A Cross-Toolkit GUI Development Solution

Nandakumar Edamana
Department of Computer Science,
University of Calicut,
Malappuram, Kerala, India
ORCID: 0000-0002-1014-6658
Email: nandakumar@nandakumar.co.in

***Abstract*—A Widget Toolkit is a software library that helps a programmer develop applications that has Graphical User Interface. There exist multiple toolkits, which differ in various aspects, which are incompatible with each other. In this paper, I summarize the initial phase of my attempts to create a highly automated cross-toolkit application development strategy and related tools.**

***Index Terms*—Graphical user interface, code reusability, declarative user interface model, portability, widget toolkits, Gtk+, Qt, Android, Prayogam**

I. Introduction

Being robust, correct, and efficient are some of the important qualities that a software product should have. However, in order to serve a wide userbase, it has to be user-friendly and cross-platform also. User-friendliness, at its lowest level, implies one doesn't have to spend a lot of time figuring out how to get the basic functionalities of the program working. Graphical User Interface (GUI) is a key factor that makes an application user-friendly, if it is intended for non-technical users.

Being cross-platform implies that one doesn't have to switch to a different operating system, CPU architecture, or even another desktop environment in order to install or use the program. This is important because not even technical users would like to switch from a platform that they are attached to.

To present the user with a GUI, application developers usually seek the help of a widget toolkit. And in order to make the application cross-platform, developers are sometimes forced to use multiple toolkits, whcih makes the development and maintenance hard.

A. Widget Toolkits

A Widget Toolkit is a software library that helps applications interact with the user via GUI. Its services include drawing widgets (or controls; e.g.: buttons, text entry boxes) on the screen, updating them based on the requests from the application as well as the actions of the user, calling back the application on events like mouse click, etc. Popular examples include GTK+ and Qt.

B. Cross-Platform Development vs Cross-Toolkit Development

Cross-Platform Development is a term widely used by developers to refer to the practice of making a software package available on different operating systems, architectures, etc., with little or no modification. This is usually accomplished with the help of:

- Programming languages with compilers and interpreters available for multiple platforms;
- Shared libraries that are available for multiple platforms;
- Platform-specific wrappers and macros written by the developer himself.

When it comes to GUI development, one has to rely on some *Widget Toolkit* library in order to present a graphical user interface in front of the user. One can use low-level APIs including Xlib or the Windows API, or some high-level ones like GTK+ and Qt. High-level toolkits usually tend to be cross-platform. But this is not necessarily true always. For example, Android API is high-level yet platform-specific, while Xlib is low-level but cross-platform.

By *Cross-Toolkit Development*, I refer to the practice of writing a program once, and compiling it against different widget toolkits. None of those toolkits need not be cross-platform, but the application as a whole will be cross-platform, if it is compiled against at least one toolkit for each desired platform.

C. Need of Cross-Toolkit Development

The primary goal of some toolkits is to make the application platform-independent, and one may ask why do we need a cross-toolkit development solution. The answer is, portable libraries always introduce runtime dependency or overhead. Also, they may not integrate well with every platform, despite being in a "working" condition. The only solution to this problem is to use the native (or recommended) GUI API of each platform. To accomplish this without having written separate code for each toolkit, we need some cross-toolkit solution.

D. Case Studies Shwoing the Need of an Automatic Tool

As stated in the last secion, we need some tool to automate the cross-toolkit development process. Two examples are listed here to substantiate this.

1) Transmission BitTorrent Client: Transmission BitTorrent Client is an example for projects that use multiple toolkits (separately though). It has macOS, GTK+, Qt, Web and console front-ends. However, a quick analysis of the source code (Transmission 2.94 code from https://transmissionbt.com/download/, retrieved 18 Feb 2019) reveals that each front-end

has a bunch of source files maintained for it, and they are not generated from a single source.

2) moneyGuru: The developer claims moneyGuru to have followed an approach better than Transmission to be cross-toolkit. It is true that the project uses less toolkit-specific code following the Model-View-Controller model, but the idea of a GUI code generator has not been implemented [2].

