FlexiXML
A portable user interface rendering engine for UsiXML

José Creissac Campos
Departamento de Informática/CCTC
Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
jose.campos@di.uminho.pt

Sandrine Alves Mendes
ALERT Life Sciences Computing, S.A.
Rua Daciano Baptista Marques, 245
4400-617 Vila Nova de Gaia, Portugal
sandrine.mendes@alert.pt

ABSTRACT
A considerable amount of effort in software development is dedicated to the user interaction layer. Given the complexity inherent to the development of this layer, it is important to be able to analyse the concepts and ideas being used in the development of a given user interface. This analysis should be performed as early as possible. Model-based user interface development provides a solution to this problem by providing developers with tools that enable both modeling, and reasoning about, user interfaces at different levels of abstraction. Of particular interest here, is the possibility of animating the models to generate actual user interfaces. This paper describes FlexiXML, a tool that performs the rendering and animation of user interfaces described in the UsiXML modeling language.

Keywords
Tool Support, User Interface Description Languages (UIDLs), UsiXML

INTRODUCTION
User interface development is a complex process. In the long run, the success of an interactive system hinges on having considered both the users of the system appropriately, as well as the technologies for its development. Model-based development provides a solution to manage such complexity. In this paradigm, declarative models are created that range from abstract concerns with domain and task knowledge, through the design of the intended dialog, down to the concrete interaction styles and execution platforms to be used.

A model-based approach encourages a more sustainable development process. In particular it allows capturing a rigorous description of the design, thus facilitating the construction of prototypes via the animation of the models. In turn, the development of prototypes fosters a better understanding of a systems design, and facilitates the participation of users in the development process. These prototypes can be repeatedly tested and adapted to the users' needs. Prototyping and testing does not guarantee the absence of errors, but minimizes the likelihood of such errors in more advanced stages of the project, making it possible to assess the users' reactions to the interface design under development.

This paper describes FlexiXML, a new user interfaces rendering tool for the UsiXML modeling language. FlexiXML acts as a renderer and animator, enabling users to interact with an interface expressed in UsiXML.

The rest of the paper is organized as follows. First related work is discussed. Then the UsiXML modeling language is described. Next the technology used to implement the FlexiXML tool is described, followed a description of the tool itself, and an example of application. The paper ends with conclusions and an outline of future work.

RELATED WORK
As stated above, the core language for this project is UsiXML (USer Interface eXtensible Markup Language) [14]. UsiXML is a User Interface Description Language (UIDL), and will be discussed in the next section. Besides UsiXML, other UIDLs dealing with different aspects of a graphical interface have been put forward. Examples include: AUIML (Abstract User Interface Markup Language) [3], that focuses on enabling user interfaces to be deployed to different device types; UIML (User Interface Markup Language) [2], a markup language standardized by OASIS; XIML (eXtensible Interface Markup Language) [1]; IBM's WSXL (Web Service Experience Language) [2]; or Mozilla's XUL (XML User Interface Language) [9]. For an extensive survey of XML-compliant languages for user interface description see the paper by Garcia et al. [12].

The selection of UsiXML was a result of its broad scope, which allows the specification of the different models needed during interface development, and a result of the fact that there is an active community supporting the development of the language. The UsiXML language allows for the interface specification to be made at different levels of abstraction, and provides transformation mechanisms between them. Of particular interest in this context are models that deal with the design of the concrete

user interface (the components that make up the interface, their layout, and behaviour), and the execution context.

To support user interface modeling and manipulation of the models, UsiXML has a set of tools, which can be divided into two categories:

- Editors – Tools for the creation of UsiXML descriptions. The way the descriptions are created varies with the type of tool. Examples of this type of tool include, SketichXML [5] that uses as input hand-drawn interfaces (sketchs), GrafiXML [8] where the interface is created by direct manipulation of components on the screen, VisiXML [5] where the interface design is done in Microsoft Visio, among other tools.
- Interpreters – Engines to generate graphical user interfaces described in UsiXML. Each interpreter generates interfaces with a set of specific characteristics. Examples of this type of tool include: FlashiXML [16] that generates vectorial interfaces, HaptiXML [7] that generates graphical user interfaces in 3D with interaction by touch, QtkiXML [6] setting interfaces for multiple platforms, InterpiXML [15] that allows simultaneous interpretation of various UsiXML descriptions.

Although the UsiXML community already has several tools focusing on different aspects of the language, ongoing development of the language means that it does not have an updated animation tool to facilitate a process of rapid prototyping and analysis. The tool that more closely approximates the desired outcome is FlashiXML. However this tool is not compatible with the current version of UsiXML, and not easily upgradable. Currently UsiXML is in version 1.8, and FlashiXML is compatible with version 1.4.6 only.

This project aims to create a completely new interpreter tool that supports the current version of UsiXML, providing a number of additional features when compared with FlashiXML. Specifically, the main goals are to provide a tool that:

- is capable of both interpreting the latest version of UsiXML, and adapting to new versions;
- is implemented in a platform independent technology enabling web access.

Additionally, we intend to take the opportunity to create a generic and expandable application, as exposed in the FlexiXML Platform section.

