Can GUI implementation markup languages be used for modelling?

Carlos Eduardo Silva and José Creissac Campos

Departamento de Informática, Universidade do Minho & HASLab, INESC TEC
Braga, Portugal
{cems,jose.campos}@di.uminho.pt

Abstract. The current diversity of available devices and form factors increases the need for model-based techniques to support adapting applications from one device to another. Most work on user interface modelling is built around declarative markup languages. Markup languages play a relevant role, not only in the modelling of user interfaces, but also in their implementation. However, the languages used by each community (modellers/developers) have, to a great extent evolved separately. This means that the step from concrete model to final interface becomes needlessly complicated, requiring either compilers or interpreters to bridge this gap. In this paper we compare a modelling language (UsiXML) with several markup implementation languages. We analyse if it is feasible to use the implementation languages as modelling languages.

Keywords: User Interfaces, Modelling, Markup languages

1 Introduction

With the current plethora of available devices, applications need to migrate and adapt between those different devices. This is particularly the case for Web applications, which might be run in a diversity of browsers and form factors. We are particularly interested in the possibility of automatically performing the adaptation of user interfaces (UI). For example, in the context of interactive kiosks or public screens.

Model-based User Interface Development (MBUID) is a basis for the solution to this problem. Models (and model transformations) provide a means to both, reason about the design of the user interface, independently from concrete implementation details, and to refine the models to concrete user interfaces. Work in MBUID is based on the existence of adequate modelling languages, known as User Interface Description Languages (UIDLs) [8, 14]. A typical UIDL will support describing an interface at several levels of abstraction, and performing transformations between those levels. The Cameleon Reference Framework for model-based development of multi-target user interfaces [2] identifies four such levels:

– Concepts and Task model — describes the tasks to be performed and the entities users manipulate in their fulfilment;

- Abstract User Interface (AUI) — describes the UI independently of any concrete interaction modality and computing platform;
- Concrete User Interface (CUI) — describes an instantiation of the AUI for a concrete set of interaction modalities;
- Final User Interface (FUI) — corresponds to the UI that is running on a computing platform either by being executed or interpreted.

A diversity of UIDLs has been proposed over the years, using different language paradigms, and covering a variety of applications areas and interaction styles. In [14] a number of proposals is described, with foci as diverse as user interfaces for safety critical systems [11], tangible interaction [15], or 3D user interfaces [16]. Notations used are a mix of textual markup languages and graphical notations. Markup languages, in particular, have gained considerable popularity (see [8] for a survey). Relevant languages in this category include: UIML, which supports both device independent and modality independent UI descriptions [9]; XIML, currently in development by Redwhale Software [13]; MariaXML, the successor of TeresaXML, supports Rich Internet Applications (RIAs), multi-target user interfaces, and applications based on the use of Web services [12]; and UsiXML, a UIDL that aims to cover all aspects of a user interface, for instance, portability, device independence, multi-platform support, amongst others [10]. Moreover, UsiXML is structured according to the Cameleon reference framework.

Typically these languages will cover some or all the abstraction levels in MBUID, from the Concepts and Task to the CUI models. The FUI will be obtained either by interpretation of the CUI models, or by compilation into some target implementation language. Since the interpreters are themselves developed in some specific implementation technology, FUIs are in any case expressed in a different technology from the more abstract models.

Implementation languages, however, have also been evolving from the typical imperative languages like C or Java into declarative markup languages such as HTML or XAML. Although markup languages are usually associated with Web applications, other platforms are also adopting them, for example, Android.

Implementation technologies are therefore moving towards solutions that are closer to what is used for modelling. This begs the question of whether a clear separation between modelling and implementation languages still exists, or whether it will be possible to bridge the gap between a FUI and its models. Hence, in this paper we analyse the feasibility of using declarative markup implementation languages at higher levels of abstraction for MBUID.

In order to carry out this analysis, we will compare UsiXML against a number of implementation languages. UsiXML was choosen since it follows the Cameleon Reference Framework. The implementation languages we chose to analyse are: MXML (Flex), XAML (Silverlight) and HTML5 (three of the most used languages in Web applications development); Android XML (to cover mobile applications); and the LZX Markup language from OpenLaszlo, an industry framework that generates Flash and HTML.

The remainder of the paper is structures as follows: Section 2 presents an overview of the different markup languages we are analysing in this paper. Section 3 compares the different languages. Section 4 presents a case study with the markup languages. The paper ends with discussion and conclusions in Section 5.

2 Markup Languages Overview

Markup languages are declarative languages, where the code is written in the form of annotations called tags. Building UIs with declarative languages is a paradigm shift when comparing to imperative languages. Instead of defining how to build the interface, we define what the interface is. For example, in order to build, imperatively, a UI with a window and a button, we would first build a window, then build a button, and afterwards define that the button is inside the window. Building the same UI declaratively we would define a window, and a button inside the hierarchy of the window. Another aspect that made these languages prosper is their easier understandability, especially by non programmers.

