TkRibbon: Windows Ribbons for Tk

Georgios Petasis

**Software and Knowledge Engineering Laboratory,
Institute of Informatics and Telecommunications,
National Centre for Scientific Research "Demokritos",
Athens, Greece
petasis@iit.demokritos.gr**

Abstract

This paper is about TkRibbon, a Tcl/Tk extension that aims to introduce support for the Windows Ribbon Framework in the Tk toolkit. The Windows Ribbon is a graphical interface where a set of toolbars are placed on tabs in a notebook widget, aiming to substitute traditional menus and toolbars. This paper briefly describes Windows Ribbon framework, the TkRibbon Tk extension and presents some examples on how TkRibbon can be used by Tk applications.

1 Introduction

In this paper TkRibbon is presented, a C extension for the Microsoft Windows operating system that aims to introduce support for the Windows Ribbon framework in the Tk toolkit. The Windows Ribbon framework is a rich command presentation framework that aims to unify traditional multilayered menus, toolbars and task panes into a single, modern alternative graphical element. The Ribbon framework consists mainly of two primary user interface components:

- The Ribbon command bar. This component contains the application menu, a set of standard tabs, a set of contextual tabs (activated by the currently application element that has the focus), and a Help button. TkRibbon currently offers complete support for this interface component.
- A rich contextual menu. There is no support for the time being in TkRibbon regarding this interface component.

An example of a Windows Ribbon can be seen in Figure 1. The Ribbon framework, from the application point of view, can be decomposed into two distinct but dependent development platforms:

- A XAML-based markup language, which describes the controls, their properties and their visual layout.
- A set of COM C++ interfaces that ensure interoperability between the Ribbon framework and the application.

The Windows Ribbon framework is implemented as a COM object, which can be attached to any application window, and taking control of the window upper part (Figure 1) it redraws the window as needed, reacting to user keyboard/mouse input and requests from the application. The Ribbon framework is built around "commands": when the user activates a command (either by pressing a button, selecting an item, etc.), the framework uses COM messaging to communicate the action to the application. In addition, communication with the application can occur in some other cases, i.e. when the framework requests missing

information about a control (like a missing or invalidated property). TkRibbon acts as a mediator between the Ribbon framework and a Tcl/Tk application, translating messages into callbacks written in Tcl and Tk virtual events, while "packaging" a Ribbon instance as a Tk widget.

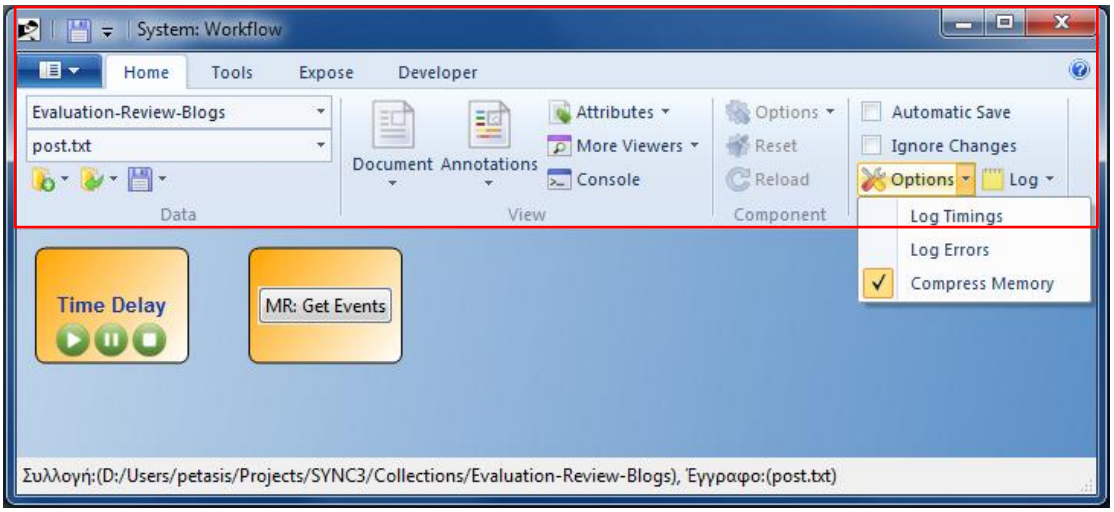

Figure 1: An example of a Windows Ribbon. The marked area is drawn by the Ribbon framework (including part of the window decoration).