3) Subsurface: Subsurface was an application developed with GTK+, and it had to migrate to Qt eventually [5]. Such migrations occationally happen, either because the developers wanted to port the program to a new platform, or because the current toolkit didn't satisfy their requirements, or because there was a version upgrade with the toolkit to be cared of. Manual migration is often tedious, even to a newer version of the same toolkit.

E. Why Unification is Impossible (or Unwanted)

Some beginner GUI developers might get frustrated with the presence of multiple toolkits and incompatibilities between them. He may ask, *why we still don't have a single GUI toolkit that can be used everywhere?*

A universal toolkit should be impossible, for differnet people have different tastes and needs. Technically speaking, these are the main reasons why no toolkit has become the de-facto standard:

- Different people like different **programming languages**, and different toolkits have different language bindings.
- Some toolkits perform well on some **Operating Systems** (e.g.: GTK+ on GNU/Linux).
- Some toolkits perform well on some **Desktop Environments** (e.g.: GTK+ on GNOME and Qt on KDE).
- Some toolkits perform well on some **devices** (e.g.: Android UI on mobile devices).
- Different toolkits might have **different optimizations**: speed, memory, appearance, and ease of use.

There will be no universal toolkit unless somebody can come up with a universal operating system, a universal desktop environment, and a universal programming language whose implementations satisfy all the requirements of everyone. But diversity is good.

F. Possible Approaches

Following are some possible approaches towards cross-toolkit development. All of them were carefully considered before choosing one for implementation.

1) Developing a wrapper library for each toolkit
2) Writing wrapper macros for each toolkit
3) Developing a dynamic library that converts UI markup to the calls for the toolkit in use
4) Source-to-source compilation
5) Code generation from a common UI description file

Developing a wrapper library means implementing the API of one toolkit with functions from all the others. For example, if a GTK+ wrapper for Qt could be written, then a GTK+ program can be compiled against this library (instead of the original *libgtk*) so that all the GTK+ calls are mapped to Qt calls, and the program ultimately uses Qt. But such a library (or set of libraries) means either runtime or statically-linked overheads.

Writing macros or inline functions can avoid overheads, but it may prove impractical when a variety of programming languages are considered. Macros and inline functions are thin wrappers and all the programming languages have to be compatible for them to work.

The third approach, UI markup to function call conversion on-the-go would be a highly portable approach, but it has the disadvantage of runtime dependency or overhead.

Source-to-source compilation will be a clean solution that doesn't require the programmer to learn anything new. But developing such a compiler involves writing something that resembles a decompiler, for it has to first gather the high-level requirements from a relatively low-level source code (as an example, consider GTK+ code written in C).

And that leaves the option of generating code for different toolkits for a common UI description file. This is a good choice, for it has zero runtime dependency and zero performance impact (if implemented correctly). The next question is, which UI description language should such a code generator accept.

II. Designing a Cross-Toolkit Solution

If one is to design a cross-toolkit GUI development solution, there can be many challenges. However, there are some factors that make the task easy also.

A. Challenges

The key challenge is the degree of differences between each toolkit in various aspects. They include:

- Linguistic
- Widget classes and hierarchy
- Property names
- Event handling
- Auxilliary UI description languages (GtkBuilder XML for GTK+, XUL for Qt, etc.)

The others include the lack of documentation for some toolkits and some being proprietary.

B. Helping Factors

However, there are some helping factors also:

- There exist cross-platform compiler toolkits like gcc.
- There exist technologies like JNI (Java Native Interface) to enable the integration of modules written in Java with the ones written in C or C++.
- Widget toolkit tend to have some fundamental similarities.

Fundamental similarities include basic ideas like widget inheritance (for example, a Toggle Button inherits a generic Button).

III. NGUIGEN

nguigen is a project started by me with the goal of developing a cross-toolkit GUI code generator. In its initial phase, it tries to generate C/GTK+, C++/Qt and Java+XML/Android code from a single UI description file. The file format is custom, but in the future it aims to support existing UI markup formats and even code written in C as the input. More toolkits and platforms can also be supported.

