Constraint Adaptability of Multi-Device User Interfaces

Kris Luyten Jo Vermeulen Karin Coninx

Hasselt University – transnationale Universiteit Limburg
Expertise Centre for Digital Media – IBBT
Wetenschapspark, 2
B-3590 Diepenbeek (Belgium)

{kris.luyten,jo.vermeulen,karin.coninx}@uhasselt.be

ABSTRACT

Methods to support the creation of multi-device user interfaces typically use some type of abstraction of the user interface design. To retrieve the final user interface from the abstraction a transformation will be applied that specializes the abstraction for a particular target platform. The User Interface Markup Language (UIML) offers a way to create multi-device user interface descriptions while maintaining the consistency of certain aspects of a user interface across platforms. We extended the UIML language with support for layout constraints. Designers can create layout templates based on constraints that limit the ways a user interface can rearrange across platforms. This results in a higher degree of consistency and reusability of interface designs.

THE USER INTERFACE MARKUP LANGUAGE (UIML)

The UIML specification [2] is a high-level canonical markup language to describe the structure, style, content and behavior of a user interface. The declarative nature of UIML allows a clear separation of the user interface, its content, the mapping of its abstract concepts onto concrete widgets and the application logic. UIML's separation of concerns enables reuse of the user interface and promotes consistency across different platforms.

UIML does not contain metaphor-specific tags (e.g. `<window>`), but only generic tags (e.g. `<part>`, `<property>`, ...). A set of abstractions can be defined in a vocabulary and allows the designer to specify a user interface by referring to these abstractions. The vocabulary defines how abstractions can be translated into a concrete presentation. This is a typical example of an interface being specified in terms of abstract interaction objects which will be mapped onto concrete interaction objects afterward [7]. Nevertheless, it is challenging to choose the set of abstractions defined in a UIML vocabulary so a wide range of different platforms can be supported. Since these are specified separately, UIML is extendable to new devices and UI metaphors, when they become available. Several levels of abstraction can be supported by UIML.

An UIML document exists of several parts [1] that are shown in figure 1. Together they make up the Meta-Interface Model

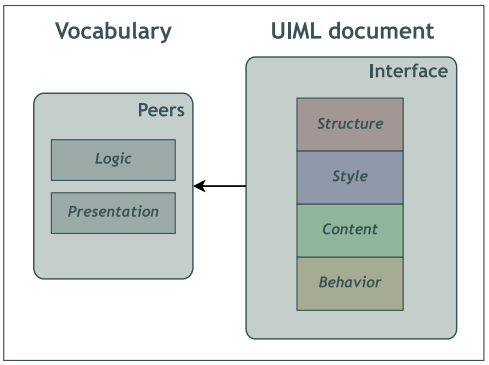


Figure 1. The UIML Meta-Interface Model

(MIM).

Interface describes four parts of the user interface: structure, style, content and behavior. The *structure* describes the "hierarchy" of the user interface. It defines the different parts that are contained in the user interface. *Style* describes properties of the parts defined in the structure. This allows to change properties of the interactors like text, color, font, etc. The *content* component separates the content of the interface (e.g. the list of items that has to appear in a list presentation) of the other parts. The *behavior* of a UIML document defines rules with actions that are triggered when some condition is met. Some kind of event mechanism is offered to the user interface designer this way.

Peers defines mappings to entities external to the UIML document, and is divided into the presentation and logic. *Presentation* contains the mapping with the concrete user interface toolkit. It defines a "vocabulary" to be used with a UIML document. Finally, the *logic* component defines how to bind the user interface with the application logic.

Listing 1 shows the structure and style components of an example UIML document. For more details about the UIML language, we refer to the specification [1].

Listing 1. Structure and style section for a simple dictionary user interface

```
<structure>
  <part class="Frame" id="OuterFrame">
    <part class="HBox" id="hl1">
      <part class="VBox" id="vl1">
        <part class="Label" id="TermLabel" />
        <part class="Combo" id="TermList" />
      </part>
    </part>
  </part>
</structure>
<style>
  <property part-name="OuterFrame" name="label">
    Simple Dictionary</property>
  <property part-name="TermLabel" name="text">
    Pick a term:</property>
</style>
```

We have developed a multi-device UIML renderer for the .Net platform: *Uiml.net* [6]. Our renderer can be deployed on desktop computers, tablet PCs, digital television and on mobile devices such as PDA's, Mobile Phones... It works on different implementations of the .Net framework (The Microsoft standard framework, the Microsoft Compact framework and Mono[1]), by consequence it can be considered platform independent. This renderer enables us to create one user interface design in a high-level XML-based language and reuse it on different platforms. The renderer takes advantage of the widget sets that are available on the platform to transform the UIML document into the final user interface.