This section presents an overview of each of the markup languages chosen for this analysis: UsiXML, MXML, XAML, HTML5, Android XML and LZX. These are all XML-based markup languages, providing tags for describing different input/output controls (buttons, labels, input fields, etc.) and containers, supporting the definition of a UI in terms of the components that it contains. Usually they will be associated with some technological framework, responsible for rendering the interface and for more advanced features such as expressing behaviour (typically through a scripting language). Since some of these markup languages have more than one technology available to create the UI, we chose one of them to analyse. For example, for XAML we considered Silverlight.

2.1 Modelling languages — UsiXML

The USer Interface eXtensible Markup Language (UsiXML) [10] is a UIDL that supports the description of user interfaces at the different levels of abstraction identified in the Cameleon reference framework. In particular, it supports the creation of domain models, task models, AUI models and CUI models.

The language supports multi-context and multi-target UI development through transformations either between abstraction levels (reification/abstraction), or through changes of context at the same abstraction level. The notion of context is dependent on the specific details of a given development, but might include the users, the technological platform, and/or the environment in which the interaction takes place.

Available tags change between the different models, as their concepts are different. For example, a control tag in a AUI model might correspond to a button tag in a CUI model.

2.2 Implementation technologies

MXML (Adobe Flex) In 2004 Macromedia introduced its framework to develop RIAs, named Flex. Flex can be seen as a developer driven framework to produce Flash content. Flex applications produce as output Flash files (*.swf*) and thus run just like Flash applications. In November 2011, Flex became open-source as Adobe donated it to the Apache Software Foundation.

Flex is composed of a scripting language (ActionScript) and of an XML markup Language (MXML). Their relationship is similar to the relationship between JavaScript and HTML. MXML contains the tags expected of an implementation declarative language.

XAML (Silverlight) In order to merge the benefits of the Windows Presentation Foundation (WPF), Microsoft's desktop application user interface framework, with the RIAs' benefits, Microsoft developed Silverlight. It brings applications similar to the ones developed in WPF to all major platforms through their Web browsers. Silverlight applications run in an ActiveX browser plug-in that is installed in the local machine similarly to the Flash plug-in to run Flash based applications.

The user interface is written in a markup language called eXtensible Application Markup Language (XAML). XAML, although originally developed for WPF, was also adopted as the user interface modelling language of Silverlight and Windows 8 Metro interfaces. XAML has tags for the most common widgets in UI development.

HTML5 HTML5 is the fifth major revision of HTML, the main language of the Word Wide Web. It succeeds the previous version (HTML4), which became a W3C Recommendation in 1997, and aims to improve over that version in order to enable more complex Web pages to be built.

Despite the fact that the HTML5 specification is still under development, the language has gained increased acceptance and support. One of the major driving factors behind its development and acceptance was the increase in the Internet quota of mobile phones.

New features in HTML5 include:

- New semantic elements to better describe a Web page (such as: *nav*, *aside*, *section*, *article*, *header*, and *footer*), in order to diminish the use of the generic *div* tag.
- New multimedia tags have been added, *audio* and *video*, replacing the *object* tag. These tags enable the quick integration of videos from other resources into a Web page. Moreover, multimedia can now be set to preload or to autoplay and can also have integrated controls.
- New attributes were added. For example, the *draggable* and *dropzone* attributes enable support for native drag-and-drop functionality. Another new attribute is *hidden* indicating that the element is not yet/no longer relevant.

– The *canvas* tag was added, which supports bitmap graphics. Most browsers currently support 2D canvas, but there are some experimental builds with 3D canvas support.

Android XML Android is an Open Source platform (Apache License) targeting mobile devices. It is released by Google under the Open Handset Alliance and is based on the GNU/Linux operating system. Android applications are written in the Java programming language. However, instead of using the Java Virtual Machine (JVM), Android uses the Dalvik Virtual Machine, which is optimized for mobile devices.

Unlike the other languages analysed in this document, which require a markup language to develop the UI, or other languages in which the UI is built programmatically, Android allows the UI to be built both ways. The use of markup for development is recommended since it has the advantage of separating presentation from behaviour, thus making the user interface implementation easier to understand. Nevertheless, it is always possible, even for interfaces defined via markup, to build interface objects programmatically at runtime.

OpenLaszlo (LZX) OpenLaszlo is an Open Source platform which enables the development of interfaces using a specific markup language called LZX. It can then generate applications in either Flash or HTML. The goal is to, in the future, enable the platform to produce applications in other languages, for example, Silverlight. Thus, a user interface description in LZX can be seen as a CUI. This makes it relevant to compare it with UsiXML since in both cases concrete languages are intend to be used as a basis to generate UI description in other languages.

The LZX language is an XML-based language, with JavaScript as the scripting language. It was developed to be similar to HTML and JavaScript. However, the declarative language includes some object-oriented programming features such as: inheritance; encapsulation; and polymorphism. Therefore, LZX can have objects, attributes, events and methods like any object-oriented programming language. Moreover, LZX eases data manipulation by allowing data binding to XML elements.