Regarding the technologies to implement the tool, the choice was made to use the Adobe Flex software development toolkit. The use of Flex and ActionScript 3 provides a robust solution, with better performance, and a more modular architecture, than what can be achieved by using Flash and ActionScript2. The use of the Adobe's AIR runtime environment allows FlexiXML to be relatively independent of the computing platform where it runs. The only requirement is that there is an AIR runtime for the target platform.

USIXML

UsiXML (USer Interface eXtensible Markup Language) is an XML-compliant markup language for user interfaces (a XML-based User Interface Development Language – UIDL) that describes a user interface independently of programming language, computing platform and working environment. This UIDL enables description of a user interface at a high level of abstraction without requiring programming skills, enabling analysts, designers, programmers and end-user to use it during the development life cycle.

Levels of abstraction

UsiXML allows for user interfaces to be modelled at several levels of abstraction. The language is inspired by the Cameleon framework (**C**ontext **A**ware **M**odelling for **E**nabling and **L**everaging **E**ffective interacti**ON**) [13], which defines development stages for interactive applications with multiple contexts. In the current context the relevant layers are the Concrete User Interface (CUI) layer, comprising specifications of the user interface (in terms of interaction objects and their relationships) which are independent of the computing platform; and the Final User Interface (FUI) layer, the interface that can be executed or interpreted in a context of use (a specific computing platform and a set of specific devices, using specific interaction objects). A rendering engine performs a transformation from a CUI to a FUI. That is, a Reification transformation – converting an interface model at a more abstract level to a more concrete one.

Relevant models

The UsiXML language consists of a number of models that together address the needs of the framework described in the previous section. A detailed description of the different UsiXML models falls outside the scope of this paper. However, in order to contextualize the work, a brief presentation of the main models interpreted by FlexiXML will be made.

UiModel – User Interface model

The UIModel is the core model of the graphical interface specification. This component contains the common features to all models, such as version, author, or creation date, among others.

The UIModel aggregates a number of other models. In the specific case of the FlexiXML interpreter only the following are required: the concrete user interface model (cuiModel), the context model (contextModel) and the resources model (resourceModel). These are the models that contain information needed for constructing the final user interface.

CuiModel – Concrete User Interface model

The CuiModel specifies the concrete user interface as described previously. It defines the objects that make up the graphical interface (CIO - Concrete Interaction Objects),

and the relations between them (CUIR - Concrete User Interface Relationships). Particularly relevant are graphical transitions, which enable the specification of control flow.

Since FlexiXML has a fixed platform (the Air runtime), only the Environment and Stereotype features can vary. At this stage Stereotype was considered the relevant feature. In the stereotype, the feature that has more interest is the language. Using it, it is possible to define the language in which the generated application will be viewed. A UsiXML model can define more than one context, allowing user to view the application in different languages.

ResourceModel – Resources Model

The ResourceModel defines values for the attributes of the graphical objects that depend on the context (e.g. location, language, culture, etc.). This model contains all kinds of content that can be attributed to an interaction object (content, tooltip, etc).

ADOBE FLEX

As stated above the FlexiXML has been developed using the Adobe Flex[1] software development toolkit (hence the name *Flex*iXML).

Adobe Flex (or simply Flex) is an open-source framework for the development of cross platform Rich Internet Applications. The framework provides a library of components to build graphical user interfaces. These components can be extended to build new ones. User interface layout is declaratively defined using MXML, an XML-based user interface markup language. MXML also supports a predefined set of behaviors, such as transitions between elements. For more complex control logic, the object oriented Action Script 3 language is used. Action Script is a dialect of ECMAScript, and as such it shares its syntax and semantics with JavaScript.

Applications developed in Flex are compatible (i.e. can run) with all main browsers and operating systems. They can be run on a browser resorting to the Adobe Flash® Player plug-in, or directly on the desktop with the cross-platform Adobe AIR runtime environment.

Adobe AIR[2] is a cross platform runtime environment that enables Rich Internet Application to be run on the desktop to simulate native applications. Properly packages and signed applications will gain access to local resources in the host machine, bringing them closer to the flexibility and power of native applications. Runtime environments are available for most mainstream operating systems, including mobile operating systems such as Android and iOS. This enables an application developed in Flex to be run in a multitude of different platforms as either a Web or desktop application.

Besides the discussion above, other motivation to choose Adobe Flex as the implementation technology included:

- the fact that it enables access to a number of different data sources (different databases, XML files, etc.);
- the fact that it supports changing the user interface at runtime;
- the fact that it provides better performance when compared with previous versions of Flash.

USIXML vs. FLEX

One of the problems with tools such as FlashiXML is that the mapping between UsiXML and the implementation technology is hardcoded in the tool. This makes keeping the tool up-to-date with the language difficult. To avoid this pitfall we have opted for a configuration based approach when designing FlexiXML. Instead of hard coding the mapping in the tool, a configuration file is used to explicitly provide this mapping. This creates a decoupling between the tool implementation and the specific version of the language being used with the goal of easing maintenance and upgrades. It also should enable FlexiXML to support other markup language than UsiXML.