A. Differences from the Existing UI Markup Systems

UI Description Languages (UIDL) already exist, and they are considered to be a promisiong solution for flexible GUI development. [8] They are also known as *UI Markup Lnaguages* (as Wikipedia calls it). They include Mozilla's XUL (pronounced *ZOOL*) and Qt's QML. However, a new solution was still required because:

- Some are proprietary
- Most of them are toolkit-specific
- Most are XML-based, and sometimes hard to maintain
- Have runtime dependencies (i.e., they are not intended for precompiling)

The last problem could be solved by developing a compiler, but it still cannot produce satisfactory cross-toolkit output if the underlying language doesn't have such a goal.

Declarative UI models are observed to have the goals of programming language independece and operating system independence [1]. However, toolkit-independence is not seen to be stated as the primary goal (or even a goal at all).

ViewsQt is a research project that converts XML-based UI desscription written for .NET to Qt, but it doesn't support any other toolkit, and it is unclear whether any runtime dependencies are there (other than Qt) [4] .

B. Workflow

The current workflow of using nguigen is as follows:

1) The application developer creates a UI description file;
2) The developer invokes nguigen with this file as the input;
3) nguigen generates C, C++, Java and XML source files based on GTK+, Qt and the Android APIs
4) The developer optionally manipulates these files and adds his own files to the codebase;
5) The developer compiles the application against any (or all) of the supported toolkits.

Figure 1 gives an overview of how nguigen works.

C. Relation with MVC

The best way to develop applications using nguigen would be to follow the Model-View-Controller model. However, nguigen doesn't enforce this. Once the skeleton code has been generated, the programmer is free to manipulate it in any manner, implementing any model. However, a different approach than MVC might call for more toolkit-specific coding.

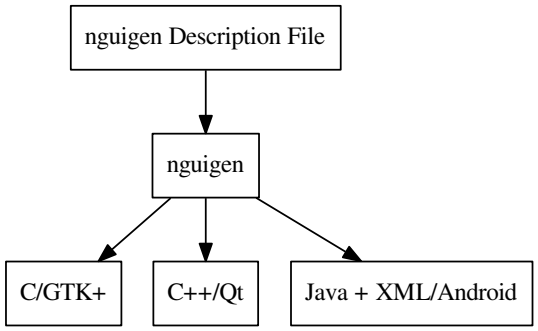

Fig. 1: nguigen overview

nguigen Name	GTK+	Qt	Android
button	GtkButton	QPushButton	Button
entry	GtkEntry	QLineEdit	EditText
hbox	GtkBox	QBoxLayout	LinearLayout
label	GtkLabel	QLabel	TextView
vbox	GtkBox	QBoxLayout	LinearLayout
mainwindow	GtkWindow	QMainWindow	Activity

TABLE I: Mapping of widget names

D. Toolkits, Platforms and Features Supported

In the initial phase, the toolkits in consideration were GTK+, Qt, and the Android API only. Platforms included GNU/Linux (GNOME and KDE), ReactOS (to emulate Qt on Windows) and Android 4.4.2 for Android API. Only a basic set of components, properties and events were in consideration.

E. Language Selection for the Experiment

nguigen generates code in the following languages. Each toolkit has some native language and that was the criteria for the selection.

- C for GTK+
- C++ for Qt
- Java and XML for Android API

F. Widget Nomenclature and Mapping

Only a handful of widgets were in consideration in this stage, and even their mapping was challenging (by mapping, I mean choosing the right class in the target toolkit to implement a particular widget). Table I shows the mapping of a selected number of widgets. Table II and III shows property mapping and event mapping respectively, again only a few samples to illustrate the point.

nguigen Name refers to the nomenclature used in the UI description file given as the input to nguigen. In the absence of standardization, the current approach is to follow the GTK+ nomenclature, and use the names from other toolkits while using toolkit-specific features.

nguigen Widget	nguigen Property Name	GTK+	Qt	Android
button	label	label	text	text
label	label	label	text	text
mainwindow	title	title	title	title

TABLE II: Mapping of property names

nguigen Widget	nguigen Event Name	GTK+	Qt	Android
button	clicked	clicked	released	Click

TABLE III: Mapping of event names

G. Input File Format

Although reusing an existing format is the best choice, incompatibility between different toolkits forces us to do more research before adopting an existing one. Hence a tmporary format has been designed to create the UI description files, which can be given as the input to nguigen.