3 Comparing the Languages

When comparing the languages, we were interested in assessing to what level the implementation languages provide features that are similar to the modelling language in question. The goal was not to assess the quality of each language *per se*, but to compare their expressive power and usefulness.

With this goal in mind, two orthogonal dimensions can be considered: the level of abstraction at which it is possible to use a language, and the coverage provided by the language for modelling the different architectural layers of a user interface (Presentation, Dialogue and Functional core — cf. the Arch/Slinky model [6]). Additionally, from a more practical perspective, we were interested in analysing the technological availability and applicability of the languages.

Regarding the level of abstraction, by definition UsiXML covers all levels, while the implementation languages are used to develop actual user interfaces. For the current analysis we will mainly focus on the CUI level. The most obvious candidate for an intersection between the two worlds.

Regarding user interface layers, markup languages are typically used to describe structure. In this case, the presentation layer. Hence, that will be the focus of the analysis. Nevertheless, we mention other aspects where relevant. In particular, support for expressing behaviour. Hence, in comparing the languages the following criteria were considered:

– Behaviour: This criterion captures the different actions that can be performed using the declarative language only, with no scripting involved.
– Style: This criterion defines the type of styling associated with the technology.
– Vector graphics: In the last few years, the availability of a canvas to enable drawing vector graphics has become increasingly important.
– Tags: A comparison of the tags available in each language, taking UsiXML as the reference. The tags we chose to analyse were the ones present in the CUI examples of the FlashiXML tool (a UI renderer for UsiXML).

Regarding technological availability and applicability we choose to consider:

– License type: Depicts the accessibility of the technology.
– Tools: This criterion considers the different tools associated with the languages.
– Targets: These technologies can be available in a single or in several platforms.

Table 1 compares the different languages according to these criteria.

In terms of license, although some of these technologies started as proprietary software, currently the only proprietary one is Silverlight. The tools criterion is the one that differs most from language to language. The only language that simply requires a Web browser to run is HTML5. Flex and Silverlight both require a plug-in to be installed. UsiXML and LZX have tools that either compile the models to other languages, or perform runtime interpretation and rendering. Target technologies include Flash and DHTML in the case of LZX; and Flash (FlashiXML [1]), Flex (FlexiXML [3]), OpenLaszlo (UsiXML2OpenLaszlo), Tcl/Tk (QTKiXML [4]), and Java (InterpiXML [5]), amongst others, in the case of UsiXML. Android applications are the only ones that do not run as, or compile to, Web applications, they run in the Android OS. Therefore, Android is the only analysed technology that is single platform.

Regarding styling capabilities, all the technologies either have CSS styling, or a specific styling done exclusively using markup. UsiXML has a *stylesheet* tag in its specification. There are tools, such as FlexiXML that support CSS for styling. However, the tool we used, FlashiXML does not support styles. All the languages analysed had Vector graphics support.

Table 1. Markup Languages Comparison

Languages	UsiXML	Flex	Silverlight	HTML5	Android	LZX
License	OpenSource	OpenSource	Proprietary	OpenSource	OpenSource	OpenSource
Tools	Interpreters and compilers	Flash plug-in	Silverlight plug-in	Web Browser	Android OS	Compiles Flash and DHTML
Targets	Multiplatform	Multiplatform	Multiplatform	Multiplatform	Singleplatform	Multiplatform
Behaviour	Multimedia, Transitions	Multimedia	Multimedia	Multimedia	Multimedia	Multimedia
Styles	Stylesheet	CSS	Markup	CSS	Markup	CSS
Vector Graphics	Yes	Yes	Yes	Yes	Yes	Yes
Tags	box	Group	StackPanel	div	LinearLayout	view
	gridBagBox	Grid	Grid	table	TableLayout	grid
	textComponent	Label	TextBlock	label	TextView	text
	outputText	Label	Label	label	TextView	text
	inputText	TextInput	TextBox	input type=”text”	EditTex	edittext
	imageComponent	Image	Image	img	ImageView	image
	comboBox	ComboBox	ComboBox	select	Spinner	combobox
	item		ComboBoxItem	option	item	textlistitem
	button	Button	Buttom	button	Button	button
	radioButton	RadioButton	RadioButton	input type=”radio”	RadioButton	radiobutton
	behavior					
	event					
	action					
	methodCall					
	methodCallParam					
	transition					
	graphicalTransition					

Regarding supported tags, the first conclusion when looking at Table 1 is that, unlike UsiXML, the implementation markup languages are not prepared to handle behaviour tags. They handle behaviour by using a non-markup scripting language. On the contrary, UsiXML does not have an associated scripting language. However, UsiXML's behaviour tags handle basic generic behaviour situations, like window transitions, only. Therefore, if more complex behaviour is needed, it falls to the developer of the interpreter or compiler to choose whether or not to support a scripting language. For example, FlashiXML uses ActionScript as the scripting language.

Moreover, the layout tags are the ones that are prone to having most differences between the languages. Languages like HTML do most layout by using CSS, while other languages have styling options and different layout options. For instance, Android has tags for Linear Layout, Relative Layout, Table Layout, Grid View, Tab Layout and List View.