Three types of mapping were identified as being needed. A mapping between the user interface elements of the markup language and the widgets in the implementation technology; a mapping between the events in the markup language and the events supported by FlexiXML (in this case those supported by Action Script 3); and finally a mapping between window transitions in UsiXML and animation effects in Flex.

Regarding the first mapping, Figure 1 illustrates the mapping of four elements: windows, buttons, text components and checkboxes. In each case, a class in the Flex implementation is identified. In the first three cases, special purpose widgets, derived from the native widgets, are used. In the last case, a native widget is used.

```
<ComponentsMapper>
  <window component =
          "Classes.Components.FlexiXMLWindow"/>
  <button component =
          "Classes.Components.FlexiXMLButton"/>
  <textComponent component =
            "Classes.Components.FlexiXMLText"/>
  <checkBox component = "mx.controls.CheckBox"/>
<ComponentsMapper>
```

Figure 1 - Components mapping

Regarding the second mapping, Figure 2 illustrates how UsiXML events are mapped to events in the FUI. In this case four events are mapped: release, depress, rollOver and rollOut. These vents are mapped to corresponding mouse events: mouse up, mouse down, mouse over, and mouse out.

[1] http://www.adobe.com/products/flex/ (visited 14/07/2011)

[2] http://www.adobe.com/products/air/ (visited 14/07/2011)

```
<EventsMapper>
   <release event = "mouseUp"/>
   <depress event = "mouseDown"/>
   <rollOver event = "mouseOver"/>
   <rollOut event = "mouseOut"/>
</EventsMapper>
```

Figure 2 - Events mapping

Finally, regarding the third and last mapping, Figure 3 examples of mapping between window transitions in UsiXML, and animation effects in Flex. Hence, box in/out transitions are mapped to zoom effects, fade in/out transitions are mapped to corresponding fade effects, and close/open transitions are mapped to corresponding visibility effects.

Using these mappings, it becomes possible to easy tailor how the user interface is generated. For example, we could change how a depress or release event is detected at the interface, or specify that a fade event should be mapped to a zoom animation.

```
<ActionsMapper>
  <transition>
   <boxOut effect="Classes.Animation.Zoom"
           direction = "OUT"/>
   <boxIn effect="Classes.Animation.Zoom"
          direction = "IN"/>
   <fadeOut effect="Classes.Animation.Fade"
            direction = "OUT"/>
   <fadeIn effect="Classes.Animation.Fade"
           direction = "IN"/>
   <close effect="Classes.Animation.Visibility"
          direction= "OUT"/>
   <open effect="Classes.Animation.Visibility"
         direction = "IN"/>
  </transition>
</ActionsMapper>
```

Figure 3 - Actions mapping

FLEXIXML

Put simply, the main goal of FlexiXML is to produce Final User Interfaces from Concrete User Interfaces. Context and Resources models provide additional information that shapes the generation of the generated user interface. By definition Adobe's Air runtime environment is considered as belonging to the context of use. Currently the CUI model defines the user interface, the Context model defines the available user languages, and the Resources model defines language dependent attributes.

In the next sections the main features of the FlexiXML tool are introduced. An overview of the application is made, its architecture is described, and an example of use presented.

FlexiXML is structured around the concept of plugins, where each plugin implements a set of specific functions. This approach allows for constant evolution of the tool through the integration of new plugins and other features, with no impact on existing ones.

Figure 4 presents the architecture of the tool. This architecture can be divided into 3 layers:

- Application Manager – this layer performs application management. Its main responsibilities include loading and coordinating available plugins, and dealing with messages localization.
- Plugins – This is the layer where plugins are stored. In addition to the two default plugins (Project and Player), it is possible (through the Application Manager) to integrate additional plugins into this layer.
- CORE Data Manager – This layer is responsible for storing and providing information that is shared by all plugins, the most relevant being the CUI model to be rendered and animated.

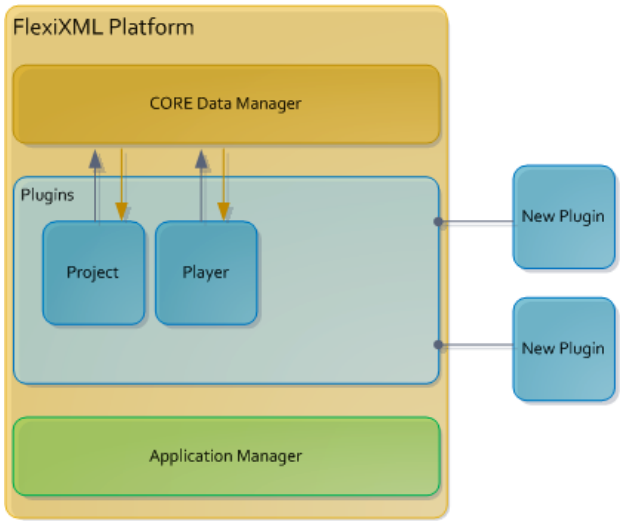

Figure 4 - FlexiXML platform

As stated, besides allowing FlexiXML to integrate new plugins, the current version provides two default plugins: Project and Player.