The rest of the paper is organised as follows: chapter two presents an overview of how TkRibbon can be used in order to load and display a Ribbon that is contained in a resource DLL. The whole chapter is a running event starting from the XAML markup describing the Ribbon to the loaded Ribbon inside a Tcl/Tk application. Chapter three concentrates on the interaction between the Ribbon framework and the application, through a brief presentation of how a Ribbon requests information from the application, how Ribbon events are communicated to the application and how the application may pose requests to a Ribbon instance. Finally, chapter four concludes the document by reporting on the current status of the TkRibbon extension.

2 Creating a Ribbon

The process of implementing a Ribbon and incorporating it into a Tk application involves three basic tasks: describe the Ribbon controls in XML markup, compile the markup into a resource DLL, and write the code that loads the Ribbon from the resource DLL and handles the interaction between the Ribbon widget and the application. These tasks are illustrated in Figure 2.

2.1 Writing the Ribbon Markup

The first step in creating a Ribbon is, of course, to design it: to decide the tabs, the task panes each tab will have, and the controls that will be placed in each task pane. There is extensive documentation by Microsoft on how Ribbons should be designed in the "Ribbon User Experience Guidelines"[1], while documentation and tutorials can be found at the "Windows Ribbon Framework Developer Guides"[2].

[1] The "Ribbon User Experience Guidelines" can be found at:
http://msdn.microsoft.com/en-us/library/cc872782.aspx
[2] The "Windows Ribbon Framework Developer Guides" can be found at:
http://msdn.microsoft.com/en-us/library/dd742866%28v=VS.85%29.aspx

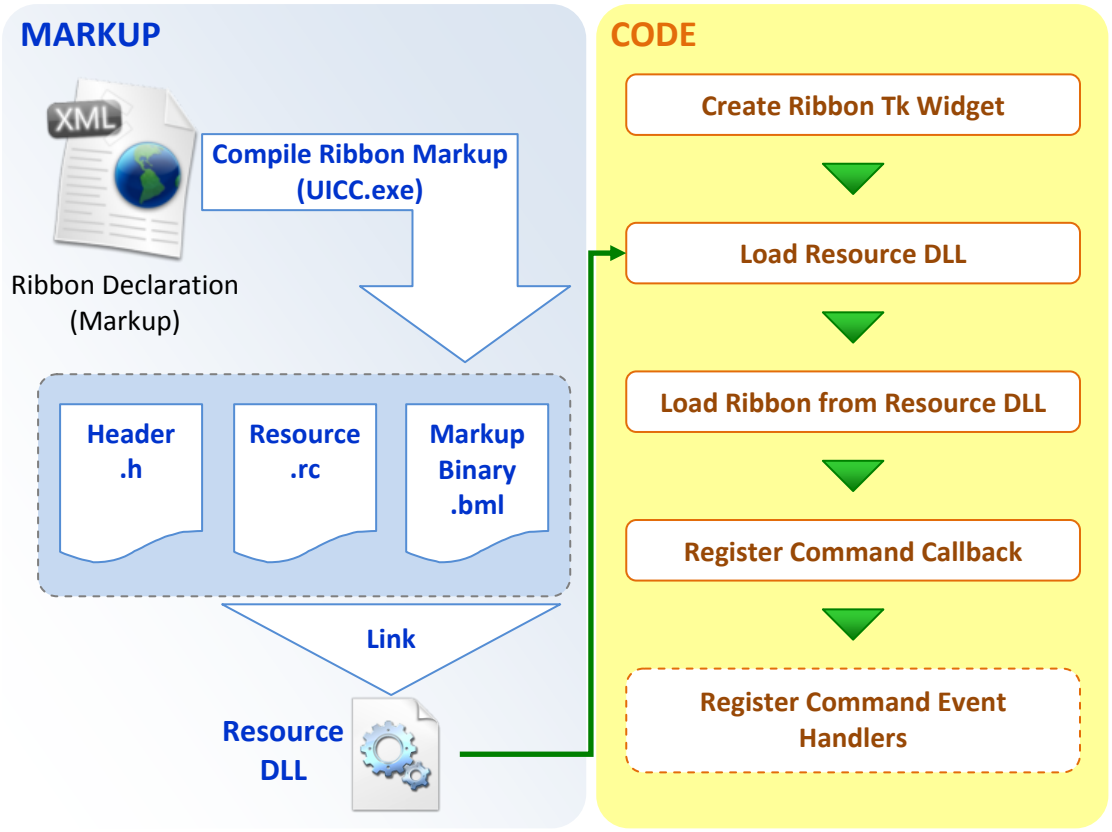

Figure 2: Creating a Ribbon in a Tk application.

