Jelly: A Multi-Device Design Environment for Managing Consistency Across Devices

Jan Meskens Kris Luyten Karin Coninx

Hasselt University – tUL – IBBT
Expertise Centre for Digital Media
Wetenschapspark 2, B-3590 Diepenbeek, Belgium
{jan.meskens,kris.luyten,karin.coninx}@uhasselt.be

ABSTRACT

When creating applications that should be available on multiple computing platforms, designers have to cope with different design tools and user interface toolkits. Incompatibilities between these design tools and toolkits make it hard to keep multi-device user interfaces consistent. This paper presents Jelly, a flexible design environment that can target a broad set of computing devices and toolkits. Jelly enables designers to copy parts of a user interface from one device to another and to maintain the different user interfaces in concert using linked editing. Our approach lowers the burden of designing multi-device user interfaces by eliminating the need to switch between different design tools and by providing tool support for keeping the user interfaces consistent across different platforms and toolkits.

Categories and Subject Descriptors

H.5.2 [**Information interfaces and presentation**]: User Interfaces – Graphical user interfaces, Prototyping

General Terms

Design

Keywords

Design tools, multi-platform GUI design, GUI builder

1. INTRODUCTION

The diversity of consumer computing platforms has become a common fact these days. Porting applications across different platforms implies the creation of a suitable User Interface (UI) for each targeted computing device. Using commercial design tools, this is a time consuming and cumbersome task since designers have to build every UI from scratch for each computing device or toolkit. Instead of designing every UI manually, researchers have proposed transformation approaches that automatically generate a suitable UI for every platform [21, 8, 16]. However, besides a few specific application domains [19, 8], these approaches are mostly hard to control and often generate undesired results [17].

AVI '10, May 25-29, 2010, Rome, Italy

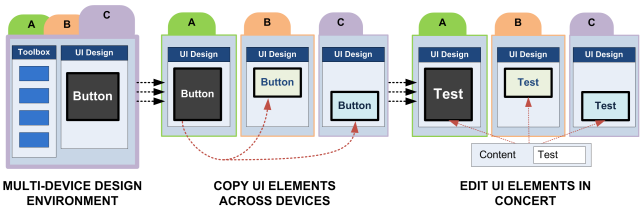


Figure 1: The Jelly multi-device design environment enables designers to copy UI elements across devices and to maintain the content of these copies in concert.

Due to the problems with automatic UI generation, designers refuge to manually designing UIs and need to master different design tools for different target platforms. Switching between design environments can be disorienting (e.g. if consistency among platforms is important), but also the different styles for designing interfaces and the incompatibilities between design environments causes problems [9]. The lack of interoperability between design tools makes it hard to reuse UI elements across devices and to keep the *content* of multi-device UIs consistent [13, 14]. UI content can be for example the text of a button or the items in a container widget.

To overcome the aforementioned issues when designing cross-device UIs manually, it would be beneficial if designers could create UIs for different computing platforms and toolkits inside one design environment which has the ability to share and edit parts of UIs across devices. We introduce *Jelly* (see Figure 2), a UI design environment manifesting this idea.

The three contributions to multi-device UI design research presented in this paper are (see Figure 1):

- The *Jelly design tool* to design UIs for multiple computing platforms (see Figure 1, left). The cornerstone of this design tool is a flexible architecture that can target a broad set of computing platforms and toolkits including mobile phone toolkits such as Android or Windows Mobile, web toolkits such as Adobe Flex, and PC-based toolkits such as Java Swing and the Windows Presentation Foundation (WPF). Jelly is tightly