Project plugin

The Project plugin is responsible for loading the project. A project file identifies two further files: a UsiXML file with the CUI model describing the interface, and an ActionScript file with the dialogue control (i.e. the event handlers associated to controls in the CUI model). This arrangement promotes reuse since different CUI models can be used with the same event handlers and vice-versa.

The Project plugin loads the two files. A parser is then responsible for interpreting the file describing the interface and for filling the core data structures with this information so that others plugins might be able to use it. The parser to

be used is determined by the UIDL selected by the user in this plugin. The mapping between the parsers and the UIDLs is defined in a configuration file. Configuration files are explained later in this article. Currently, only the UsiXML parser is available, but other can be integrated. To do this the parser must implement the `parse()` method.

Player plugin

The Player plugin is responsible for generating the graphical interface of the loaded project. In addition to generating the interface, it allows changes to the generated interface at runtime, such as changing the style or language. This is the plugin where the user specifies the programming language to be used for user interface generation.

At the time of generation of the interface, the Player Plugin accesses the CORE Data Manager to get information about the current model. This information is then interpreted, and the relevant components and/or objects created that represent it. For each component defined in the model the corresponding graphical component, its behavior, its content and the possible transitions to other components are created. The mapping between each UsiXML element and the widgets/controls in the interface is defined in a configuration file (described later in this paper).

Presently FlexiXML includes a generator for ActionScript 3 only. However, other generators can be integrated. To do this the generator must implement the interface defined in Figure 5.

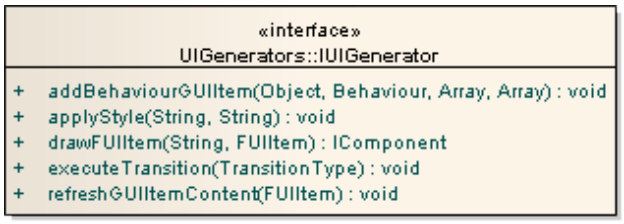

Figure 5 – Generator interface.

As showed in the figure, a generator must be able to:

- Add behavior to a widget (`addBehaviourGUI-Item`);
- Apply a style to the graphical user interface (`apply-Style`);
- Draw a graphic component (`drawFUIItem`);
- Play transitions between components (`execute-Transition`);
- Update the contents of a widget (`refreshGUI-ItemContent`).

New plugins

The list of plugins that FlexiXML provides is defined in an XML configuration file. The Application Manager reads this file during the initial process of starting the application. For each plugin, this file indicates its name, description, and the class implementing it. It is this class that the Application Manager will load to make the plugin available.

For the integration of a plugin into the platform to be possible, the plugin must be defined as a specialization of class `PluginBase` (see Figure 6).

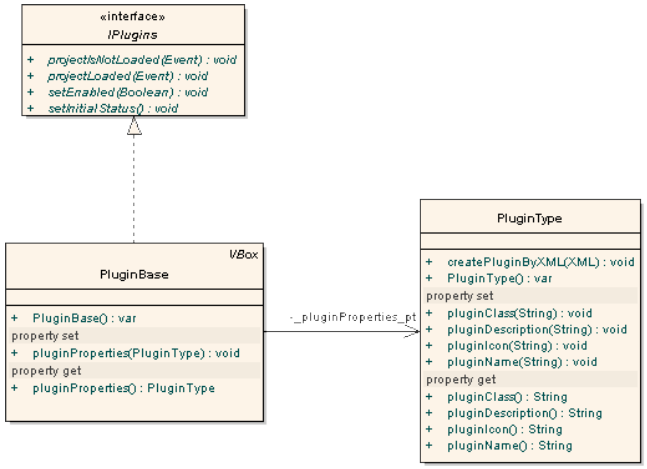

Figure 6 - Class diagram of a FlexiXML plugin.

The subclasses of `PluginBase` inherit and must set a `PluginType` object, which contains all the necessary properties for plugin characterization: name, description, icon, etc. In addition to setting these properties, the class must implement the `IPlugin` interface. This interface defines all required methods for FlexiXML to be able to interact with the plugin. The methods defined in this interface are:

- **`setInitialStatus()`** – defines the initial state of the plugin: active or inactive;
- **`setEnabled(enabled:Boolean)`** – assigns a specific state to the plugin: active or inactive;
- **`projectLoaded(event_evt:Event)`** – method executed whenever a new project is uploaded into the application;
- **`projectIsNotLoaded(event_evt:Event)`** – method executed whenever there is no longer a project in the application.

Configuration

As already mentioned, a set of configuration files allows changing the behavior and visual aspects of user interfaces generated by FlexiXML. The main configuration files are:

- Messages – All messages used in the application have an associated code. The message string associated with each code is defined in a XML configuration file. The existence of this file allows the FlexiXML to be localized without the need to recompile the code.