Aside from the behaviour tags, all other tags have correspondence between the markup languages. The only exception is the *item* tag in Flex, which handles *comboboxes* by binding to data collections only, and does not allow the explicit declaration of single items. Therefore, we can do a direct and easy translation between the different markup languages. Such translation would also need to have in consideration the different attributes of the tags.

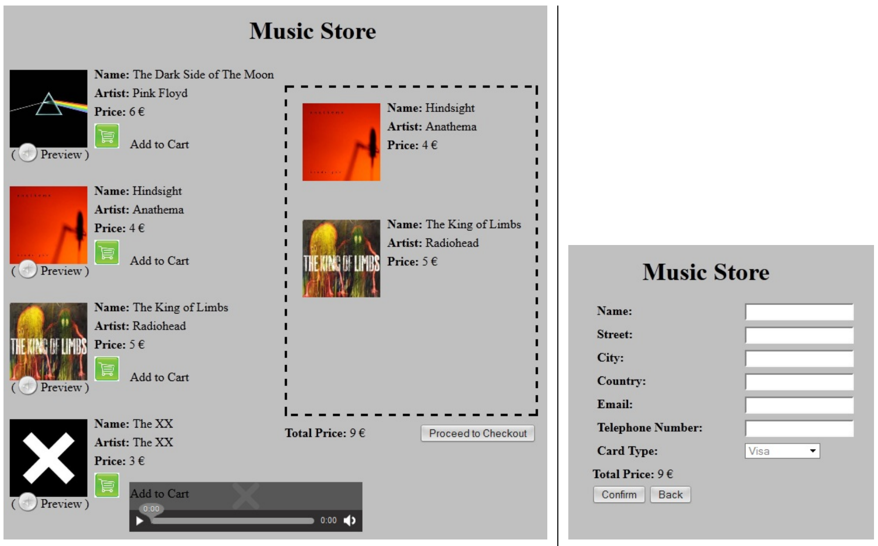
However, the translation between the languages is not always unidirectional. For instance, the *box* tag in UsiXML corresponds better to the *div* tag in HTML5. Nevertheless there are other tags, specifically since HTML version 5, that also correspond to a box in UsiXML but have specific semantic meanings such as the *nav*, *aside*, *section*, *article*, *header*, and *footer* tags.

Moreover, some tags can be changed according to the styling they are given. For example, the *span* tag in HTML5 is an inline element whereas a *div* tag is a block-level element. By changing the styling, we can have a *span* tag behaving as a *div* tag and vice-versa. This aspect is quite difficult to address in these languages translation.

4 Case Study

Since in theory the translation seemed to be feasible, we decided to investigate the issue further by developing a small example application in each of the languages. The goal was to evaluate the markup languages' strengths and weaknesses through a case study. The example application simulates a Web store that sells CDs, called Music Store, and is based on a similar application from the FlashiXML examples suite. The customer is able to add CDs to a shopping cart, by selecting them from a list of available CDs, and afterwards fill his/hers personal details to buy the chosen CDs.

The application is composed of two main frames (see Figure 1). The initial frame (*Shop Frame*), depicted in Figure 1-a), comprises a list of the albums in the music store (left), and a basket (another list) to keep track of the customer's selected items (right). Each album has a preview button which enables a small



Music Store

Name: The Dark Side of The Moon
Artist: Pink Floyd
Price: 6 €
Add to Cart
(Preview)

Name: Hindsight
Artist: Anathema
Price: 4 €
Add to Cart
(Preview)

Name: The King of Limbs
Artist: Radiohead
Price: 5 €
Add to Cart
(Preview)

Name: The XX
Artist: The XX
Price: 3 €
Add to Cart
(Preview)

Name: Hindsight
Artist: Anathema
Price: 4 €

Name: The King of Limbs
Artist: Radiohead
Price: 5 €

Total Price: 9 €
Proceed to Checkout

Music Store

Name:	
Street:	
City:	
Country:	
Email:	
Telephone Number:	
Card Type:	Visa

Total Price: 9 €
Confirm Back

a) Shop frame b) Checkout frame

Fig. 1. Music Store Application

preview of the album to be played. The customer can buy an album either by clicking on the "*Add to Cart*" button next to the album, or by dragging and dropping the album cover into the basket area. When an album is added to the basket, the total value of the shopping cart is updated accordingly.

Once the desired alguns have been selected, the customer can move on to *Checkout* by clicking on the "*Proceed to Checkout*" button. This button replaces the *Shop* frame with the *Checkout* frame, depicted in Figure 1-b).

The *Checkout* frame comprises several text boxes and a listbox, which enable customers to fill in their details, specifically, the name, address and credit card type. Afterwards, the user can confirm the transaction or go back to the previous frame, where he can rebuild the basket list again.

The application screen size is kept small so that the example can also cover mobile applications. This is relevant because we are considering Android XML, which is used for applications running on smartphones and tablets.