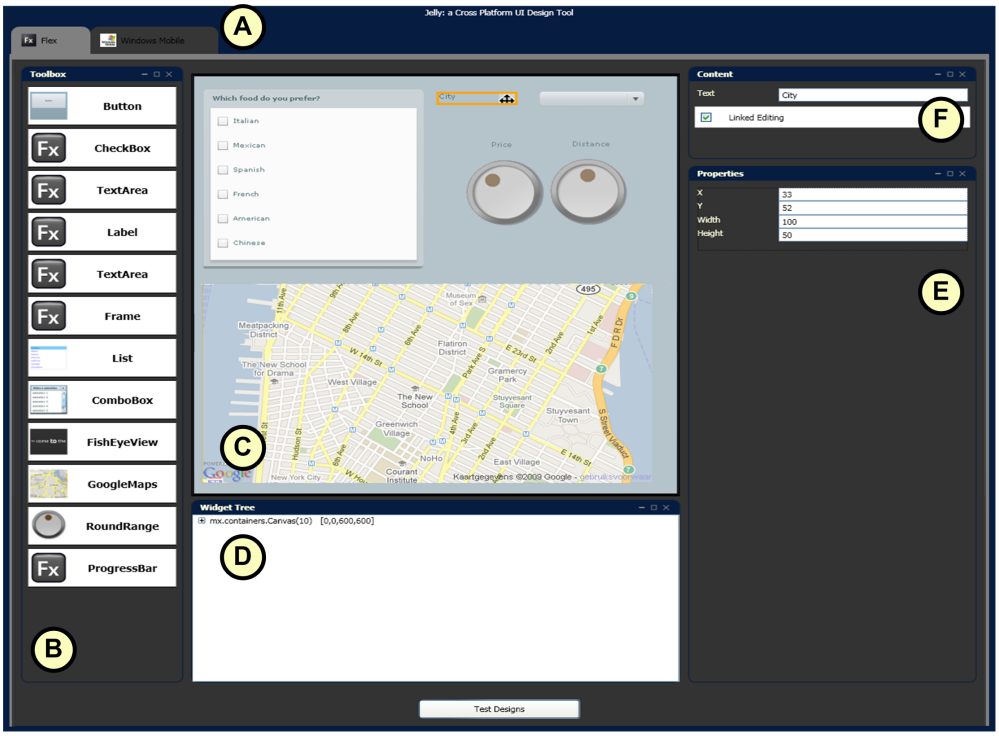


Figure 2: Jelly is a multi-device graphical UI design environment that allows designers to copy UI elements across different UI toolkits and platforms.

coupled with device emulators, which facilitates UI *testing* without deployment time.

- A *cross-device copy* extension to Jelly, allowing designers to copy parts of a UI across devices (see Figure 1, middle). For copying UI elements between device specific toolkits, Jelly employs an abstract UI description under the hood. This higher level of abstraction enables Jelly to compute how the copied element can be represented on a different device that has a different toolkit.

- *Linked editing*, a technique that allows designers to edit the content of UI elements across devices in concert (see Figure 1, right). This technique originates from the CodeLink [23] programming editor where it enables programmers to modify duplicated source code. In Jelly, linked editing is employed to keep the content of duplicated UI elements consistent across different UIs on multiple devices.

2. FIELDWORK

To learn about the existing needs for supporting multi-device UI design, we met with 9 UI design professionals across four different design companies. During these meetings, we stimulated discussion about future multi-device tool support by demonstrating the existing Gummy [15] multi-device GUI builder. This GUI builder employs a sequential design approach, where designers first design a UI for one single platform. When targeting another platform, Gummy transforms this design into an initial design for the new platform. Designers can then refine this design to reach the desired result.

All of the UI design professionals rejected a tool that automatically transforms UIs from one platform to another. They argued that mobile device UIs often support different requirements than desktop UIs. When automatically transforming a complete UI from one device to another, it would be difficult to take these differences into account. Instead of transforming complete UIs, several designers noted that transforming *parts of a UI* might be a better approach. This requires a tool that supports a *parallel design approach*, where designers edit UI designs for multiple devices in parallel.

The designers stressed the importance of *testing* during the UI design process. Testing helps to discover how a UI can anticipate on target device constraints such as display size, screen resolution and interaction mechanisms. A barrier when testing UIs is the time it usually takes to deploy a design to a device or device emulator. One designer mentioned: *"When building ITV applications, I connect my PC directly to the TV screen. This increases development time because the application can be compiled and tested directly on the PC instead of using the slow deployment process on the ITV emulator"*. This testing barrier is similar to the information barrier in end user programming as identified by Ko et al. [12], and the need for testing tool support in UI design as identified by Grigoreanu et al. [9].