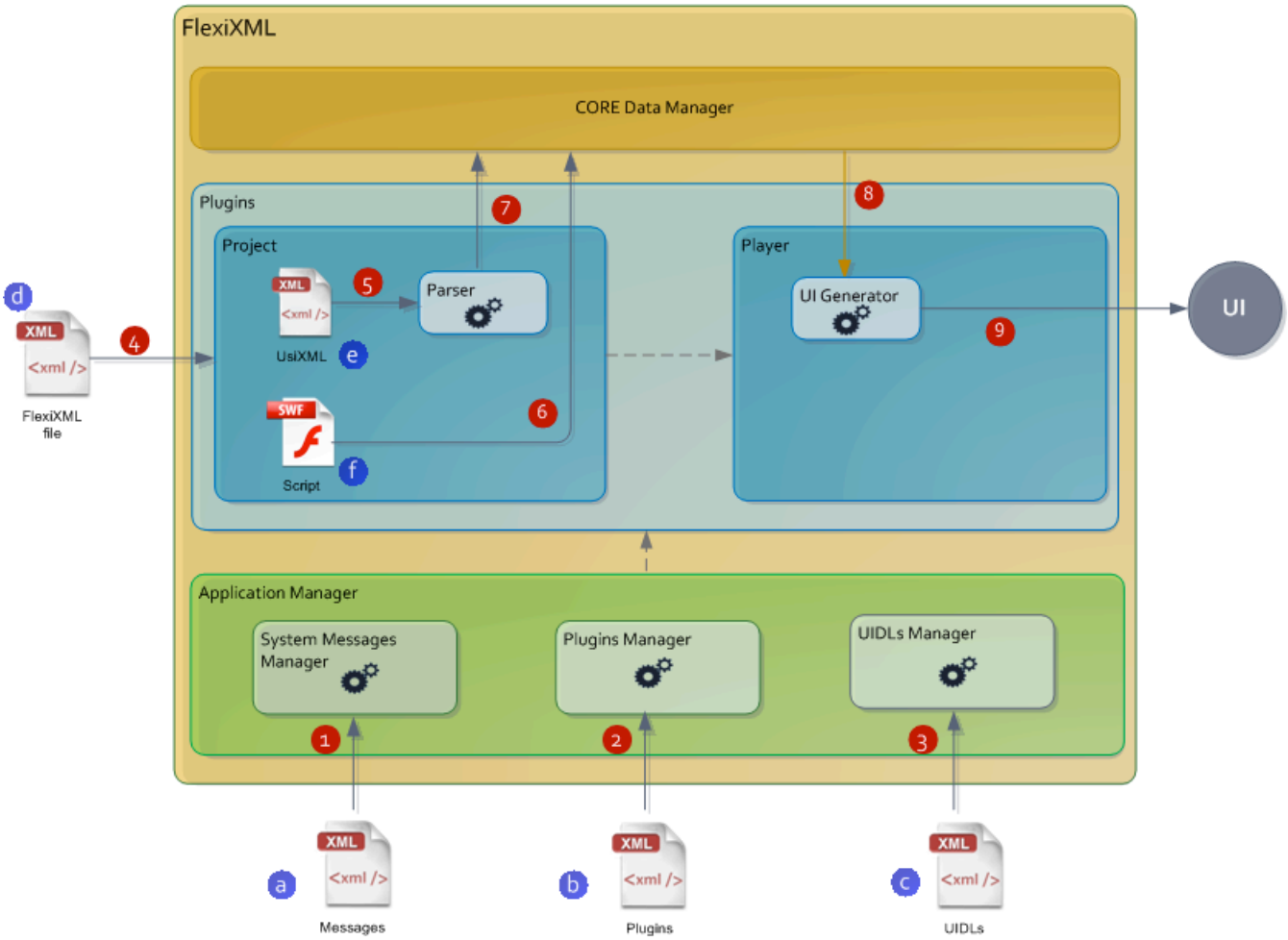

Figure 7 - FlexiXML workflow.

- Plugins – Defines the list of available plugins. If a new plugin needs to be inserted, it is necessary to include its information in this file for the application be able to load it.
- UIDLs – This file defines the list of UIDLs that FlexiXML interprets. In addition to listing the available UIDLs, it defines all the characteristics needed for their integration into the application. These include: the parser for the UIDL, the available programming languages for generating the interface, the mapping between the UIDL objects and the widgets that can represent it, among others.
- Styles – The list of styles available in the Player plugin is defined in this XML file. Thus, whenever there is the need to insert new styles, they simply have to be added to this file.

Workflow of the Generation Process

Previous sections, have described the basic building blocks of the FlexiXML's architecture. The process carried out by the tool for generating a graphical interface is now described. This process is depicted in Figure 7. The figure identifies both the inputs to the tool (labeled with letters *a* to *d*), and the flow of information (labeled with numbers *1* to *9*).

FlexiXML takes as input a set of configuration files (labels a, b and c in Figure 7) that are interpreted by dedicated managers (steps 1, 2 and 3 in Figure 7). These managers keep this information, which (once the tool is running) can then be accessed by any one of the plugins that has been loaded (see below). The managers are:

- System Messages Manager – this component is responsible for loading of the messages to be used in the application;
- Plugins Manager – this component is responsible for loading the listed plugins into the application;
- UIDLs Manager – this component is responsible for loading information of available UIDLs, and making this information available to plugins.

Once this initial processing has been done, the application becomes available, with all the plugins that have been configured.