With this example we intend to analyse both the languages' capabilities in the context of developing traditional Web applications, with buttons and forms to interact with the user, but also some more advanced RIAs features. RIAs, in comparison to traditional Web applications, present a number of improvements: no page refreshing; shorter response times; drag-and-drop capabilities; multimedia animations. From these improvements, we chose to implement in our application two frames without page refreshing, drag-and-drop, and multimedia, by allowing previewing an album by playing a small audio sample. Next we will present the main aspects that differ in building the application for each language.

4.1 UsiXML

Regarding UsiXML, we developed a CUI model, taking as a basis the original Music Store application. The final user interface will have to be generated using an appropriate renderer. In this case, we chose to use FlashiXML.

UsiXML has several types of layouts for CUI modelling, such as: *box*, *groupBox*, *flowBox*, *gridBox*, *gridBagBox*, and *listBox*. However, the FlashiXML renderer seems to work with only a few of them, therefore, we built the entire application with the *box* tag. This tag is flexible since it has an attribute called type that defines the orientation of the child elements.

FlashiXML cannot handle styling, which is aggravated as in this language every element requires quite a few attributes. For example, the labels for the name and street in the *Checkout* frame were coded as follows:

```
<textComponent id="name" defaultContent="Name:" width="100"
    height="25" borderWidth="0" fgColor="000000" isBold="true"
    textSize="16" textHorizontalAlign="right" numberOfLines="1"/>
<textComponent id="address" defaultContent="Street:" width="100"
    height="25" borderWidth="0" fgColor="000000" isBold="true"
    textSize="16" textHorizontalAlign="right" numberOfLines="1"/>
```

By looking at the code it is noticeable that style sheets would make a significant impact in the coding. Moreover, the same pattern is repeated throughout the whole application.

In terms of behaviour, UsiXML does not have tags for drag-and-drop. Therefore, the drag-and-drop was implemented entirely using ActionScript. The transition between the *Shop* frame and the *Checkout* frame, however, was entirely implemented through the declarative language, as depicted in the following code (that models the *Checkout* button in the *Shop* frame):

```
<button width="100" height="25" isEnabled="true" isVisible="true"
    defaultContent="Checkout" id="button_go" name="button_go">
    <behavior id="behavI8">
        <event id="evtI8" eventType="depress" eventContext="button_go"/>
        <action id="actI8">
             <transition transitionIdRef="Tr1"/>
             <transition transitionIdRef="Tr2"/>
             <transition transitionIdRef="Tr3"/>
             <transition transitionIdRef="Tr4"/>
        </action>
    </behavior>
</button>
```

Four transitions are used (see the *transition* tags): Tr1 and Tr2 are fade-out transitions (of the *Shop* frame and of the background) and TR3 and TR4 are fade-in transitions (of the *Checkout* frame and of the new background). These transitions are defined elsewhere in the model.

Another issue is that UsiXML has a *videoComponent* tag but not a *audio-Component* one. Thus, we assumed that like some other languages, the *video-Componen*t is used in both cases. Nevertheless, the tag was not tested since FlashiXML does not handle these components. A possibility to add multimedia in FlashiXML is to use ActionScript to do the entire process. However, in that case we are relying in a specific implementation technology. Arguably, we would have something closer to a FUI.

4.2 MXML (Flex)

In Flex layout is defined using containers: *Group* behaves like a simple box, *HGroup* arranges the elements horizontally, and *VGroup* arranges the elements vertically. Elements can also be arranged according to relative or absolute co-ordinates. The code bellow shows the coding of the two labels and the button below the drop area in the *Shop*.

```
<s:HGroup x="444" y="491">
    <s:Label  text="Total Price" styleName="labelS"/>
    <s:Label  id="totP" text="0 €" styleName="labelT"/>
</s:HGroup>
<s:Button x="578" y="486" label="Proceed to Checkout"
          click="button1_clickHandler(event)"/>
```

In terms of multimedia elements, these can be handled with tags if using the Flash library which has a *Sound* and a *Video* tag for audio and video respectively. Drag-and-drop is supported exclusively through ActionScript.

4.3 XAML (Silverlight)

Layout design in Silverlight is very flexible. There are built-in layouts like grids, stackpanels or listboxes, but there is also the option of controlling the elements' position by using margin, padding or horizontal and vertical alignments.

Adding multimedia is very straightforward. We add a media element in the XAML, as follows:

```
<MediaElement x:Name="media" AutoPlay="False"
              Source="Sounds/05LotusFlower.mp3" />
```

Afterwards, the event handler uses the following C# code to start playing the audio.

```
  media.Position = TimeSpan.Zero;
  media.Play();
```

This example also shows that to access a XAML element in C# we just need to invoke his name.

Silverlight drag-and-drop has some disadvantages. For instance, only a few elements can have drag-and-drop action controls. Specifically there are the following controls (the names identify their purpose):

– ListBoxDragDropTarget,
– TreeViewDragDropTarget,
– DataGridDragDropTarget, and
– DataPointSeriesDragDropTarget.