As per this format, the UI description is a simple text file that names a widget, lists it properties, and repeats the same for another widget. The syntax is shown in Listing 1.

Listing 1: Syntax of nguigen UI description file

```
IND WIDGETNAME
IND class       WIDGETCLASS
IND PROPERTY1   VALUE
IND PROPERTY2   VALUE
IND ...
IND PROPERTYN   VALUE
```

IND refers to indentation, which must be empty for the root widget. Children are placed after a parent, and has an incremented level of indentation than the parent (explained in §III-I).

The line specifying the class of the widget is optional if it can be guessed from the prefix of *WIDGETNAME*. nguigen currently accepts a number of prefixes including *btn_* for button and *lbl_* for label.

H. Widget Creation

All the three toolkits in consideration–GTK+, Qt and the Android API–support both hardcoded widget creation and some UI description language based on XML or similar formats. Although nguigen aims to produce output in both ways eventually, only one mode per widget kit is supported in this early stage:

- C code for GTK+ (i.e., not GtkBuilder)
- C++ code for Qt (i.e., not QML)
- Android Layout XML (i.e., not Java code)

Hardcoding seemed to have performance benefits and better compatibility across versions, and hence the choice for GTK+ and Qt. Although Android API supports hardcoded GUI, XML seemed to be the de-facto standard. Also, XML-based layout is much easier to access from multiple classes (using `R.id.NAME`).

I. Containment (Having Child Widgets)

One widget containing one or more children is an important idea used throughout in GUI programming [1].

nguigen finds the parent-child relationship by looking at the level of indentation. If Widget B that is described after Widget A has a higher level of indentation, then Widget B is a child of Widget A.

Listing 2: Widget containment by indentation

```
hbox_search

        entry_query

        btn_search
        label   _("Search")
        on_clicked      on_btn_search_clicked
```

When the code shown in Listing 2 is processed, what we'll get is a horizontal container with one text entry and a button as its children. Multi-level containment is also possible, which is not shown in the example.

The code generated are shown in listings 3, 4 and 5, excluding the code for event handler connection (explained in the next section).

Listing 3: C code generated by nguigen for GTK+

```
GtkWidget *hbox_search;
GtkWidget *entry_query;
GtkWidget *btn_search;

hbox_search = gtk_box_new(
    GTK_ORIENTATION_HORIZONTAL, 0);

gtk_widget_show(hbox_search);

entry_query = gtk_entry_new();
gtk_container_add(GTK_CONTAINER(hbox_search),
    entry_query);

gtk_widget_show(entry_query);

btn_search = gtk_button_new_with_label(_("
    Search"));
gtk_container_add(GTK_CONTAINER(hbox_search),
    btn_search);
gtk_button_set_label(GTK_BUTTON(btn_search), _
    ("Search"));
```

Listing 4: C++ code generated by nguigen for Qt

```
hbox_search = new QHBoxLayout_hbox_search();
entry_query = new QLineEdit_entry_query();
hbox_search->addWidget(entry_query);
btn_search = new QPushButton_btn_search();
hbox_search->addWidget(btn_search);
btn_search->setText(_("Search"));
```

[1]The term *hierarchy* is not used to avoid confusion with the idea of one class of widget inheriting the properties of a more general one (e.g.: a Toggle Button inherting a generic Button).

