Towards platform independence: retargeting GUI libraries on .NET

Judith Bishop and Basil Worrall
Department of Computer Science
University of Pretoria
Pretoria, South Africa
Jbishop@cs.up.ac.za, basil.worrall@dariel.co.za

ABSTRACT

Platform independence is an illusive goal when a system includes libraries which have hardware or low-level software dependencies. To move such code to a different platform, the developer is faced with rewriting several sections to interface directly with a different library or toolkit. We propose an approach where the code remains the same, and the library is replaced *ab initio* by a machine-independent engine which is retooled into a front end and a back end, of which only part of the backend needs to change for each platform. Our starting point is the .NET framework's SSCLI platform, Rotor, and the Views GUI engine, which initally ran only on Windows. Views is an XML-based windowing system which provides the functionality of the System.Windows.Forms library, missing from Rotor. ViewsQt is a conversion of the original Views project to support a retargetable back-end. Experiments have shown that the ViewsQt code is portable, with only a few changes to the C++ classes required to compile and execute the code on the Linux and Mac OS X operating systems. On the Windows platform, ViewsQt works well with both the .NET Framework and Rotor. This paper describes the methodology we developed for porting libraries in general, discusses the case study of ViewsQt, and indicates where such work would be applicable for other technologies. Comparison is made with multi-platform toolkits such as Gtk+, and .NET's new XAML notation.

Keywords
Platform independence, GUI toolkit, .NET, Qt, Rotor, retargeting methodology, Linux port, Views, XAML

1. INTRODUCTION

The innovative move of Microsoft to undergo a standards process for their .NET framework and C# language raised hopes of platform interoperability being added to the language interoperability already supported by .NET [9]. Apart from portability, Microsoft's implementation of the CLI (Rotor) was intended as a basis for experiment and Microsoft itself used it in order to test out its ideas on generics, which are available in the Gyro add-on, and are now planned for the next release of Windows, codenamed Longhorn [10].

The CLI (Common Language Infrastructure) included the definition of the C# language and many of its key libraries, such as System and System.Collections. However, not all .NET libraries are included in the standard, with a notable omission being System.Windows.Forms, which provides GUI capability. This means that developers cannot *express* GUI functionality in their programs (since it will not compile) and there is no way, in the standard, to hook into the operating system to *render* and *handle* GUIs even if they could. GUIs are a primary need of many programs, but the issue of portability extends to third-party libraries as well: how would they piggy-back on Rotor?

Standing back, one can see that the problem is one of having invested in developing a program based on a particular library, and then finding that the program cannot migrate to a new platform, because of the library's reliance on hardware or low-level software. If the library is a large and critical one, such as a GUI, then any alternative to a complete re-implementation would be desirable.

Although this paper will concentrate on GUI libraries, other emerging hardware-oriented technologies have the same problem of portability.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-86943-01-1
Copyright UNION Agency – Science Press, Plzen, Czech Republic

Among these are tangible user interfaces (TUIs) and mobile applications. TUIs integrate digital information with everyday physical objects such as electronic tags and barcodes. Papier-Mâché [11] is an open-source toolkit for building TUIs with a high-level event model to facilitate portability. CrossFire [12] is a third-party product built on top of .NET. Crossfire uses a booster to the CLR to enable code in VB to run on the compact frameworks used by a variety of mobile devices, such as cell phones and palmtops. In this way, Crossfire also enhances portability.

Multi-platform GUI toolkits have long been popular for enhancing the capabilities of languages and packages lacking built-in GUI facilities. Recent examples are RAPID for Ada [6], FranTk for Haskell [9] and SMLTk for ML [10]. Because these languages have no UI capability of their own, they adopt the interface of the toolkit, and the programmer inserts code to interact with the toolkit directly.

In the .NET world, there have been similar projects to port GUI toolkits onto the CLI. Gtk# is a translation by the Mono project of the Gtk+ toolkit into C# [1][1]. The programmer familiar with Gtk will feel comfortable calling the well-known methods, but a .NET programmer with a Windows program to port could be at a loss. For example, creating a label, textbox and button in Gtk# is done with:

```
Label label = new Label("Password");
Entry entry = new Entry();
Button button = new Button("Submit");
```

which is quite different to the Windows equivalent of:

```
Label label = new Label();
label.Text = “Password”;
Textbox entry = new Textbox();
Button button = new Button();
button.Text = “Submit”;
```

In other words, Gtk# is not a means for porting *existing* Windows programs via the CLI to the Linux platform. Qt# is a similar project intended to provide a binding of Qt to C#, and is still under development. And of course there is PIGUI which is based on Tcl’s TK and is distributed with Rotor.

This paper addresses the issue of retargeting a library across languages and platforms, without rewriting it or creating a new wrapper for its programming interface. Our contribution is in providing a methodology that can be followed for other libraries, as well as in identifying potential stumbling blocks on the .NET framework, and proposing solutions.

[1] Throughout this paper, projects and products whose primary source of information is a website are listed at the end of the paper, but not referenced in the text.

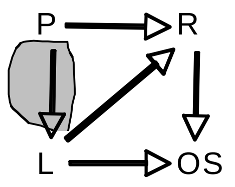
The methodology is explained via a case study of the life cycle of our platform-independent GUI engine Views. We show how we were able to take a library dependent on Windows and. via a combination of Rotor, Views, our retargeting methodology and the Qt toolkit, to achieve the same GUI functionality on other platforms, including Mac OS X and Linux.