Once the configuration of the tool is set up, the process for generating a graphical interface can start. For that, the tool

needs to load the two project files: one containing the UsiXML model (label e), and the other containing dialogue control written in ActionScript (label f). The process starts (step 4) by receiving as input a project file (label d) where the location of these two files is provided. The UsiXML model is interpreted by the Project plugin (step 5), which sends the information therein to the CORE Data Manager, together with the dialogue control information (steps 6 and 7). The Project plugin then creates the entities representing the components in the CUI model, and which the Player will afterwards interpret. The CORE Data Manager centralizes all the information that can be shared by the plugins. Thus, when a plugin needs information about the current project it must request it from this manager.

Once the project is loaded into the CORE Data Manager, the Player can generate a graphical user interface (step 9) based on the information provided by the CORE Data Manager (step 8).

AN ILLUSTRATIVE EXAMPLE

This section presents an example of a GUI generated using the FlexiXML tool. The example is an application to display and listen to music albums. Given the size of the model, only a few excerpts are presented here. The full model can be downloaded from the project's webpage[3].

Design

The user interface is generated inside a main box (ii) in the application's main window (i). In the concrete case of the "Music Player", the user interface consists of a window that can be divided into two main areas (see Figure 8): a header box (item *iii* in the figure) containing the application's controls; and an area (CurrentView) for displaying information about the albums collection (item *vi* in the figure).

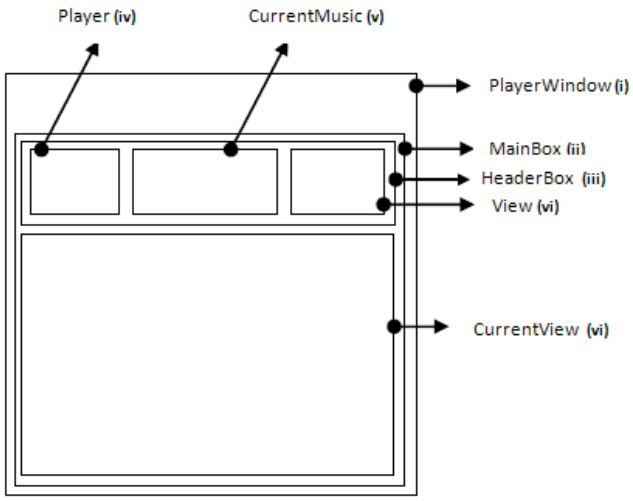

Figure 8 - Decomposition of the application in areas.

```
<cuiModel id="musicPlayer-cui"
          name=" musicPlayer-cuiModel">
  <window id="playerWindow" …> (i)
    <box id="mainBox" …> (ii)
      <box id="headerBox" …> (iii)
          <box id="playerBox" …/> (iv)
          <box id="currentMusicBox" …/> (v)
          <box id="viewsBox" …/> (vi)
      </box>
      <box id="currentView" …> (vii)
         […]
      </box>
    </box>
  </window>
</cuiModel>
```

Figure 9 – CUI model of the MusicPlayer application.

Figure 10 - Music player (coverView and gridView) generated by FlexiXML.

The header box is itself subdivided into three areas:

- Player – the area where the buttons to control the music are placed (item *iv* in the figure);
- CurrentMusic – the area that displays information about the music currently playing (item *v* in the figure);
- View – the area where the buttons to switch between the different display formats of the albums list are placed (item *vi* in the figure).

[3] http://FlexiXML.di.uminho.pt (visited 14/07/2011)

Figure 11 - Graphical transitions between views.

The View area contains three buttons for toggling between the three available display formats:

- ListView (to see the albums and theirs songs in list format);
- GridView (to see the albums in a grid);
- CoverView (to see the albums one at a time, by their cover).

Users may, at any time, change the view being used.

Figure 9 shows the basic structure of the model. The excerpt shown identifies the main structural components. For readability, the details of each component are omitted. The end result of the generation process is shown in Figure 10.

Behaviour

The above model describes the structure of the user interface. It is now necessary to model its behaviour. This will be illustrated with a concrete example.

As can be seen in Figure 10, it is possible to have different views of the album collection. Switching between views is achieved by pressing the corresponding buttons in the interface. Figure 11 illustrates the effect of pressing the "GridView" button: the "GridView" view should be displayed, and the previous view hidden.

The model specifying the behaviour of the `GridView` button is presented in Figure 12. When a depressed event happens in the button, a sequence of three transitions is fired: `GridViewTr1`, `GridViewTr2`, and `GridView-Tr3`. In addition, the method `updateGridView` must be invoked. This method is responsible for providing the necessary data to this view (e.g., list of albums to view).

Figure 12 defines the sequence of transitions, but does not describe what each transition actually is. That is done in Figure 13. There, it can be seen that `GridViewTr1` and `GridViewTr2` correspond to fade-outs of the grid and over views, respectively, while `GridViewTr3` corresponds to the fade-in of the list view.

```
<button id="gridButton”>
  <behavior id="gridView">

    <event id="gridViewEvt"
           eventType="depress"
           eventContext="gridButton"/>

    <action id="gridViewAct">
        <transition transitionIdRef
              ="GridViewTr1" />
        <transition transitionIdRef
              ="GridViewTr2" />
        <transition transitionIdRef
              ="GridViewTr3" />

        <methodCall methodName
              ="updateGridView"/>

    </action>

  </behavior>
</button>
```

Figure 12 - Specification of the behavior of the "GridView" button.

```
<graphicalTransition id="GridViewTr1"
           transitionType="fadeOut">
   <source id="gridButton" />
   <target id="listView" />
</graphicalTransition>

<graphicalTransition id="GridViewTr2"
           transitionType="fadeOut">
   <source id="gridButton" />
   <target id="coverView" />
</graphicalTransition>

<graphicalTransition id="GridViewTr3"
           transitionType="fadeIn">
   <source id="gridButton" />
   <target id="gridView" />
</graphicalTransition>
```

Figure 13 - Specification of the graphical transitions triggered by the "GridView" button.