UIML enables reuse of large parts of the user interface across different platforms. However, there is no abstraction for the layout of a user interface, resulting in UIML documents that can only be reused for a limited set of target devices. This is equivalent to what can be achieved with a generic vocabulary for the same family of devices, as was described by Ali et al. in [3]. Although the language also enforces consistency in various manners (a detailed discussion will follow in the next section), the absence of a layout abstraction causes inconsistencies in the layout of the user interface and in the part hierarchy.

SUPPORT FOR CONSISTENCY IN UIML

The structure section in a UIML document essentially defines the containment hierarchy. Although this should mainly stay the same for a wide range of devices, the same hierarchy can not be reused for more extreme conditions such as very small screens or multiple screens. In practice the structure specification also contains widget-set specific layout information. This results in multiple alternative structure specifications for the same user interface to keep them consistent for different widget-sets on different devices.

The separation of style and structure allows to decouple the user interface structure from its visual presentation. Different visual representations for the user interface elements can be provided which do not affect the user interface structure: the structure is kept consistent independent of changes in appearance.

The decoupling of the content from the other parts of the interface, makes it easy to update one without altering the others [3]. The content of a user interface has to be specified only once and can be referenced many times by different platform-specific versions of the same interface. This eliminates inconsistencies between these different versions.

The behavior element consists of a sequence of rules, each with a condition and a list of actions. A condition can hold when an event fires. Events are independent of the supported widget sets and use the same *class* and *name* mapping concepts as the mapping mechanism of parts. This ensures that we can use the same event class for the user interface on different platforms. This way, UIML guarantees that the interaction of each platform-specific version of the user interface behaves consistently. The event class is in turn mapped onto the specific type of event used by the target platform. For instance, we could define a *selection* event class, which gets mapped to an `OnClick` event for direct manipulation interfaces and to an `onfocus` event for a speech interface.

The logic component describes the functionality of the application while hiding the application itself and the communication between the user interface and the application with it. UIML abstracts the way the application logic can be addressed from within the user interface. UIML enables a consistent and platform-independent binding between the user interface and the application logic since this abstraction can be reused for other UIML-based user interfaces.

DEVICE-INDEPENDENT UIML (DI-UIML)

UIML achieves abstraction in many ways. The specification of user interface elements, interaction, content and the application logic are all platform-independent. However, UIML has no support for plastic layout management, which results in platform-specific layout adjustments for each of the target devices. Our solution supports a consistent layout in a wide range of circumstances, while still being flexible enough to adjust to extreme conditions.

For this purpose, Device-Independent UIML (DI-UIML) extends standard UIML with a high-level way to describe the graphical user interface layout. The designer is no longer bothered with widget-set dependent details. Our approach is based on the combination of spatial constraints and a constraint solving algorithm. The interface designer specifies the layout by defining constraints on the user interface components, such as *buttonA left-of labelB*. Afterwards, the constraint solver tries to find a solution that adheres to these constraints. Constraints are resolved on the level of the *abstract interaction objects*, so are *independent of the concrete representation* of the widgets.

Constraints allow us to specify the layout in a declarative manner and integrate smoothly with UIML. The designer can focus only on *what* is the desired layout, rather than *how* this layout is to be achieved. Furthermore, constraints allow

[1] `http://mono-project.com/`

partial specification of the layout, which can be combined with other partial specifications in a predictable way [5]. This is useful in our case to define the layout at several levels, taking advantage of an interface containment hierarchy (e.g. the parts in the structure section of listing 1 specify a containment hierarchy). For example, we can define that container *selection* is *left-of* container *content*. The *selection* and *content* containers can then each on its own specify the layout of their children. When a change in this layout requires the containers to grow, shrink or move, the upper-level layout constraints will be reevaluated. This allows us to define generic *layout patterns*. These define the layout of a number of containers, which can afterwards be filled in with a specific widget hierarchy using its own layout specification.