The rest of the paper is structured as follows. In section 2 we introduce the retargeting methodology. Section 3 briefly describes Views, which is the basis for the case study. Sections 4 and 5 look at the retargeting process in detail. In Section 6 we evaluate the outcome, and in Section 7 discuss related work. Views is an ongoing project, so the conlusions in Section 8 include mention of late-breaking projects and future work.

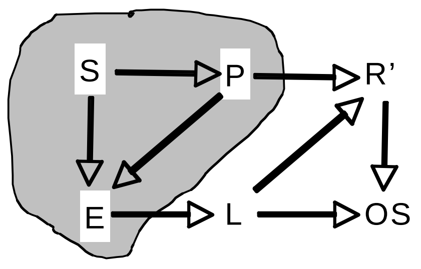
2. RETARGETING METHODOLOGY

2.1 Overall plan

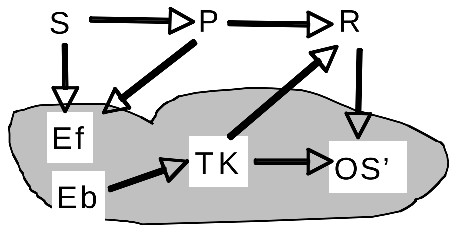
The retargeting methodology we developed is explained in the stages shown in Figure 1.



(a) Normal operation of a library



(b) Introduction of GUI specification and engine



(b) Library replaced by OS-independent toolkit

Figure 1 Stages of the retargeting methodology

We start off with a program P using a library L running on a given runtime R (virtual rachine) and operating system OS which supports L's low-level activity. An example would be a program in C# (P) using System.Windows.Forms (L) on the CLR (R) on Windows (OS).

In the first step towards gaining independence of the operating system, we introduce a GUI specification S (in XML notation) to specify the function of the library, in other words the programmer's interface. Instead of the label, textbox and button code:

```
Label label = new Label();
label.Text = "Password";
Textbox entry = new Textbox();
Button button = new Button();
button.Text = "Submit";
```

we would write the XML specification:

```
<Label text='Password'/>
<Textbox name=entry/>
<Button name=button text="Submit"/>
```

We then replaced the GUI creation and handling functions of System.Windows.Forms by this XML interface plus the Views engine E. Although this phase could run on an alternative runtime R', such as Rotor, it still needs the rendering ability of the Windows dll. Thus stage (b) can still only run on the Windows OS. This work is discussed in [4]. A welcome side effect of the XML notation is that the existence of the library becomes program- (and therefore language-) independent.

In the third stage, which is the subject of this paper, we take the engine and split it into a front and back end, Ef and Eb. The interface between the two parts is chosen so that it can operate with an existing cross-platform toolkit TK. The system can now run on any platform OS' on which the toolkit runs. In our case, we inserted Trolltech's Qt (TK), which runs on the same operating systems that Rotor does, but *also* on Linux (OS'). Thus the retargeting is complete.

2.2 General retargeting steps

The methodlogy can be applied in other spheres. The three steps to be followed in the process of achieving stage (c) platform independence are:

1. *Understand the design and implementation of the original system.* In our context, the original system is the version of Views that relies on the Windows dll. In this step our objective is to model the contractual agreement between the existing components of the Views system, and in so doing provide a point of reference for implementing this interaction in the retargeted version. For example, when the system is given the instruction to render a button, positioned relative to a textbox, we not only have to ensure that a button and a textbox are rendered, but also that their relative positioning remains intact.
2. *Extract the common components from the original system, and put them into an interface.* The model of contractual interaction developed in the first step needs some (similarly) abstract representation in the code. An interface is ideal for this purpose, as it allows any appropriate implementation to take its place in the run-time environment, yet provides enough structure and usage information to limit the breaking of the contract between the user of the interface and its implementer. Typical common components in GUI systems would be the XML parser and the window and control manipulation mechanisms.
3. *Write a toolkit-specific implementation of the C# interface which pulls in the services of the extracted common components.* Here we take the toolkit and translate (or aggregate) its functionality to the expectations of the model and its interface. It is here that we make sure that when the user wants a button, they get a button, so to speak.

We now make this methodology concrete by considering our case study, the retargeting of System.Windows. Forms to Linux.

3. THE CASE STUDY - VIEWS

3.1 The objective

The intent of the Views project is to provide a GUI system for the Rotor platform that would share Rotor's platform independence, and enhance it by offering programmers the much-needed support to provide GUIs with their Rotor applications [3]. We were not in the business of duplicating large effort, so the intention was always that Views would rely on an existing underlying GUI renderer to actually display the GUI. When running on Windows, or on Rotor on a Windows platform, Views makes use of the System.Windows.Forms dll to perform this function.

From the outset of the Views project, it was envisaged that this reliance on one platform would be removed by refactoring the Views code so that an independent toolkit (e.g. Tcl/TK or Qt) could be plugged into the system, allowing it to run on the platforms these toolkits support (which, in most cases, are also the platforms that Rotor supports). In terms of the user's code, the interface would remain the same (including the XML notation).