Once the Ribbon has been designed, it must be described in XML using the Ribbon markup. The Ribbon XML contains two major parts. The first part defines the commands (with properties like a numeric id, a command name, the command label, tooltip, etc.), while the second part defines the organisation of the Ribbon in tabs, groups inside a tab, and commands inside a group (Figure 3). As an example, let's assume a Ribbon that has a single tab labelled "Home", with a single group labelled "Commands" that contains a single command, of type button, labelled "Exit". The following code illustrates the XML markup that describes this Ribbon:

```
<?xml version='1.0' encoding='utf-8'?>
<Application xmlns="http://schemas.microsoft.com/windows/2009/Ribbon">
  <Application.Commands>
    <Command Name="cmdExit" Symbol="cmdExit" LabelTitle="Exit"
             TooltipTitle="Exit" TooltipDescription="Exit Application..." />
  </Application.Commands>
  <Application.Views>
    <Ribbon>
      <Ribbon.Tabs>
        <Tab>
          <Group>
            <Button CommandName="cmdExit" />
          </Group>
        </Tab>
      </Ribbon.Tabs>
    </Ribbon>
  </Application.Views>
</Application>
```

Figure 3: XML markup of a simple Ribbon, containing a single command button.

For the purpose of the example, we will assume that the XML markup shown in Figure 3 is saved in a file named "ribbon1.xml"

2.2 Compiling the Markup

The XML markup must be compiled in order to be usable in any application, and there is a dedicated compiler for this in the Windows 7 SDK: the UI command compiler (UICC), which can be invoked from the command prompt:

```
uicc.exe ribbon1.xml ribbon1.bml /header:ribbon1.h /res:ribbon1.rc /name:RIBBON1
```

The parameters such as /name: are optional, but it is a good idea to identify each Ribbon with a custom name, instead of the default "APPLICATION_RIBBON", allowing multiple Ribbons to be placed inside a single resource DLL. Once the Ribbon markup has been compiled, a resource DLL can be easily created:

```
rc.exe ribbon1.rc
link.exe /NOENTRY /DLL /MACHINE:X86 /OUT:ribbon1.dll ribbon1.res
```

2.3 The TkRibbon widget

Up to now, there is nothing different in how a DLL containing one or more Ribbons is prepared. However, using any Ribbon from a DLL is somewhat different under TkRibbon, deviating from having to link the Ribbon to the code that uses it. Being an extension for the dynamic language Tcl, TkRibbon implements all the needed middle-ware to load a Ribbon UI from a DLL and interact with it. The main disadvantage of this approach is that interoperability is limited to what has been implemented by TkRibbon, suggesting that some advanced operations (i.e. serialising and restoring button layout changes done by user) are not yet supported.

In order to load and use a Ribbon from a Tk application, the steps shown in the code section of Figure 2 have to be followed. A minimal implementation in Tcl, which loads and uses a Ribbon from a DLL, is shown in Figure 4.

```
package require Tk
package require tkribbon
set ScriptDir [file dirname [file normalize [info script]]]
## The resources DLL containing the Ribbon...
set RibbonDLL $ScriptDir/ribbon1.dll
## Create a Ribbon widget:
set toolbar [tkribbon::ribbon .ribbon -command \
                        onRibbonUpdatePropertyDispatch]
## Load the resources DLL: must be executed at least once for each DLL...
$toolbar load_resources [file nativename $RibbonDLL]
## Load the Ribbon UI from the DLL...
$toolbar load_ui [file tail $RibbonDLL] RIBBON1_RIBBON
## Pack the widget at Toplevel top: ensure expanding is false!
pack $toolbar -side top -fill x -expand false
## Important: The Ribbon will not be drawn, unless the window is
## large enough!
wm geometry . 300x250 ;# The minimum size for showing the Ribbon!
## Events:
foreach event {Execute Preview CancelPreview CreateUICommand ViewChanged
               DestroyUICommand UpdateProperty} {
  bind $toolbar <<on$event>> [list onRibbonEventDispatch $event %d]
}
proc onRibbonUpdatePropertyDispatch {args} {
  puts "onRibbonUpdatePropertyDispatch: args: $args"
};# onRibbonUpdatePropertyDispatch
proc onRibbonEventDispatch {event args} {
  puts "onRibbonEventDispatch: event: $event, args: $args"
};# onRibbonEventDispatch
```

Figure 4: Loading and using a Ribbon from Tcl/Tk.