Listing 5: XML code generated by nguigen for Android

```
<LinearLayout xmlns:android="http://schemas.
    android.com/apk/res/android"
 android:id="@+id/hbox_search"
 android:layout_width="match_parent"
 android:layout_height="0dp"
 android:layout_weight="1"
 android:orientation="horizontal"
>
</ToolBar>
<EditText xmlns:android="http://schemas.
    android.com/apk/res/android"
 android:id="@+id/entry_query"
 android:layout_width="0dp"
 android:layout_height="match_parent"
 android:layout_weight="1"
>
</EditText>
<Button xmlns:android="http://schemas.android.
    com/apk/res/android"
 android:id="@+id/btn_search"
 android:layout_width="0dp"
 android:layout_height="match_parent"
 android:layout_weight="1"
 android:text="Search"
>
</Button>
</LinearLayout>
```

J. Event Handlers

Changes in names is not the only challenge when it comes to event handling. Apart from having its own nomenclature, each toolkit has its own way of creating and registering event handlers. For example,

- GTK+ accepts regular functions as event handlers.
- Qt requires you to derive a widget class in order to define event handling methods.
- In Android, you have to create event listener classes or override some methods sometimes.

Moreover, what parameters these handlers receive on callback and what velue they should return also vary.

As of now, nguigen invokes the specified callback function for GTK+ and Qt, with minimal information (such as the target widget of the event) passed as the arguments. However, since direct reusability is impossible, the Android output will only have a skeletion hanlder which the programmer has to populate.

Listing 7, 8 and 9 shows the code generated by nguigen for GTK+, Qt and Android respectively.

Listing 6: Input registering the event and specifying the name of the handler

```
btn_search
on_clicked      on_btn_search_clicked
```

Connection of GTK+ event handlers is straightforward. The event name and the handler function name from the nguigen UI description file could be directly passed to the function *g_signal_connect* [6].

Listing 7: Auto-generated event handler connection code for GTK+

```
g_signal_connect(
  G_OBJECT(btn_search),
  "clicked",
  G_CALLBACK(on_btn_search_clicked),
  NULL);
```

For Qt, the widget itself has to be implemented as a derived class. Connecting the handler is done in the constructor while the callback method is implemented as a *Qt slot* [7].

Listing 8: Auto-generated event handler connection code for Qt

```
class QPushButton_btn_search :
public QPushButton
{
  Q_OBJECT

  public: QPushButton_btn_search() {
    connect( this,
      SIGNAL(released()),
      this,
      SLOT(onReleased()) );
  }

  public slots:
  void onReleased() {
    on_btn_search_clicked();
  }
};
```

In Android, creation of a derived class is not necessary, but creating at least one listener class per widget is needed (can be inlined also). [3]. For practical reasons, nguigen creates one class per event.

Listing 9: Auto-generated event handler connection code for Android

```
/* In the class MainActivity: */
findViewById(R.id.btn_search).
    setOnClickListener(new
    btn_search_on_clicked_Handler());

/* In the file Handlers.java: */
class btn_search_on_Click_Handler implements
    View.OnClickListener {
  @Override public void onClick(View v) {
    /* Event handling code */
  }
}
```

K. Implementation Details

A prototype of nguigen is implemented as two bash scripts and supporting Makefiles. bash is selected because it supports grep, sed, etc. for easy parsing of the UI description file; and one can easily mimic some advanced data structures (like trees) using the filesystem and bash can handle it directly. However, this method has several limitations, and bash is intended to be replaced in a later stage.

The two bash scripts are namely *gen0.sh* and *gen1.sh*. The first file parses the UI description file and generates a set

of temporary files, organized in subdirectories to abstract the parent-child relationships between widgets. Now *gen1.sh* can read these files and produce toolkit-specific codes.

1) Development Environment: The key components of my developing environment were:

1) **Debian GNU/Linux 9 and Ubuntu 18.04**
2) **bash** 4.4.19
3) **gcc cross-compilation toolchain** (gcc 7.3.0)
4) **libgtk-3-dev** 3.22.30
5) **qt4-dev-tools** 4:4.8.7
6) **android-sdk** 25.0.0 from the Debian repository

2) Test Environment: The key components of my testing environment were:

1) **Debian GNU/Linux 9** representing Unix-like systems
2) **ReactOS** representing Windows-like systems
3) **Android 4.4.2**

Hardware doesn't matter much and hence not listed here.