3.2 Overview of Views

Views allows the user to specify a GUI in a simple and easy-to-learn XML notation, and then to integrate the application with this GUI through an elementary interface to the core engine. No code generation takes place, and the GUI specification can be stored in an external file so that it will not obfuscate the application's logic. A side effect of keeping the GUI specification and application logic separate is that the programmer can make simple changes to the controls in the specification (e.g. their layout, or even substituting a drop-down list for a collection of radio buttons) without having to recompile the program. From the opposite perspective, the GUI can be reused by a number of applications that require a similar front-end while presenting different results (e.g. a calculator program that prints out expressions in either standard algebraic or reverse Polish notation). More information about the use and implementation of the Views project can be found in [2,3,4].

The Views interface consists of two parts, namely

- the Views notation for specifying a GUI in XML, and notation, and
- the Views engine which provides an interface to the programmer.

We now take a brief look at each of these, to give an idea of the scope of work involved in transforming the interfaces to abstractions of an arbitrary windowing toolkit.

3.3 The Views notation

A typical GUI specification in Views consists of two types of tags – grouping and control. A third type, position tags, can also be used for finer layout control. Grouping tags may contain nested groupings and controls, and dictate a specific layout of these sub-groups or controls.

```
static string specEn =
  @"<form Text='Currency calculator'>
  <horizontal>
    <vertical>
      <Label text='Paid on hols'/>
      <Label text='Charged'/>
      <Label text='Exchange rate is'/>
      <Button name=equals text='='/>
    </vertical>
    <vertical>
      <Textbox name=eurobox/>
      <Textbox name=GBPbox/>
      <Textbox name=ratebox/>
      <Button name=clear text='Reset'/>
    </vertical>
  </horizontal>
</form>";
```

Figure 2 A Views specification

For example, the `<horizontal>` group specifies that all groups and controls contained within it be placed side by side from left to right. Each tag has some valid attributes, among which are numeric values, strings, colors, alignment values and size measures. Figure 2 shows a typical Views specification.

To create a GUI, the programmer passes the specification to an instantiation of the Views Form class, as in:

```
Views.Form f = new Views.Form(specEn);
```

Figure 3 shows the corresponding GUI as drawn by the Windows renderer.

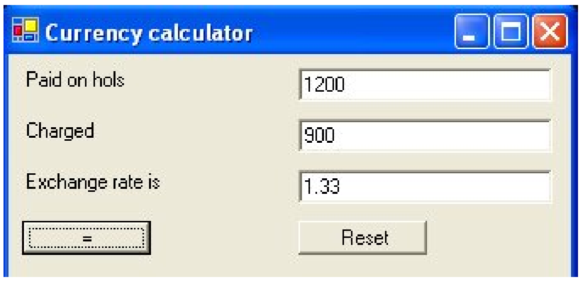


Figure 3 A GUI produced by Views

3.4 The Views programmer interface

Views presents a small, yet complete number of functions the user can use to query and alter the controls defined in the specification, and to react to simple “clicked” or “moved” events.

There are three variations of Get methods, namely GetControl, GetText and GetValue. The GetControl method is the means through which the application is informed of events occurring in the GUI. GetControl blocks until an event occurs, upon which it returns the name of the control where the event occurred. The GetText method accepts the name of a control that can display text (e.g. labels, buttons, textboxes), and returns the text that control is currently displaying as a string. GetValue is similar, and is used for trackbars, checkboxes etc. Two of the three types of Put methods, PutText and PutValue, are the logical counterparts of the Get methods. Views also provides a PutImage method. Part of the program associated with the specification above is shown in Figure 4.

A feature of Views is that is not “black box”: any of the controls can be accessed by name, and their attributes changed. For example, to change the text of the equals button in the form f from “=” to “equals”, and colour it yellow, we use:

```
Button b = f["equals"];
b.Text = "Compute";
b.BackColor = Color.Yellow;
```

Using the C# implicit operator facility for overloading parenthesees, implicit conversions are defined for all controls that may be used inside a

Views form, so that casting to the data type of the extracted control is unnecessary.

```
for (string c = f .GetControl();
     c!=null; c = f .GetControl()) {
  switch (c) {
  case "reset":
    euro=1; GBP=1;
    f.PutText("eurobox",
              euro.ToString("f"));
    f.PutText("GBPbox",
              GBP.ToString("f"));
    break;
  case "equals":
    euro=double.Parse(
         f.GetText("eurobox"));
    GBP=double.Parse(
         f.GetText("GBPbox"));
    f.PutText("ratebox",
         (euro/GBP).ToString("f"));
    break;
   default: break;
  }
}
```

Figure 4 Event handling in Views

3.5 Why Views?

If the goal is to retarget existing programs based on Windows, why is a new library such as Views a good idea? Firstly, the XML front-end achieves language portability, and its notation is quicker and easier to write and modify than the equivalent method calls and property accesses of a traditional GUI library. An alternative to coding GUIs by hand is to use a GUI builder to lay out the window, and have it generate the embedded program code, as Visual Studio does. However, large amounts of generated and embedded code are considered to be both confusing and error-prone.

An alternative is to have the GUI builder generate the XML, and we have such a system for Views in prototype. XAML takes this approach too, as does RAPID [5]. A comparison of Views with other XML based systems is undertaken in section 7.