Thus, for an element to be draggable, it has to necessary be a child element of one of the previous layouts. A second option would be to achieve drag-and-drop by manually implement the click handlers. A third option would require the use of an external library called Drag and Drop Manager. In our implementation we chose to use the *ListBoxDragDropTarget* control, as follows:

```
<toolkit:ListBoxDragDropTarget AllowDrop="True"
                               AllowedSourceEffects="Copy">
    <ListBox x:Name="Listbox">
        <StackPanel Name="spPF">
            ...
```

Therefore, every album is a *StackPanel* inside a *Listbox*. A difference between this implementation and the applications from the other languages is that instead of dragging just the image, it is possible to drag anywhere in the album area.

4.4 HTML5

The layout in HTML5 is clearly more difficult to define than in most other languages tested in this document. By using CSS, developing the layout feels less natural than using boxes and predefined layouts.

On the contrary, drag-and-drop is easy to implement in HTML5. Just by adding the *draggable* attribute to an element, that element can be dragged across the application. Nevertheless, in order to define where the elements could be dropped (in this case, the shopping basket) a small amount of JavaScript was required.

The new multimedia tags, in this case the audio tag, are very useful. Just by adding the following code:

```
<audio controls="controls" hidden>
    <source src="sounds/05LotusFlower.ogg" type="audio/ogg"/>
    <source src="sounds/05LotusFlower.mp3" type="audio/mpeg"/>
</audio>
```

the audio file is available in the application, and playback and volume controls are added. The controls are depicted in Figure 1, in the black box at the bottom of the main frame. Controls are browser specific. In this particular case the Firefox browser controls are being shown.

A problem with building an application in HTML is the different browsers' reactions to the same code. Furthermore, with HTML5, the browsers have even more differences. For example, in the previous audio tag, both an *ogg* and an *mp3* file were added, since neither the current version of Opera nor of Firefox play *mp3* files. Moreover, browsers are still updating to add the new HTML elements. For example, in Internet Explorer the *hidden* tag does not set the elements to invisible, thus, both frames and controls appear when the application starts. This is expected to improve with time.

4.5 Android XML

Despite mobile phones' screen sizes being much smaller than a traditional computer's screen, we opted to keep the application exactly the same. To compensate for the screen size, we added vertical scrollbars to navigate up and down the albums list, the shopping basket list, and the entire form in the second frame. Another development decision was to keep the layout as the default Android light layout. The major difference from the applications modelled in the other languages is the form, which now has a different look, more appropriate for mobile systems.

The Android XML seems to be more verbose than all the other markup languages analysed in this document. As an example, a simple label would be defined as follows:

```
<TextView android:id="@+id/nameLabel1"
          android:layout_width="50dp"
          android:layout_height="wrap_content"
          android:text="@string/namelabel" />
```

Another interesting characteristic in Android development is that it encourages keeping an XML file name strings.xml where all the strings should be stored. For instance, the string for the label in the above example (*namelabel*) is stored in that file as follows:

```
<string name="namelabel">Name:</string>
```

This separation between the strings and the actual interface's source code, enables one to easily change the strings' contents in the future.

In terms of multimedia, Android doesn't have a tag for audio. Nevertheless it has a tag for video called *VideoView*.

Drag-and-drop in Android is achieved by using the *setOnTouchListener* method in the elements that should be dragged, and then using the method *startDrag()* to enable the drag. The elements that are expecting drops should implement the *onDragListener*. This listener uses a method called *getAction()* which retrieves the current action of the drag. This action can be whether the drag element has entered or exited the drop area, or whether the drag element has been dropped in the drop area. This last action is the one we are interested in in this particular application.

4.6 LZX

The layout and design process in Laszlo is easy both to accomplish and to learn. The main component is called "*view*" which visually is a rectangular container. Obviously there can be nested views, and they are used to organize the elements on the rendered application. Moreover, application elements can be arranged easily on the page, by using layouts. For instance:

```
  <simplelayout axis="x" spacing="6"/>
```

arranges all elements according to the *"x"* axis and with a spacing value of 6 between them. Furthermore, elements can also be placed with relative and absolute positioning like in HTML. Nevertheless, using the boxes for arrangement is more understandable.

OpenLaszlo has a multimedia tag for both audio and video called *videoview*. Moreover, we can also add video and audio as resources of regular views. However, in the current version, multimedia only works when the application is compiled to Flash.

The drag-and-drop of the albums was hard to implement. In Laszlo, we had to implement the methods to start and stop the dragging and also the methods to check if the element where we dropped the image was the correct one. For example, the code for the last method was the following:

```
<method name="droppedInView" args="theView">
    <![CDATA[
        var absX = theView.getAttributeRelative( "x", canvas );
        return (this.x > absX && this.x < absX+theView.width);
    ]]>
</method>
```

The method determines whether the place where an object has been dropped is inside the view sent as a parameter. In this particular case this is calculated by looking at the absolute coordinates of the X axis. The `<![CDATA[` and `]]>` tags allow us to write characters that would otherwise not be possible in XML files (for example the '<' and '>' signs).