The result of running the code of Figure 4 is shown in Figure 5. It is interesting to note that elements missing a name in the XML markup specification of the Ribbon were assigned automatically generated names by the XML markup compiler UICC.exe. These elements (tab, group) can be assigned a name through commands: a command must be created for every element (tab, group) and assigned through the "CommandName" attribute to the corresponding XML tag.

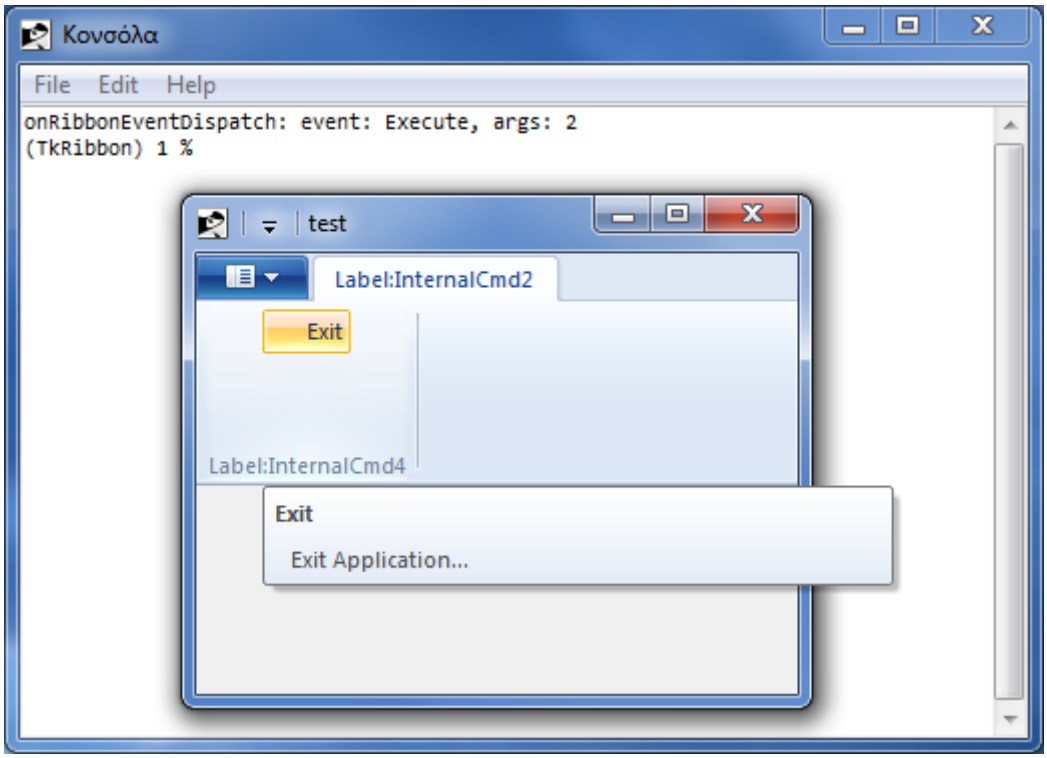

Figure 5: The loaded Ribbon

3 Interacting with a Ribbon

Interaction with an instance of a Ribbon is achieved through two means:

- A callback that can be registered to a widget through the "-command" widget option. This callback will be called each time the Ribbon wants to retrieve a control property from the application and is expected to return a suitable value, according to the control property requested.
- A virtual event of the form <<on*>>, delivered to the Ribbon widget. Events are not expected to return a value.

Controls inside a Ribbon UI are identified by an integer id, assigned by the XML markup compiler (UICC.exe). A mapping between the symbolic names and the assigned numeric ids can be found in the header file (.h) produced by UICC.exe, and is up to the application to map ids to symbolic names, if desired. The contents of "ribbon1.h" are shown in Figure 6.

```
// ************************************************************************
// * This is an automatically generated header file for UI Element definition *
// * resource symbols and values. Please do not modify manually.            *
// ************************************************************************
#pragma once

#define cmdExit 2
#define cmdExit_LabelTitle_RESID 60001
#define cmdExit_TooltipTitle_RESID 60002
```

```
#define cmdExit_TooltipDescription_RESID 60003
#define InternalCmd2_LabelTitle_RESID 60004
#define InternalCmd4_LabelTitle_RESID 60005
#define InternalCmd6_LabelTitle_RESID 60006
```

Figure 6: The contents of "ribbon.h", the header file produced by UICC.exe for the markup of our example.