Although Views was primarily aimed at beginning programmers [3], its methods and appeal extend wider, as does its use as a case study for retargeting.

4. FRONT-END FACTORIZATION

In the original, Windows-specific, implementation of Views, the process of converting a GUI specification to a visible window proceeded along the lines shown in Figure 5. The original design of Views incorporated many modular elements, the majority of which are toolkit independent. These modules represent important aspects of the system's behaviour, and should therefore be carried across to a portable version.

However, there are elements of the programmer interface to the engine that are very tightly coupled to the Windows Forms library, and cannot be migrated without change. For example, steps 1-3 in the diagram that involve processing the XML and building a tree, are platform-independent. However, laying out and displaying the GUI will depend on the renderer and, while GetControl is free of any reference to the Windows Forms Library classes, it is indirectly dependent on synchonizing with their event-triggering.

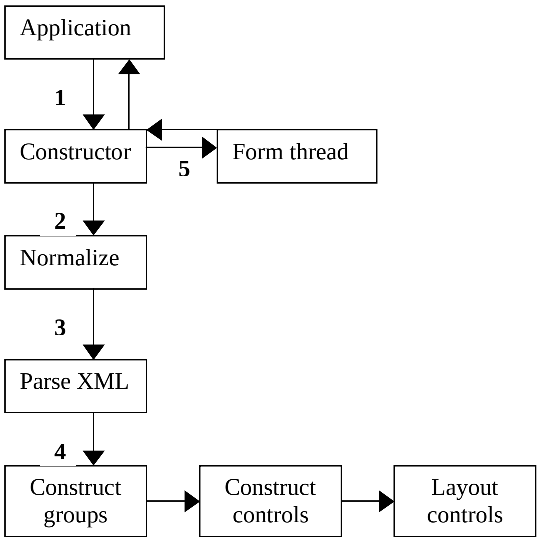


Figure 5 Control flow in Views

When considering cross-platform realization of Views, we can see that there are components that straddle the imaginary line between the front-end and the back-end. For example, the methods defined in the programmer interface are accessible to the application, yet are dependent on the toolkit. In order to successfully implement a toolkit-independent version of Views, we need to divide these grey-area components in such a way that the overall separation between the front- and back-end is solid. This will allow the back-end to be interchangeable, effectively enabling us to run Views on top of any toolkit.

The way we chose to implement this separation was to create a C# interface, called IForm, which declares all the Views API methods accessible to the application, as in Figure 6.

In the Windows.Forms implementation of Views, the XML-tree traversal builds the window by instantiating the controls, placing them and hooking up the event handlers.

```
namespace Views {
  public interface IForm {
    void HideForm();
    void StartApplication();
    String GetControl();
    String GetText(String name);
    String GetText(String name,
          int index);
    void PutText(String name,
          String text);
    void PutText(String name,
          int index, String text);
    void PutImage(String name,
          String filename);
    int GetValue(String name);
    void PutValue(String name,
          int value);
  }
}
```

Figure 6 The IForm interface

In the toolkit independent version, we do not rely on the back-end to parse or traverse the XML, so there is a requirement to construct a tree comprising toolkit-agnostic nodes which the back-end can traverse and interpret. The nodes are instances of a new class, Ctrl, which encapsulates information regarding the name, value, attributes and children of a tag in the XML specification. The tree of Ctrl nodes is built by another new class, Parser, which reproduces all the XML-processing code from the original Views.Form class.

Iform replaces Form as the class used to construct a GUI window, as in:

```
Views.Iform f =
       new QtForm.QtForm(specEn);
```

An implementation of the IForm interface can use the Ctrl tree to construct control instances specific to the toolkit, without having to be aware of the original XML tree. Thus we have successfully separated the front-end and back-end of Views. The XML has been cleared of all references to toolkit classes, and the programmer interface has been placed behind a clean interface that deals only in names and integer values. A reusable abstraction of the controls and their attributes was created to purge the back-end code of any references to the XML structure.

5. BACK-END IMPLEMENTATION

For our test implementation of the retargetable Views framework, we chose Trolltech's Qt toolkit. Qt is a complete application development library for C++, including APIs for GUI rendering, XML parsing, database connectivity and much more. Full details of our implementation are given in [17]. Some of the issues that relate specifically to .NET with Qt are mentioned here.

5.1 Language interoperability

Since Qt is written for, and in, C++, an interoperability layer (written in C#) that implements the interface is required. Thus we have a C# class, QtForm, that implements IForm, but delegates most of its functionality to a wrapper class, QtWrapper. The latter consists of a set of simple wrapper methods that correspond with the methods defined in IForm, and a set of private, static methods that link with externally defined C++ methods.

Two additional issues were solved at this point. First, because C# and C++ have different mechanisms for dealing with strings, it was necessary to write marshalling methods that convert between the two.

The second aspect is the entry-point specification in the DllImport attribute attached to the GetText method. The C++ linker provides a specially encoded string for every method declared to be externally visible in the source code, called its entry-point. This string can be used by other languages to discover the method within the dll that is produced from the C++ source code. Unfortunately the entry-point is compiler-specific, and also differs from OS to OS. Thus, until a truly platform independent entry-point specification mechanism is found, the QtWrapper class will require adjustment for every platform/compiler combination to which ViewsQt is ported.