Context (Language)

As already stated, FlexiXML supports localization of the interface via the definition of language contexts. To illustrate this use of context, the steps needed to make the interface available in two languages are now put forward.

Two models must be created to build the application in different languages: the ContextModel (to create the languages) and the ResourceModel (to specify the contents of the objects in each language). Figure 14 defines two languages for the "Music Player" application: English and French. Then Figure 15 shows the specification of the application title in the two languages. Figure 16 shows the application window with the titles in both languages.

FlexiXML also allows changing the style of the generated interface at runtime. The list of styles that can be applied is defined in a configuration file. In this file, the name of the

style, and the location of the style file (CSS or SWF format) are indicated.

Each style is defined in CSS (Cascading Style Sheet) format. This enables a style to contain images (skins), fonts, class selectors, among others. FlexiXML interprets this style from an external SWF (Shockwave Flash) file. The SWF files arise from the conversion of CSS files.

Figure 17 shows the "Music Player" in two different styles. In this case, the background colors of the header and song list where changed.

```
<contextModel id="playerContextModel"
              name="playerContextModel">
  <context id="playerContext_En_US"
           name="playerContext_En_US">
      <userStereotype
        id="playerContextUser_US"
        language="en_US"
        stereotypeName="playerContextUser_US"/>
  </context>
  <context id="playerContext_FR"
            name="playerContext_FR">
      <userStereotype
          id="playerContextUser_FR"
          language="FR"

stereotypeName="playerContextUser_FR"/>
  </context>
</contextModel>
```

Figure 14 - Specification of the languages.

```
<resourceModel id="playerResourceModel"
               name="playerResourceModel">
  <cioRef cioId="playerWindow">
     <resource content="Music Player"

contextId="playerContext_En_US"/>
     <resource content="Lecteur de Musique"
               contextId="playerContext_FR"/>
  </cioRef>
</resourceModel>
```

Figure 15 - Specification of the application title in different languages.

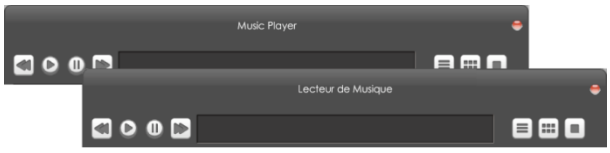

Figure 16 - "Music Player" in English and French Styles.

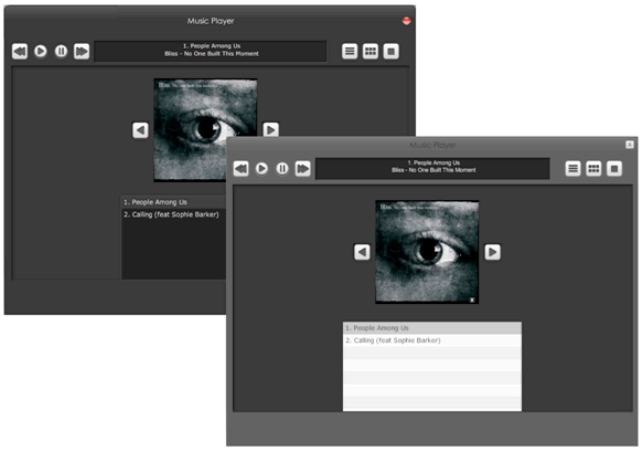

Figure 17 - "Music Player" with different styles.

CONCLUSIONS AND FUTURE WORK

The acceptance of an application depends largely on the quality of its graphical interface. Model-based user interface development helps ensure the quality of the solution can be assessed at an early stage [4], allowing for the problems identified to be analyzed as soon as possible. This is achieved through the creation of models at different levels of abstraction, from the domain model to the final user interface. To maximize model-based development, tools support is necessary both to enable analysis of the models, and to enable moving between levels of abstraction.

The aim of this project was the creation of a tool to support automatic generation of user interfaces from models expressed in the UsiXML language. The current version of the tool supports the UsiXML language, but is designed to allow the inclusion of other XML-based declarative languages.

In this version of the project, only a sub-set of the potential of the UsiXML language is used. For example, the possibility of defining characteristics of the computing platform in the context model was not considered. In alternative, a technology was used that enables the generated interfaces to be run in a variety of platforms. That is, the adaptation is not directly handled by FlexiXML, but by the runtime environment in which FlexiXML is executing.