L. Current Status

As of now, the tool is successful in illustrating the possibility of a cross-toolkit GUI generator. The code generated by it can be readily compiled against GTK+, Qt and Android, with very minimal toolkit-specific code. However, it is still in prototype stage, and requires platform-specific input to be used to develop something of practical importance.

M. Future Directions

nguigen can prove highly useful, if developments in the following directions could be made:

- Support for existing UI description languages
- Implementation in a more suitable language
- Source-to-source compilation
- Support for more toolkits
- Support for more programming languages

IV. Prayogam: A Sample Product

Prayogam is a very simple application developed by me as an illustration of the practicality of my cross-toolkit approach. The application has a built-in knowledgebase that teaches the practical use of various computer-related tools and techniques (for instance, *how to use the gcc command*). This is why the application is named Prayogam, for the word in various Indian languages means *Application (applying something)*.

The key components of the UI are:

- A listbox to display tags and titles ("Focuslist")
- A textview to display the content of an article
- Some navigation buttons
- A search box

The code to draw these UI components were generated by nguigen, fully for Android and partially for GTK+ and Qt.

Although nguigen is cross-toolkit, Prayogam has features that are still not fully supported by it. Hence the focus was restricted to Android, and nguigen was gradually modified to support all the features that Prayogam required, at least for the Android output. The core (Model in MVC) was written in C, and was integrated with the Java code using JNI.

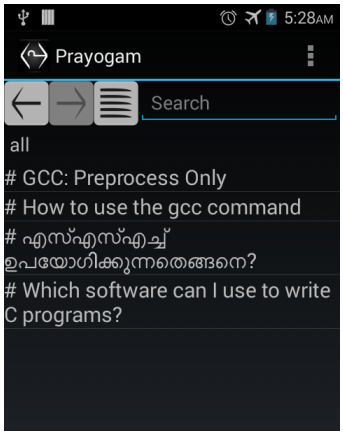

Fig. 2: Screenshot of Prayogam

Some Java code was also generated by nguigen, which was manipulated manually later. *Handlers.java*, mostly generated by nguigen, implements the Controller part, while *Functionality.java*, written manually, implements the View part, whose only job is to refresh the UI by fetching status information and data from the Model part.

V. Conclusion

It is clear that being cross-toolkit will help developers gain wider userbase. A cross-toolkit GUI code generator is needed to automate this process. Apart from making a program cross-toolkit, it can also help avoid situations like toolkit migration.

nguigen, as an experiment, has proved it possible and useful to have such a code generation system. It currently supports C/GTK+, C++/Qt and XML+Java/Android outputs. Prayogam is an example product that illustrates the practicality of nguigen.

If developments listed under §III-M could be achieved, nguigen could establish itself a successful cross-toolkit GUI development solution.

References

[1] Judith Bishop. Multi-platform user interface construction: A challenge for software engineering-in-the-small. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 751–760, New York, NY, USA, 2006. ACM.

[2] Virgil Dupras. Cross-toolkit software. 2009. https://www.hardcoded.net/articles/cross-toolkit-software.

[3] Google. Input events overview. https://developer.android.com/guide/topics/ui/ui-events.

[4] Basil Worrall Judith Bishop. Towards platform independence: retargeting gui libraries on .net. In *.NET Technologies2005 conference proceedings*, 2005.

[5] FOSDEM Website Maintainers. Video cover page of from gtk to qt: An strange journey, part 2. 2017. https://archive.fosdem.org/2017/schedule/event/desktops_from_gtk_to_qt_subsurface_mobile/.

[6] The GNOME Project. Signals: Gobject reference manual. https://developer.gnome.org/gobject/stable/gobject-Signals.html#g-signal-connect.

[7] The Qt Project. How to use signals and slots. https://wiki.qt.io/How_to_Use_Signals_and_Slots.

[8] Orit Shaer, Robert J. K. Jacob, Mark Green, and Kris Luyten. User interface description languages for next generation user interfaces. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '08, pages 3949–3952, New York, NY, USA, 2008. ACM.