Returning briefly to the implementation of the IForm interface, QtForm, we can now easily invoke the methods of the C# QtWrapper class, blissfully unaware of the underlying C++ implementation:

```
public String GetText(
       String name, int index) {
  return this.wrapper.GetText
       (name, index);
}
```

5.2 Garbage collection

When writing an interoperable program it is vital to ensure that references to elements in one language made in the other are kept valid for the lifetime of that reference. When one of the languages is managed (i.e. has built-in garbage collection), this task adopts an extra degree of complexity – the rearrangement of the heap will invalidate any references that weren't present on the stack during the collector's walk, which includes those held by the other program. In this case, the referenced object is still on the heap, indicating that a reference still exists within the managed program. More serious is the situation where the unmanaged program holds the only references to an object on the managed heap.

The garbage collector will happily free the heap space, once again invalidating the unmanaged reference.

There are two areas of ViewsQt where careful memory management is necessary to prevent errors. The first is the passing of strings between C# and C++, which happens in the QtWrapper and QtCtrl twins. The second is the pointer to the C++ QtCtrl instance held by the C# QtCtrl instance. In the context of the string-passing, a string passed from C# to C++ must not be garbage collected before the C++ code has had enough time to copy the contents to its own heap. The QtCtrl issue is slightly trickier. In this case, we wish to prevent garbage collection on the C# side so that we can tidy up the C++ heap at the end of the program.

In both cases, we stop the C# garbage collector from collecting the objects by obtaining instances of the System.Runtime.InteropServices.GCHandle class for each object. In doing so, the garbage collector treats the objects as if they had been pinned down in the heap – they cannot be moved or removed. We maintain a list of these GCHandle instances so that we can free them at an appropriate point in the execution. We don't mind the GCHandle instances themselves being moved around, as long as the objects they point to stay put.

5.3 Handling Events

There are two kinds of event handling which need to occur in an implementation of Views. The first is an internal mechanism that responds to the push-based events received from the GUI controls. A user of Views is shielded from this implementation by the second kind of handler, a pull-based (or polling) mechanism implemented in the GetControl method.

These two event handler types are complementary – when the GUI triggers an event, the internal handler looks up the name of the source control and forwards it to the GetControl. The application can then handle the event suitably. Figure 7 illustrates the two kinds of event handling interacting with each other.

In (1) the user's program calls GetControl, which blocks indefinitely. In (2) the operating system's windowing system interprets a user's gesture with the mouse or keyboard as an event, and passes it onto the event queue. The toolkit, having registered with the queue to hear about such events, picks up the information, encapsulates it in an Event object and passes it onto views in (3). Views extracts the name of the user-interface control (in this case button X) from the event information and passes it, in (4), to the user's program as the return value of the GetControl method.

In ViewsQt, we instrument push-based event handling by providing "slot" methods that are invoked when a control's "signal" is emitted. This is not unlike C#'s event implementation, where a multi-cast delegate (slot) is associated with a specific event (signal) published by an object. (In both C# and Qt, any object may fire events.) While it is possible to create a separate method for each kind of signal that each kind of control emits, we felt it a better abstraction to filter the events in such a way that a single eventHappened signal is emitted that contains a reference to the name of the control that originally emitted the event.

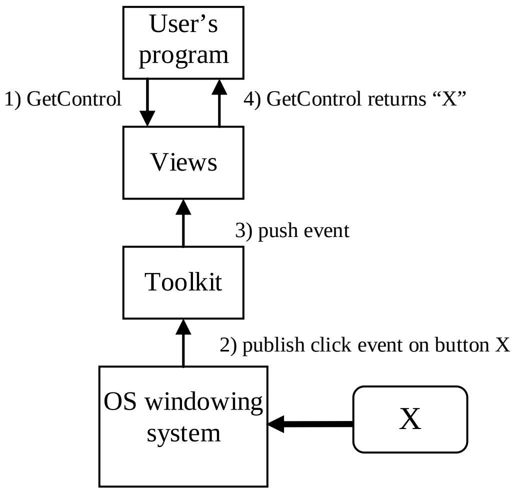


Figure 7 Event handling

This brings us to the implementation of the pull-based event handler. When a button is clicked, for example, the clicked method defined in QtWrapper is invoked. This method simply invokes a function pointer, listener, that is defined in the QtWrapper class. This function pointer references a method signature assigned to it in the SetListener method. The constructor for QtForm invokes the SetListener method defined in the C# QtWrapper class, passing it a variable called callback. This variable is in fact a C# delegate that refers to the ClickHappened defined in the QtForm class. The delegate is of type Delegate, which is declared in the C# QtWrapper class. The declaration of Delegate and the instantiation of callback are shown below:

```
public delegate void Callback(
    [In] IntPtr name);
QtWrapper.Callback callback = new
    QtWrapper.Callback(ClickHappened);
```

The C# QtWrapper class imports the setListener method from its C++ equivalent as follows:

```
static extern void setListener(
    [In] IntPtr ptr,
    [In, MarshalAs(
     UnmanagedType.FunctionPtr)]
     Callback l);
```

The MarshalAs annotation specifies that the reference to the Callback passed to setListener should be converted to a native function pointer. This amazingly simple mechanism allows native C++ code to easily invoke methods defined in C#. A proviso is that the method signature in C++ must specify its method-pointer argument using an equivalent descriptor.