4.7 Applications' Comparison

After all the applications were built, we analysed them according to a number of metrics. The results are depicted in Table 2. The first criterion was the number of different tags present in each applications. UsiXML is clearly the one with a greater diversity of tags. Nevertheless, that greater number can be related to having behaviour tags also, which can also be seen in Table 1. HTML5's high value is related to this new version featuring new tags to bring more expressiveness to the language.

Table 2. Application Comparison

Languages	UsiXML	Flex	Silverlight	HTML5	Android	LZX
Number of Different Tags	23	14	12	18	12	17
Total Number of XML lines	215	139	182	182	189	187
Total Number of Scripting lines	70	102	128	85	98	33
Total Number of styling lines	0	6	0	93	0	0
Total Number of lines	285	247	310	360	287	220
Percentage of Scripting lines	24,56	41,3	41,29	23,61	34,15	15
Total Number of Tags	141	107	139	106	132	146
Total Number of Attributes	653	236	646	121	572	186

The second criterion defines the total number of XML lines. The two outliers are UsiXML with the biggest number of lines, and Flex with the lowest number.

The third criterion is the total number of scripting lines. In this criterion LZX is clearly the language that requires less scripting. Mostly due to the fact that the scripting code in LZX is greatly embedded with the XML code. For instance, the code for the Back button in the *Checkout frame* is the following:

```
<button onclick="back();">
    Back
    <method name="back">
        shop.setAttribute('visible',true);
        payment.setAttribute('visible',false);
    </method>
</button>
```

This code shows that not only is the script written inside the method tag but also that the elements (shop and payment) are easily invoked and altered.

The forth criterion shows the total number of styling lines. We decided to do styling only when needed. HTML5 is the language that normally requires styling, mostly for layout purposes. It is also interesting to notice that, although such a high number of lines were used in styling, the XML file size is still similar to that of Silverlight, Android and LZX which had no styling in this implementation.

This leads to the fifth criterion, total number of lines, where HTML5 is clearly the one that requires more lines. While LZX and Flex took the least amount of coding.

The percentage of scripting lines criterion show us how much imperative programming we need comparing with the whole application. Both Flex and Silverlight require a high amount of imperative programming comparing with the rest of the technologies analysed.

In terms of the total number of tags, UsiXML and LZX have more tags than the rest. It is interesting to note HTML5's behaviour since it was one of the languages with most XML lines, but is the one with less total number of tags.

The last criterion is the total number of attributes. In this criterion, UsiXML, Silverlight and Android clearly use many more attributes than the other languages. HTML5, although the language with the higher total number of lines, is in this criterion the one with lowest number of attributes, reflecting the styling effect in these metrics.

While the above numbers do not provide a objective measure of quality of the different languages, they are useful in showing that no obvious differences can be seen between the model developed in UsiXML and the applications developed in the other languages. This corroborates our belief that it is possible to use markup implementation languages as modelling languages at (at least) CUI level.

Another important aspect regards which language to choose. For that choice, aspects such as the learnability of the languages are relevant. The Cognitive dimensions of notations proposed by Green and Petre [7] would be useful to make such analysis. That however was not the specific aim of our study. In any case, if we had to choose an implementation language for modelling, LZX and HTML5 look the most promising since when comparing the percentage of scripting lines required they had even better results than UsiXML.

5 Conclusions

In this paper we have compared different declarative GUI implementation languages with a declarative modelling language. The motivation behind the work is the possibility of performing adaptation of web applications' user interfaces to different devices and form factors. Given that the implementation technology has moved towards declarative markup languages, we were interested in analysing the viability of using the interfaces expressed in those languages as models of the user interfaces.

Looking at the results, we see that not all aspects of a user interface can be handled declaratively. In particular, implementation languages are not prepared to handled behaviour declaratively. This limits the specification of the interface we can perform using declarative languages only. To be fair, this is an issue also in terms of the modelling language, as UsiXML provides few behaviour specific tags too (e.g. transitions). In fact markup languages are, in general, geared towards describing the structural aspects of the user interface.

Some of the languages have several tags to define the same concept. Although this increases the expressiveness of the language, it also decreases the level of abstraction of an hypothetical model. For instance, in HTML5 we can have *div*, *nav*, *aside*, *section*, *article*, *header*, and *footer* tags, all corresponding to a box at a higher level of abstraction. Nevertheless, regardless of the lager number of tags in a language, it still falls to the developer the decision to use them or not. Hence, we can think of defining profiles or dialects of the language for CUI (AUI) modelling. It can be decided, for example, that a box should always be modelled by a *div* tag. This will allow us to embed a modelling language inside a implementation language, taking advantage of all the tool support that is available.

This embedding of a modelling language inside an implementation language is particularly relevant when it comes to animating the models. As the analysis has shown, the fact that specific players are needed for modelling languages, raises a number of issues in terms of support for specific languages features and language versions. While this also happens for the implementation languages, the industry and community support behind languages such as HTML means that evolution of those technologies will be much faster. However, it must be noted that regarding aspects as model transformation and context adaptation, the tool support provided by UsiXML related tools will be lost.

