

6.3 Clear and easy to read output

One common defect that we wanted to avoid in our translator, was the generation of Ada code that no human being could read and understand.

In order to output a clean code that would be easy to examine, we decided to generate a "composite widget" for each top level widget of the interface. A composite widget is an object that extends an existing type (typically a top level window) by adding a set of other widgets to it. This approach makes it very easy to see the group of widgets created, and offers a simple way to access all the fields.

Compared to other possible approaches, this one shows many advantages. If you consider for example the C code generated by GtkGlade, which generates local variables that are then stored in a hash table, this makes the generated code heavier and the user's code more complex.

Figures 6 and 7 provide a typical example of code generated by Gate and GtkGlade.

This example gives a good overview of the advantages that GtkAda brings to Gtk+:

```
type About_Dialog_Record is new Gtk_Dialog_Record with record
   Dialog_Vbox : Gtk_Vbox;
   Button : Gtk_Button;
end record;

type About_Dialog_Access is access all About_Dialog_Record'Class;

procedure Initialize (About_Dialog : access About_Dialog_Record'Class) is
begin
   Gtk.Dialog.Initialize (About_Dialog);
   Set_Title (About_Dialog, "About The Editor");
   Set_Policy (About_Dialog, True, True, False);
   Set_Position (About_Dialog, Win_Pos_Mouse);
   About_Dialog.Dialog_Vbox := Get_Vbox (About_Dialog);
   Set_Border_Width (About_Dialog.Dialog_Vbox, 2);
...
end Initialize;
```

Fig. 6. Sample Ada code generated by Gate

```
GtkWidget*
create_about_dialog ()
{
  GtkWidget *about_dialog;
  GtkWidget *dialog_vbox;
  GtkWidget *button;

  about_dialog = gtk_dialog_new ();
  gtk_widget_ref (about_dialog);
  gtk_object_set_data_full (GTK_OBJECT (about_dialog), "about_dialog", about_dialog,
                            (GtkDestroyNotify) gtk_widget_unref);
  gtk_window_set_title (GTK_WINDOW (about_dialog), _("About The Editor"));
  gtk_window_set_position (GTK_WINDOW (about_dialog), GTK_WIN_POS_MOUSE);
  gtk_window_set_policy (GTK_WINDOW (about_dialog), TRUE, TRUE, FALSE);

  dialog_vbox = GTK_DIALOG (about_dialog)->vbox;
  gtk_widget_ref (dialog_vbox);
  gtk_object_set_data_full (GTK_OBJECT (about_dialog), "dialog_vbox", dialog_vbox,
                            (GtkDestroyNotify) gtk_widget_unref);
  gtk_widget_show (dialog_vbox);
  gtk_container_set_border_width (GTK_CONTAINER (dialog_vbox), 2);
...
  return about_dialog;
}
```

Fig. 7. Sample C code generated by GtkGlade

- In order to ensure a clean memory deallocation, the C code duplicates for each subwidget the code to reference a widget (`gtk_widget_ref`) and registers a cleanup function (`gtk_object_set_data_full`). This is done automatically by GtkAda when doing a `Gtk_New`, as explained in section 5.
- With GtkAda, there is no need to associate artificially the widget children to their parent (using `gtk_object_set_data_full`), since this is done in a natural way by the type extension.
- There is no need to cast the widgets (see section 4.1).
- The widgets are typed instead of being anonymous pointers (`Gtk_Button` instead of `GtkWidget *`).
- The *use clauses* even if not mandatory, provide output that is lighter and easier-to-read.

6.4 Merging user's changes

The ability to go back and forth between the user interface and the generated code is a powerful feature that simplifies greatly the development of user interfaces.

Usually GUI builders approach this need by generating "read only" files and skeleton files that are automatically generated at first and subsequently modified manually. The read-only files contain code to build the widgets themselves, taking into account all their properties, while the skeleton files serve as templates where the user implements his callback subprograms. It is occasionally useful to modify some of the generated files themselves. This capability is usually not provided by GUI builders.

With Gate, we decided to implement both. The former because it seemed cleaner to separate the automatically generated construction code from the user's code. The latter because we want to provide the maximum flexibility.

In order to implement this capability, we decided to take a very simple but still powerful approach, based on a shell script and the "patch" and "diff" tools.

Here are the details of the operations performed by our script, to wrap our translator transparently for the user:

- Create the working directory: a different directory is created for each project file. This operation is usually done once.
- Compute a diff between the previously generated files (if any) and the current files (possibly modified).
- Generate the raw Ada files: this is a call to the real translator that does not have to worry about merging code and overriding files.
- Save the raw files as being the "previous files"
- Apply the diff to a copy of the raw Ada files.

Another possibility would have been to use *ASIS* during code generation, to be able to find the right procedures to modify, remove or add. But this would have required much more work than the simple-minded but powerful script described above. Furthermore, this approach would not work when the modified files do not compile properly, which can often be the case when the application is in the development phase.

7 Conclusion

As we explained, many of the choices we made can be applied to the construction of similar bindings for another language. However, it is clear that thanks to the use of some of the more powerful features of Ada (genericity, tagged types and access to subprograms), we have managed to build a high level binding to the Gtk+ library that provides compile-time checks, code readability, reusability and robustness.

Moreover, by integrating the various components of a graphical user interface, GtkAda now offers a complete solution for end-user applications written in Ada.

References

1. GIMP home page. http://www.gimp.org/.
2. GNOME home page. http://www.gnome.org/.
3. GLADE home page. http://glade.pn.org/.
4. GtkAda home page. http://gtkada.eu.org/.
5. GtkAda User's Guide. http://gtkada.eu.org/gtkada_ug.html.
6. GtkAda Reference Manual. http://gtkada.eu.org/gtkada_rm_toc.html.
7. Gtk+ home page. http://www.gtk.org/.
8. Havoc Pennington. *GTK+/Gnome application development*. New Riders, 1999.
9. Eric Harlow. *Developing Linux Applications with GTK+ and GDK*. New Riders, 1999.
10. David Odin. *Programmation Linux avec Gtk+*. Eyrolles, 1999.
11. ACT-Europe home page. http://www.act-europe.fr/.