5.4 Matching the libraries

In retargeting a library via a third party toolkit, it is inevitable that not all features offered in the original will be matched in the other. We were fortunate that there was only one such disparity between Forms and Qt, the DomainUpDown, which displays a single string from a list of strings, with up/down buttons to select other strings in the list. The closest equivalent in Qt is the QSpinBox, which by default displays a single integer in a range, with up/down buttons to select the next/previous value. We found it was possible to achieve a mapping by extending the class and overriding some methods. The code the user writes remains unchanged despite this underlying change, which meets the requirement that retargeting Views should not change the front-end syntax or semantics.

5.5 The Linux port

Since Linux has such a huge following, expecially in academia, it was a primary objective to get Views onto this platform. Once Views had been retargeted to Qt, thus eliminating the dependence on Windows.Forms, it could be run on Rotor (and all its platforms) as well as Mono (and its platforms). A group of students undertook the port to Linux, which involved writing the make files and resolving issues of paths and error messages. It was interesting that the port to Debian Linux did not work immediately on other Linux versions, such as Gentoo and Mandrake, and work is progressing on those.

6. EVALUATION

6.1 Example

Figure 8 (a) and (b) show a GUI with a variety of controls as rendered by ViewsQt and Views, both running on Windows. The program is taken from Chapter 5 of [3]. The back-end abstraction can be seen to work, at least in the Qt case. That is, constructing an IForm instance that mediates between the Views front-end and objects specific to the back-end GUI toolkit is not difficult, and most of the retargeting effort lies in implementing the objects.

Furthermore, these objects are not especially complex, but it is important to instrument all the functionality expected by the front-end, and to accommodate issues of interoperability between languages.

As mentioned above, we tried as far as possible to keep the code that a user of Views would write the same across both implementations. This was not possible in the case of the main application thread, but in such cases a balance must be struck between that which we would rather not to do and that which we cannot do. Adding a single line of thread-related code to the application forms this balance.

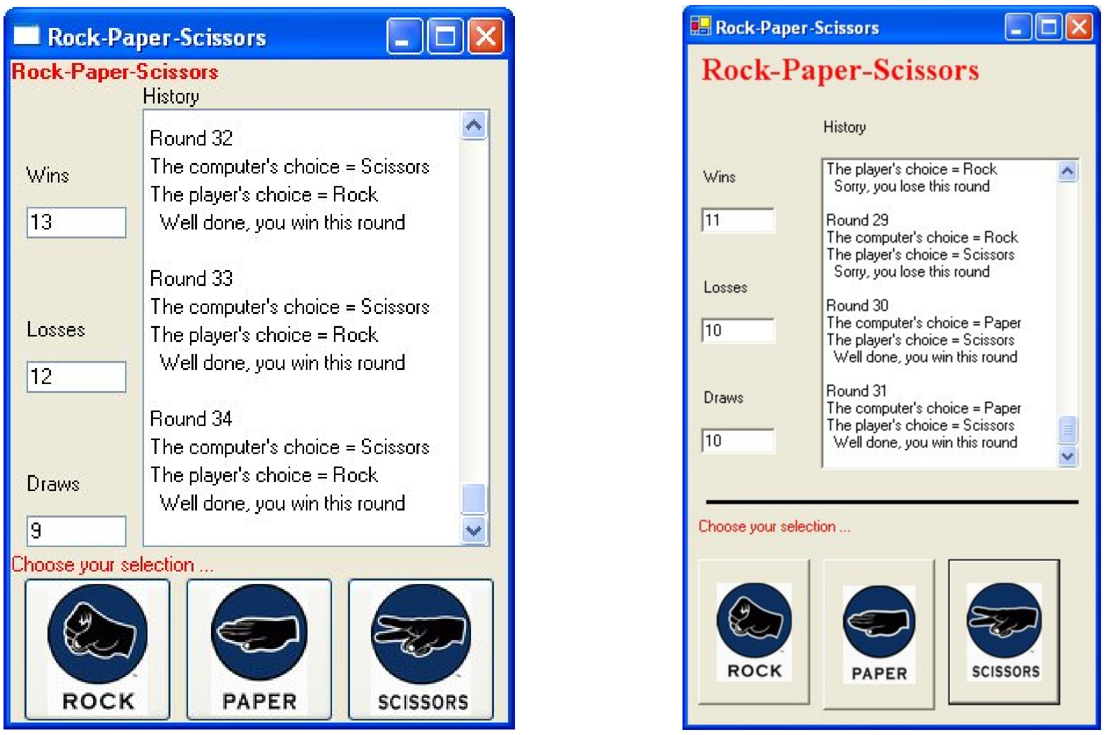
6.2 Other platforms and languages

Using Rotor as the base CLI, ViewsQt was successfully run on BSD UNIX and MacOS X. It is also worth reiterating that because of the language interoperability of .NET, ViewsQt, although written C# and C++, is available to programmers writing applications in other .NET languages. Specifically, it has been tested with programs written in C++ and Visual Basic. So far, the programs run correctly, and no changes to Views have been required.