We developed Cassowary.net, a port of the Cassowary constraint solving toolkit [4] to the .NET platform. This solver is used by our UIML renderer to realize the specified layout and maintain a consistent layout of the user interface across devices.

USER INTERFACE CREATION WITH DI-UIML

Figure 2 shows the Rhythmbox[2] interface redone using UIML. The interface can now be used on different platforms and with different screen sizes (figure 2). Before, the designer could reuse most of the UIML document for different platforms, except the layout specification and some changes in the structure specification. Our solution solves this problem and makes it possible to reuse the interface design with different widget sets for different screen sizes without manual intervention. If other behavior is required, a designer can add or remove a constraint to obtain the envisioned effect, and is no longer bothered by any platform-specific problems while designing the interface.

Listing 2 shows how the designer (or design tool) can use the new `<layout>` tag in a UIML document. A layout pattern is a set of spatial constraints that can be applied to a part subtree. In listing 2 the GTK VBox layout pattern (cfr. listing 1) can be obtained by specifying that *Part1* is placed above *Part2* and that the parts are left-aligned. Constraints can be specified either directly in the UIML document, or included by referring to an external layout pattern by using a predefined alias. This can be compared with the usage of CSS for XHTML, but is not limited to a particular widget set nor dependent on a web browser.

Listing 2. Specifying the layout in a UIML document

```
<layout part-name="vl1">
  <d-param name="Part1" class="*" />
  <d-param name="Part2" class="Choice" />
  <constraint>
    <alias name="above">Part1,Part2</alias>
  </constraint>
  <constraint>
    <rule>Part1.left=Part2.left</rule>
  </constraint>
</layout>
```

[2] `http://www.gnome.org/projects/rhythmbox/`

The layout template presented in listing 2 is a parametrized layout template. A layout template can be applied to a part hierarchy by inserting it as a child of a part hierarchy. Notice the `class` attribute defines the required type of interactor. At the moment of writing we have full support for constraint-based layout management in Uiml.net, the parametrized layout templates are still work in progress however. This extension to the UIML language allows to achieve a greater level of consistency and reuse while reaching a higher level of abstraction in the user interface specification. Listing 3 shows how a template can be applied onto a part hierarchy. Notice the part class `Combo` will be matched by the `Choice` from the layout template in listing 2, since Combo is a possible instantiation for the Choice class according to the vocabulary that is used. More details about this mapping scheme can be found in the future work section.

Listing 3. Applying a layout template on listing 1

```
<part class="Frame" id="OuterFrame">
  <layout id="vl1">
    <part class="Label" id="TermLabel"/>
    <part class="Combo" id="TermList"/>
  </layout>
```

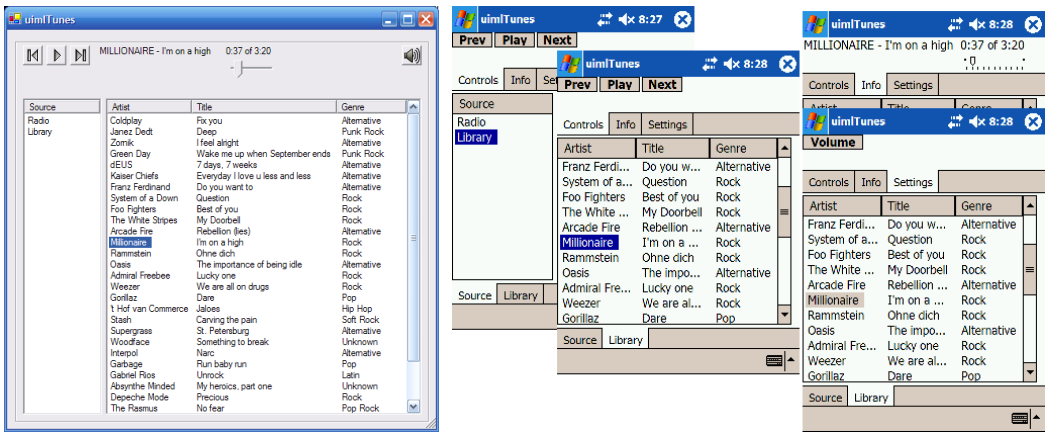
Previously, designers had to specify a platform-specific layout for every instantiation of the user interface. This process relied on the careful and precise work of the designer in order to keep the different layouts consistent. Furthermore, this process introduced a lot of work, because for every new target platform, the layout had to be almost completely redesigned. One could even wonder if using an abstract user interface specification like UIML was advantageous, since there was still a large part of the interface that had to be rewritten for every platform.