Another aspect that might create difficulties, in particular if we consider deploying the models to different languages, relates to managing layout and expressing behaviour. The languages are very different in terms of these aspects. In fact, the amount of layout options differ significantly from language to language, and while they all resort to scripting to express behaviour, the scripting languages used differ. The answer here might be to look for behaviour oriented languages to complement the models with behavioural information.

In terms of limitations of the analysis, it must be recognised that our analysis was focused mainly in CUIs. The capabilities of implementation languages at higher levels of abstraction, like AUIs, requires further consideration. Moreover, our analysis was targeted specifically at graphical user interfaces (i.e. we have not considered what issues might be raised by other interaction technologies such as the use of multimodality). This happens because the notion of context that interests us the most relates to the form factor of the device displaying the interface.

Another aspect is that nowadays a relevant number of Web applications is built dynamically. That is, the markup used to generate the interface is not written directly by the developer. Instead, code is written that generates (or, at least, manipulates) the markup. This means the markup will only be available at run time, which in turn means that we need dynamic code analysis techniques to be able to obtain and transform the user interface.

Hence, as future work we intend to, on the one hand further develop the notion of embedding a modelling language in a implementation language, and on the other hand, study techniques for the dynamic analysis of the interface in order to extract and transform the models.

Acknowledgments

This work is funded by the ERDF – European Regional Development Fund – through Programme COMPETE, and by the Portuguese Government through FCT – Foundation for Science and Technology, project ref. FCOMP-01-0124-FEDER-015095. Carlos Eduardo Silva is further funded by the Portuguese Government through FCT, grant SFRH/BD/71136/2010.

The authors wish to thank the anonymous reviewers for their helpful comments on an earlier version of this paper.

References

1. Youri Vanden Berghe. Etude et implémentation d'un générateur d'interfaces vectorielles à partir d'un langage de description d'interfaces utilisateur. Master's thesis, Université catholique de Louvain, 2004.
2. Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A Unifying Reference Framework for Multi-target User Interfaces. *Interacting with Computers*, 15:289 – 308, 2003.
3. José C. Campos and Sandrine A. Mendes. FlexiXML - a portable user interface rendering engine for usixml. In *User Interface Extensible Markup Language - UsiXML'2011*, pages 158–168. Thales Research and Technology, 2011.
4. Vincent Denis. Un pas vers le poste de travail unique: QTKiXML, un interpréteur d'interface utilisateur à partir de sa description. Master's thesis, Université catholique de Louvain, 2005.

5. Yann Goffette and Henri Louvigny. Development of multimodal user interfaces by interpretation and by compiled components: a comparative analysis between InterpiXML and OpenInterface. Master's thesis, Université catholique de Louvain, 2007.
6. C. Gram and G. Cockton, editors. *Design Principles for Interactive Software.* Chapman & Hall, 1996.
7. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, 1996.
8. Josefina Guerrero-Garcia, Juan Manuel Gonzalez-Calleros, Jean Vanderdonckt, and Jaime Munoz-Arteaga. A theoretical survey of user interface description languages: Preliminary results. In *Proc. of the 2009 Latin American Web Congress*, LA-WEB '09, pages 36–43, Washington, DC, USA, 2009. IEEE Computer Society.
9. J. Helms, R. Schaefer, K. Luyten, J. Vermeulen, M. Abrams, A. Coyette, and J. Vanderdonckt. Human-Centered Engineering Of Interactive Systems With The User Interface Markup Language. In A. Seffah, J. Vanderdonckt, and M. Desmarais, editors, *Human-Centered Software Engineering*, Human–Computer Interaction Series, pages 139–171. Springer London, 2009.
10. Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. UsiXML: a language supporting multi-path development of user interfaces. In *Engineering Human Computer Interaction and Interactive Systems*, volume 3425 of *LNCS*, pages 200–220. Springer-Verlag, 2005.
11. David Navarre, Philippe Palanque, Jean-Francois Ladry, and Eric Barboni. ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Transactions on Computer-Human Interaction*, 16(4):18:1–18:56, November 2009.
12. Fabio Paternò, Carmen Santoro, and Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction*, 16(4):1–30, 2009.
13. Angel Puerta and Jacob Eisenstein. XIML: a common representation for interaction data. *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 214–215, 2002.
14. Orit Shaer, Robert J. K. Jacob, Mark Green, and Kris Luyten, editors. *ACM Transactions on Computer-Human Interaction Special issue on UIDL for next-generation user interfaces*, volume 16(4), New York, NY, USA, November 2009. ACM.
15. Orit Shaer and Robert J.K. Jacob. A specification paradigm for the design and implementation of tangible user interfaces. *ACM Transactions on Computer-Human Interaction*, 16(4):20:1–20:39, November 2009.
16. Chadwick A. Wingrave, Joseph J. Laviola, Jr., and Doug A. Bowman. A natural, tiered and executable uidl for 3d user interfaces based on concept-oriented design. *ACM Transactions on Computer-Human Interaction*, 16(4):21:1–21:36, November 2009.