The generated interfaces are created in FLEX and ActionScript 3. The tool, however, is structured so that the user can specify in which language (s)he wants the interface to be generated. These two characteristics are intended to make the tool as flexible as possible, not limiting users to particular languages (or language versions) for interface specification and generation, thus extending the number of users that can benefit from it.

The fact that the FlexiXML tool relies on the AIR runtime environment for execution makes it independent of any specific computing platform. Indeed, due to the use of the runtime environment, FlexiXML is available in two

formats: Desktop and Web. Where the desktop version can be used in any operating system that features an Air runtime environment (i.e. all major operating systems).

Looking back at the objectives initially set forth, the following features of the FlexiXML tool can be highlighted:

- Implementation in a recent technology (AIR, Flex and ActionScript3) enabling portability of the user interface (in the sense that it can be deployed in different platforms);
- An explicit, and configurable, mapping between the modeling language and the implementation technology (both regarding the structural elements of the interface, and regarding behaviour – supported events, graphical transitions);
- Support for runtime adaptation of the user interface via localization, and the use of styles to change the look of the interface;
- An architecture designed to support the integration of new plugins and new graphical interfaces modeling and/or programming languages.

At this stage, a number of future lines of work is open. Some of the possible improvements, and areas for future work include:

- Creating new plugins – for example, a model editor;
- Extending the widget library and the layout managers, in order to provide a wider set of user interface representations – thus supporting the creation of more realistic user interfaces;
- Supporting the generation of prototypes for different devices and platforms – while the tool can be run on a number of platform due to the AIR runtime, no attempt is made at this stage to adapt the generated interface to the device being used;
- Implementing parsers for new UIDLs, and generating the user interfaces using different programming languages and technologies.

ACKNOWLEDGMENTS

The authors would like to thank Jean Vanderdonckt and Michael D. Harrison for their helpful comments on previous versions of this paper. Sandrine Mendes would also like to thank her employer, Alert Life Sciences Computing, for sponsoring this work.

REFERENCES

1. Puerta, A., and Eisenstein, J. XIML: A common representation for interaction data. In Proc. of the 7th Intl. Conf. on Intelligent User Interfaces, ACM, 69-76.
2. Abrams, M., Phanouriou, C., Batongbacal, A.L., Williams, S., and Shuster, J. UIML: An Appliance-Independent XML User Interface Language. Proc. of 8th Inter. World-Wide Web Conference, Elsevier, 1999.
3. Argollo, M. Jr., and Olguin, C. Graphical user interface portability. *CrossTalk: The Journal of Defense Software Engineering*, 10(2):14–17, 1997.
4. Bäumer, D., Bischofberger, W.R., Lichter, H., and Züllighoven, H. User interface prototyping - concepts, tools, and experience. ICSE '96: Proceedings of the 18th international conference on Software Engineering, IEEE Computer Society, 532-541.
5. Coyette, A. *A Methodological Framework for Multi-Fidelity Sketching of User Interfaces*, Ph.D. thesis, Université Catholique de Louvain, Belgium, 2007.
6. Denis, V. *Un pas vers le poste de travail unique : QTKiXML, un interpréteur d'interface utilisateur à partir de sa description*, M.Sc. thesis, Université catholique de Louvain, Belgium, September 2005.
7. Kaklanis, N. *3D HapticWebBrowser*. http://kaklanis.googlepages.com/nickkaklanis-3dhapticwebbrowser. Last accesses on June 26, 2009.
8. Michotte, B., and Vanderdonckt, J. GrafiXML, A Multi-Target User Interface Builder based on UsiXML. Proc. of 4th Intl. Conference on Autonomic and Autonomous Systems ICAS'2008, IEEE Computer Society Press.
9. Mozilla foundation. *XUL Tutorial*, https://developer.mozilla.org/en/XUL_Tutorial. Last accessed on November 22, 2010.
10. Paternó. F and Santoro. C. One model, many interfaces. In Proceedings of the 4th Intl. Conf. on Computer-Aided Design of User Interfaces CADUI'2002. Kluwer Academics Publishers, 143-154.
11. Silva, E. *Sistemas interactivos*. Departamento de Computação, Universidade Federal de Ouro Preto. 2006.
12. Garcia, J.G., Gonzalez-Calleros, J.M., Vanderdonckt, J., Munoz-Arteaga, J. A Theoretical Survey of User Interface Description Languages: Preliminary Results, Proc. of Joint LA-Web/CLIHC'2009, pp. 52-59, 2009.
13. *The Cameleon Project – plasticity of user interfaces.* http://giove.cnuce.cnr.it/cameleon.html. Last accessed, June 26, 2009.
14. Université catholique de Louvain. *UsiXML – USer Interface eXtensible Markup Language*. http://www.usixml.org/. Last accessed on November 22, 2010.
15. Goffette, Y., Louvigny, H.-N. *Development of multimodal user interfaces by interpretation and by compiled components : a comparative analysis between InterpiXml and OpenInterface*, M.Sc. thesis, UCL, Louvain-la-Neuve, 28 August 2007
16. Vanderdonckt, J., Guerrero-Garcia, J., González-Calleros, J.M., A Model-Based Approach for Developing Vectorial User Interfaces, Proc. of Joint LA-Web/CLIHC'2009, pp. 52-59, 2009.