Our method enables designers to create new and reuse existing layout templates. Layout templates support consistent layouts for multi-device user interfaces. Under most conditions this layout can be kept consistent across platforms. When more extreme circumstances arise (e.g. the interface is deployed on a PDA, a cellphone or even distributed among several screens), the layout can adapt to the new environment by remapping abstract interactors to more basic concrete interactors and ignoring low-priority layout constraints.

FUTURE WORK

The current UIML vocabulary uses a one-to-one mapping of abstract interaction objects (AIOs) onto concrete interaction objects (CIOs). Unfortunately, this is not very flexible. We have been investigating a rule-based extension to the mapping mechanism, based on XSLT's *choice* element. This allows to select an appropriate concrete interactor for a given abstract interactor, according to a particular context of use. We call this method *1-to-N mapping*.

These context-sensitive selection rules allow a more significant adaption of the user interface to the target platform. A rule that states "If the screen space is too small for the preferred user interface, map the abstract *range* widget to a *spin*



(a) Desktop (b) PDA

Figure 2. UIML-based Rhythmbox interface rendered for different platforms.

box instead of a *slider*." would result in a user interface that utilizes the slider widget when there is enough screen space (e.g. on a desktop PC), and would otherwise (e.g. on a PDA) use the more compact spin box. Under normal conditions rich CIOs are used, while under more extreme conditions (in this example, a small amount of screen space) we fall back on basic CIOs. In contrast with other approaches, this approach requires the designer to specify her preferences and excludes unexpected adaptations. This is often required by designers and gives them the final decision over the final appearance of the user interface.

CONCLUSIONS

The User Interface Markup Language offers the user interface designer a means to create device independent interface designs. The separation of concerns (structure, style, behavior and content) allows to keep certain aspects of the user interface consistent across devices, while other aspects can differ according to the target device, toolkit or user. However, UIML lacks an abstraction for the layout of the user interface resulting in UIML documents that can only be reused for a limited set of target devices.

In this position paper we addressed how DI-UIML, an extension of UIML, supports the creation of consistent multi-device user interfaces. The use of spatial layout constraints leads to user interfaces that adapt according to the given screen space and preserve the same structure and style for a wide range of display resolutions. However, in more extreme circumstances, such as very small displays or multiple displays, the user interface can have an inconsistent representation while still being valid according to the constraints. Since the designer can add or remove constraints and define custom layout patterns with these constraints, the limits for consistency are exactly defined for the range of screen sizes for which the constraint solver can find a solution.

The work presented here is available as free software and can be found at `http://sf.net/projects/uimldotnet/`. The Uiml.net renderer for UIML and DI-UIML can be obtained here, as well as the Cassowary.net library that is used to solve the spatial layout constraints included in DI-UIML.

ACKNOWLEDGMENTS

Part of the research at EDM is funded by ERDF (European Fund for Regional Development), the Flemish Government and the Flemish Interdisciplinary institute for Broadband technology (IBBT).

REFERENCES

1. Marc Abrams and James Helms. User Interface Markup Language (UIML) Specification version 3.1. Technical report, Oasis UIML TC, 2004.

2. Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. *WWW8 / Computer Networks*, 1999.

3. Mir Farooq Ali, Manuel A. Pérez-Quiñones, Marc Abrams, and Eric Shell. Building Multi-Platform User Interfaces with UIML. In Christophe Kolski and Jean Vanderdonckt, editors, *CADUI 2002*, volume 3, pages 255–266. Kluwer Academic, 2002.

4. Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Computer-Human Interaction*, 8(4):267–306, 2001.

5. Greg J. Badros, Alan Borning, Kim Marriott, and Peter J. Stuckey. Constraint cascading style sheets for the web. In *UIST '99: 12th annual ACM symposium on User interface software and technology*, pages 73–82, 1999.

6. Kris Luyten and Karin Coninx. Uiml.net: an Open Uiml Renderer for the .Net Framework. In *Computer-Aided Design of User Interfaces*, 2004.

7. Jean Vanderdonckt and François Bodart. Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In *ACM Conference on Human Aspects in Computing Systems InterCHI'93*, 1993.