3.1 The widget command callback

The widget callback is responsible for returning the value of various properties associated with controls. There are two main cases where the callback will be called: a) to retrieve properties not defined in the XML markup and b) to retrieve the value of properties that have been invalidated by the application (which is the primary means of changing any control property). The callback can be a Tcl command prefix, while two or 3 arguments will be appended before execution. A typical prototype of such a callback is:

```
proc callback {contol_id property_type {current_value {}} {...}
```

where:

- control_id is the numeric identifier of the control (integer),
- property_type is the type of the property whose value is requested (symbolic, one of UI_PKEY_Enabled, UI_PKEY_RepresentativeString, UI_PKEY_ItemsSource, UI_PKEY_Categories, UI_PKEY_SelectedItem, UI_PKEY_BooleanValue),
- and current_value is the current property value, if such a value exists, expressed in a type suitable for the current property type (i.e. a Boolean value for UI_PKEY_BooleanValue or a Tcl list for UI_PKEY_ItemsSource).

The callback may return an updated value according to the requested property or use "return –code break" to avoid setting a value. The property type UI_PKEY_Enabled is issued by the Ribbon UI in order to retrieve which Ribbon controls are enabled or disabled. Typically, it is called for all controls in a Ribbon, when the Ribbon is loaded, or a tab is raised for the first time. The return value is expected to be a Boolean Tcl value, which if true enables the control under question. The property type UI_PKEY_RepresentativeString is used in order to retrieve a "representative" string to be shown in a control that displays dynamic content (i.e. a combobox). The return value must be a string, whose dimensions will be used to obtain a size for the control under question. UI_PKEY_BooleanValue will be used when the Ribbon widget wants to retrieve the value of a checkbutton or radiobutton Ribbon control, with a Boolean value expected as the result.

The property types UI_PKEY_Categories, UI_PKEY_ItemsSource, UI_PKEY_SelectedItem are related to controls that display a collection of items, such as galleries and comboboxes. The items in collection objects can be separated into categories: the type UI_PKEY_Categories is used to retrieve these categories. The return value can be no categories (through "return –code break"), or a set of categories, specified as a Tcl list of one of the following formats:

```
[list [list categoryName1 ... categoryNameN] {} ]
[list [list categoryName1 ... categoryNameN] [list imageResourceId] ]
[list [list categoryName1 ... categoryNameN] [list imageResourceId1 ... imageResourceIdN]]
```

The three formats differ in the way an image is specified with a category. If no images are specified, no images will be used by the Ribbon widget when displaying the categories. If a single image is specified, this image will be displayed for all categories. Finally, an image can be specified for each category individually: in this case the length of the images list must match the length of the category names list. The image resource id can either be the empty string (no image), or an integer identifier of an image resource contained in the resource DLL of the Ribbon. It cannot be the name of a Tk image.

The property type UI_PKEY_ItemsSource works in a similar way, expecting a similar return value, as a Tcl list of 3 Tcl lists. This property type will be used by the Ribbon widget in order to retrieve the contents (items) of a control showing a collection (a gallery or a combobox). The return value of this property type is as follows:

```
[list [list item1 ... itemN] images-list  categories-list ]
```

where:

- images-list can be either the empty list, the resource identifier (integer) of a single image (used for all items), or a list of resource identifiers, specifying an image for each item. In the latter case, the number of items must match the number of image identifiers, and
- categories-list can be either the empty list (when no categories have been specified), a single category name (used for all items), or a list of category names, specifying a category for each item. In the latter case, the number of items must match the number of category names.

Finally, the property type UI_PKEY_SelectedItem is used to retrieve the currently selected item. The return value must be the numeric index in the list of items, with the first index position starting at 0.

3.2 The virtual events bindings

All interaction between the Ribbon widget and the application is performed through Tk virtual events. (The only exception is when the Ribbon widget wants to retrieve the value of the property as described in the previous section.) TkRibbon delivers seven virtual Tk events to the Ribbon widget:

- <<onExecute>>: this event will be delivered when the user has executed a control, i.e. a button has been pressed or a combobox/gallery selection has been changed. The identifier of the Ribbon control can be retrieved through the "%d" code in the binding script. This is the most important event that must be processed.
- <<onPreview>>: this event will be delivered when the Ribbon wants to temporarily apply the current value of the control, in order to show a preview before the user actually applies the value. The identifier of the Ribbon control can be retrieved through the "%d" code in the binding script.
- <<onCancelPreview>>: this event will be delivered when the Ribbon wants to cancel (terminate) a previously requested preview. The identifier of the Ribbon control can be retrieved through the "%d" code in the binding script.
- <<onCreateUICommand>>: reserved for future use.
- <<onViewChanged>>: reserved for future use.
- <<onDestroyUICommand>>: reserved for future use.
- <<onUpdateProperty>>: reserved for future use.