6.3 Choice of toolkit

A key component of our methodology is the straight use of an existing multi-platform toolkit, rather than any writing or re-tooling. Three commercially available toolkits are Tcl/Tk, Gtk+ and Qt. In the planning phase of the retargeting project, Tcl/TK was considered as a viable option for the implementation. However, we chose to use Qt as Tcl/TK involved not only a significant performance trade-off (Tcl is always interpreted), but also a steeper learning curve in order to become conversant with Tcl's syntax and semantics. Qt, being entirely based on C++ and presenting a very natural programming interface, was the better choice for our purposes. However, one disadvantage to using Qt is that a development license must be purchased for the Windows version (Qt/Windows) in situations not covered by an academic licence or where the 30-day trial period is insufficient.

An important factor in choosing a toolkit is that it must be as multi-platform as possible. In this respect, Gtk+ would also have been a possibility. However, the toolkit is completely hidden from the developer, therefore there is nothing to be gained in repeating the exercise with a second toolkit.


Rock-Paper-Scissors

Rock-Paper-Scissors

History

Wins 13

Losses 12

Draws 9

Round 32
The computer's choice = Scissors
The player's choice = Rock
Well done, you win this round

Round 33
The computer's choice = Scissors
The player's choice = Rock
Well done, you win this round

Round 34
The computer's choice = Scissors
The player's choice = Rock
Well done, you win this round

Choose your selection ...

Rock-Paper-Scissors

Rock-Paper-Scissors

History

Wins 11

Losses 10

Draws 10

The player's choice = Rock
Sorry, you lose this round

Round 29
The computer's choice = Rock
The player's choice = Scissors
Sorry, you lose this round

Round 30
The computer's choice = Paper
The player's choice = Scissors
Well done, you win this round

Round 31
The computer's choice = Paper
The player's choice = Scissors
Well done, you win this round

Choose your selection ...

Figure 8 A program in ViewsQT and Views

7. RELATED WORK

In looking at related work, we concentrate on how our methodology relates to other similar attempts to provide cross-platform libraries. Predictably, the major effort in this regard has centred on GUI interfaces and toolkits, thus this section focuses on efforts in this area..

7.1 Declarative UI models

A key component of the retargeting strategy is the introduction of XML for the specification of the GUI. Two examples of the genre of declarative user interface models are IUP/LED [12] and CIRL/PIWI [7]. In both cases, a declarative language (LED and CIRL) was provided to describe the user interface in terms of its controls and layout. On the API front, they contain functions for hooking events signaled by the interface to call-back methods defined in the user's application, and functions to query and alter attributes of the controls displayed. The call-back event model is used so that the usual native windowing toolkit's events are filtered down to those relevant to the application.

Both CIRL/PIWI and IUP/LED were designed from the start to abstract the GUI description from the underlying platform's toolkit, and to provide a similar look-and-feel across the various platforms. The creators of both projects, however, lament the absence of an existing toolkit that provided a common look-and-feel across various platforms (both projects were born in the pre-Java and before any widely-accepted platform-independent toolkits, such as Qt and Tcl/TK, were available). Our work on the ViewsQt project was not hindered by these concerns because of the high-quality, platform independent toolkits available to us today.

7.2 XAML and XUL

Views belongs to the *concrete representation model* subdivision of the declarative user interface models, which describes user interfaces in terms of the controls displayed to the user, their composition and their layout. Such declarative user-interface models are not new [8,14], and XML is broadly being adopted as the favourite notation for these languages. Two modern, XML-based models are XUL and XAML.

XUL is the model used by the Mozilla family of browsers. A feature of XUL is the ability to create additional custom widgets using a related language called the Extensible Bindings Language (XBL). XUL is certainly cross platform, but its primary disadvantage is that it is tied to JavaScript for the event handlers.

XAML is the model Microsoft is making available with Version 2 of the .NET Framework, and is also the foundation for the Avalon windowing system component of the Longhorn version of Windows.

XAML is very similar to Views in that rides on the language interoperability of .NET. Unlike Views, there are no push-based event methods, and all handlers are also indicated as method names in the XML. Of course, Microsoft does not intend that anyone would actually write XAML: it is more the output notation from the GUI-builder of Visual Studio. There is nothing intrinsically cross-platform in XAML, since it still relies on System.Windows.Forms for events and rendering.

Thus XUL and XAML are variations of the stages represented by Figure 1(a) and (b). The big difference between them and Views is that both XUL and XAML allow (but do not compel) the programmer to embed event-handling code (JavaScript, and any .NET language, respectively) within the user interface declaration. The Views model, on the other hand, provides an engine that intercedes on behalf of the GUI to signal events to the host application. While the functionality offered by XUL and XAML is attractive, we contend that the separation of concerns evinced by Views' engine-based approach is cleaner and offers greater maintainability and ease-of-use to the programmer and designer.

7.3 Other multiplatform toolkits

We have already mention in Section 1 the efforts to extend platform independence beyond GUIs [11, 6] and the ports to Mono of Gtk# and Qt#. It will be interesting to see if the idiom of these toolkits becomes so entrenched with the .NET Linux community, that XAML will not in the end gain wide acceptance.

8. CONCLUSION AND FUTURE WORK

ViewsQt is a conversion of an XML-based GUI library to support a retargetable back-end. The project involved extracting the common front-end elements of XML checking, parsing, and abstract control creation from the original Views engine, and replacing references to the Windows Forms library classes with calls to a C# interface. This interface hides the toolkit-specific back-end components behind a small (and easy to learn) set of methods. Finally, we created an implementation of this interface for the Qt windowing toolkit, and provided a set of classes to delegate calls from the C# objects to their counterpart C++ objects.