3.3 Widget subcommands

The widget callback and the events are the primary means of the Ribbon framework to retrieve information from the application and notify events occurring in the Ribbon to the application. Communicating in the opposite direction (i.e. the application requesting something from the Ribbon framework) is performed through subcommands of a TkRibbon widget instance. Currently, the following subcommands are available:

- *pathname* **attach**: this subcommand attaches a Ribbon to the Tk Ribbon widget. It is an internal subcommand that is called by the TkRibbon library when a Ribbon Tk widget is created.

- *pathname* **load_resources native-dll-path**: this subcommand loads a resources (or other DLL) into the application. It is a wrapper for LoadLibrary() and requires a single argument, the native full path name of the DLL file to be loaded.
- *pathname* **load_ui module ribbon-name**: this subcommand loads the Ribbon UI named **ribbon-name** from the DLL handle **module**. The DLL handle is the name of the DLL without the path.
- *pathname* **get_property property-type control-id**: this subcommand returns the value of the specified property. **control-id** is the numeric (integer) identifier of a Ribbon control, and **property-type** must have one of the following values: UI_PKEY_Enabled, UI_PKEY_RepresentativeString, UI_PKEY_ItemsSource, UI_PKEY_Categories, UI_PKEY_SelectedItem, UI_PKEY_BooleanValue.
- *pathname* **set_property**: reserved for future use.
- *pathname* **invalidate_state state-property control-id**: invalidates the state aspect of the specified **state-property**, for control **control-id**. **state-property** must have one of the following values: UI_PKEY_BooleanValue, UI_PKEY_ContextAvailable, UI_PKEY_DecimalPlaces, UI_PKEY_DecimalValue, UI_PKEY_Enabled, UI_PKEY_FormatString, UI_PKEY_Increment, UI_PKEY_MaxValue, UI_PKEY_MinValue, UI_PKEY_Pinned, UI_PKEY_RecentItems, UI_PKEY_RepresentativeString, UI_PKEY_StringValue.
- *pathname* **invalidate_value property control-id**: invalidates the value aspect of the specified **property**, for control **control-id**. **property** must have one of the following values: UI_PKEY_BooleanValue, UI_PKEY_ContextAvailable, UI_PKEY_DecimalPlaces, UI_PKEY_DecimalValue, UI_PKEY_Enabled, UI_PKEY_FormatString, UI_PKEY_Increment, UI_PKEY_MaxValue, UI_PKEY_MinValue, UI_PKEY_Pinned, UI_PKEY_RecentItems, UI_PKEY_RepresentativeString, UI_PKEY_StringValue, UI_PKEY_Keytip, UI_PKEY_Label, UI_PKEY_LabelDescription, UI_PKEY_LargeHighContrastImage, UI_PKEY_LargeImage, UI_PKEY_SmallHighContrastImage, UI_PKEY_SmallImage, UI_PKEY_TooltipDescription, UI_PKEY_TooltipTitle, UI_PKEY_Minimized, UI_PKEY_QuickAccessToolbarDock, UI_PKEY_Viewable, UI_PKEY_GlobalBackgroundColor, UI_PKEY_GlobalHighlightColor, UI_PKEY_GlobalTextColor.
- *pathname* **invalidate_property property control-id**: invalidates the property aspect of the specified **property-type**, for control **control-id**. For valid **property** values, please see subcommand **invalidate_value**.
- *pathname* **invalidate_all property control-id**: invalidates all aspects of the specified **property**, for control **control-id**. For valid **property** values, please see subcommand **invalidate_value**.

4 Conclusions

This paper presented TkRibbon, a Tcl/Tk extension written in C++ which allows the use of the Windows Ribbon Framework from Tcl/Tk. Currently, the TkRibbon extension is in alpha state, while the first public release is pending. Despite its alpha state, many features of the Ribbon framework are already supported, while more features are expected to be supported until the first stable release. TkRibbon is hosted at SourceForge, and can be found at http://sourceforge.net/projects/tkribbon/.

A large part of this document has been devoted on how to use the TkRibbon extension in order to use a Ribbon contained inside a DLL, followed by a brief description of the available callbacks, events and widget subcommands. However, a basic understanding of how the Windows Ribbon Framework operates is required in order to understand some the available functionality, especially regarding the various property values.