Experiments have shown that the ViewsQt code is portable, with only a few changes to the C++ classes (related to interface inclusion and entry-point specification) required to compile and execute the code on the Linux and Mac OS X operating systems. On the Windows platform, ViewsQt works well with both the .NET Framework and Rotor.

Future work on ViewsQt will entail smoothing out a few wrinkles with regards to the colour and font properties of the controls, and perhaps adding support for more controls that the Views specification does not cater for (e.g. menus, status- and tool-bars). Possibly, an implementation using a second toolkit such as GTK+ will be undertaken to prove the actual retargetability of the front-end.

It is also our intention to exercise the methodology here on libraries other than simple GUIs. Examples would be speech synthesis, or the tangible user interfaces, which are attracting attention.

At the time of writing, an exciting development is the complete rewriting of Views in .NET 2, based entirely on reflection. The prototype system is operational, and is about one-sixth the length of the original because actual controls are picked up directly by name from the XML specification, rather than going through a program transformation. We will be investigating whether the same leverage can be obtained for Qt, and hence for any third part toolkit.

ACKNOWLEDGEMENTS

This work was supported by Microsoft Research and THRIP Grant no. 2788. We would like to thank David-John Miller for his enthusiastic assistance with the project, and acknowledge the inspiration of Nigel Horspool of the University of Victoria, who wrote Views. Johannes Eickhold, Mathhias Kempka and Mihael Vrbanec of TU Karlsruhe were responsible for the port of ViewsQt to Linux.

REFERENCES

[1] Niel M Bernstein, Using the Gtk toolkit with Mono, *O'Reilly ONDotNet*, online article 2004/08/9/ August 2004.

[2] Judith Bishop and Nigel Horspool. **C# Concisely**. Addison Wesley, 2004.

[3] Judith Bishop and Nigel Horspool. Developing principles of GUI programming using Views. Proc. *ACM-SIGCSE*, 373-377, March 2004.

[4] Judith Bishop, R. Nigel Horspool, and Basil Worrall. Experience with integrating Java with C# and .NET. *Concurrency and Computation: Practice and Experience*. To appear, June 2005.

[5] Martin C. Carlisle and P. Maes. RAPID: A Free, Portable GUI Designer for Ada, *SIGAda '98*, 158-164, ACM, 1998.

[6] Martin C Carlisle, A truly implementation independent GUI development tool, *Proc. SIGAda '99*, 47 - 52 , ACM, 1999

[7] D.D. Cowan et al. CIRL/PIWI: A GUI toolkit supporting retargetability. *Software—Practice and Experience*, 23(5):511–527, 1993.

[8] Paulo Pinheiro da Silva. User interface declarative models and development environments: a survey. Proc. *DSV-IS2000*, LNCS 1946, 207–226, Springer-Verlag 2000.

[9] ECMA Standard 335: Common language infrastructure (CLI), December 2002.

[10] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET common language runtime, Proc. *ACM SIGPLAN PLDI*, 1-12, June 2001.

[11] Scott R. Klemmer *et al*, Papier-Mâché: toolkit support for tangible input. CHI 2004: Proc. ACM Conf. on Human Factors in Computing Systems, *CHI Letters*, 6, 399–406, 2004.

[12] Wei-Meng Lee, Writing Cross-Platform Mobile Applications Using Crossfire, *O'Reilly ONDotNet*, online article 2004/07/12, 2004

[13] C.H. Levy et al. IUP/LED: A portable user interface development Tool. *Software—Practice and Experience*, 26 (7):737–762, 1996.

[14] C. Lüth, B. Wolff, TAS — A generic window inference system, 13th Conf on *Theorem proving and higher order logics, in LNCS* 1869, 405-422, Springer-Verlag 2000.

[15] Egbert Schlungbaum. Individual User Interfaces and Model-Based User Interface Software Tools. Proc. *ACM Intelligent User Interfaces IUI'97*, 229–232, Orlando, Florida, USA, January, 1997

[16] Meurig Sage, FranTk – a declarative GUI language for Haskell, Proc. 5th *ACM SIGPLAN conf. on Functional Programming*, 106–117, 2000.

[17] Basil Worrall, Building a retargetable XML GUI toolkit, *Polelo technical report* #6–2004.

WEB REFERENCES (checked 14/2/2005)

Avalon	msdn.microsoft.com/longhorn/ understanding/pillars/avalon/
CLI	www.ecma-international.org
Debian	www.debian.org
Gtk#	gtk-sharp.sourceforge.net
Gtk+	www.gtk.org
Gyro	research.microsoft.com/projects/clrgen/
Longhorn	longhorn.msdn.microsoft.com
Mono	www.go-mono.com
Qt	www.trolltech.com
Qt#	qtcsharp.sourgeforge.net
Tcl/Tk	www.tcl.tk
Rotor	msdn.microsoft.com/net/sscli/
Views	views.cs.up.ac.za
ViewsQt	sourceforge.net/projects/viewsqt/
XAML	link from Avalon page
XUL	www.mozilla.org/projects/xul