Several designers noted the limitation of Gummy to target only form-based UIs. Especially web designers would like to work with *customised interface controls* since this allows them to emphasise certain parts of a website. A common desire of these designers is to *reuse* components that were created with existing design tools inside a multi-device design environment. This desire is consistent with a large scale

survey of 259 designers conducted by Myers et al. [18], which showed that reusing UI components across design projects is common practice these days.

Most of the designers we interviewed were regularly facing problems to manage *consistency* across multi-device UIs. One designer mentioned that *"it would be great to have a* more powerful CSS *that can adjust a website from one device to another while preserving its content"*. Lin et al. also found that designers experience manual consistency management as a major burden [14].

3. JELLY

Building on what we learned during the fieldwork, we created Jelly: a tool for designing multi-device GUIs. This section gives a detailed description of Jelly's user interface and its usage model.

3.1 Multiple Design Workspaces

Jelly's user interface is subdivided in several individual design workspaces. When starting up Jelly, the target platforms have to be selected and for each of these platforms a design workspace will be loaded. Workspaces can be accessed through a tabpage at the top of the design environment. For example, Figure 2 shows Jelly for designing Adobe Flex webbased UIs and UIs for handheld devices on top of the Windows Mobile .NET UI toolkit. At any time in the design process, designers can add additional target platforms.

A design workspace in Jelly is not that different from most traditional GUI builders. It contains a *toolbox* showing the available user interface elements (see Figure 2-B), a *canvas* to build the actual user interface (see Figure 2-C), a *tree view* showing the UI design structure (see Figure 2-D) and a *properties panel* to change the style properties of the user interface elements on the canvas (see Figure 2-E). Jelly also contains a content panel to modify the contents of a selected UI element (see Figure 2-F).

3.2 Parallel UI Design

Jelly supports a *parallel* design approach. Throughout the design process, designers can always switch between workspaces. Every UI design workspace is structured in the same way, which does not force designers to learn new tools and different usage models when targeting a new platform. Designers can also transfer UI elements from one workspace to another, which is a unique feature since design tools for different toolkits are often incompatible.

Using Jelly's parallel design approach, a designer can for example start by selecting the Windows Mobile workspace (see Figure 3-step 1) and drag a combobox from the toolbox to the canvas. The combobox is now visualised on Jelly's canvas and is surrounded by a *manipulator* (see Figure 3-step 2). This manipulator enables designers to manipulate a component's size and position using direct manipulation. The designer can now copy this element by right clicking on it and selecting the copy option in the context menu (Figure 3-step 3). Next, she can switch to the Adobe Flex workspace through the corresponding tab (Figure 3-step 4). In this workspace, the designer can now right click on the canvas and select the *"paste as"* option (see Figure 3-step 5). Here she can select how the element on the clipboard should appear in a different toolkit on a different device. By selecting for example the "fish-eye view" option in this list, the copied combobox will be added as a fish-eye view to the Adobe Flex design. This fish-eye view is a custom horizontal listbox that enlarges the items below the mouse cursor.

3.3 Managing Consistency with Linked Editing

When an element is copied from one platform to another, Jelly considers these elements as being *linked* to each other. Jelly enables designers to edit the content of these elements in concert using the *linked editing* mechanism. This way, designers can ensure the consistency of UI content across different devices and toolkits.

For example, when copying a combobox element from the Windows Mobile design environment as a fish-eye view selection element in Adobe Flex, Jelly will recognise and remember these are linked. When the fish-eye view is selected, its properties can be modified in the content panel just as would be the case in a regular environment (see Figure 3-step 6). When the *linked editing* option is on (presented by a checkbox in the design environment), these properties will be changed for the Adobe Flex fish-eye view as well as for the Windows Mobile combobox. Afterwards, both components will contain the same contents.

3.4 Instant UI Testing

Jelly is closely integrated with device emulators (see section 4.1) and allows designers to test their designs continuously with a minimal deployment time. Clicking on the *"test designs"* button (see Figure 3-step 7) will immediately show a running version of the UI designs . Testing mostly happens on the same PC as the design environment, for instance inside a device emulator. For example, the fish-eye view can be quickly tested inside a web browser while the Windows Mobile design is running in a device emulator (see Figure 3-step 8).

Emulators are important tools, since they provide the designer with a relatively precise approximation of the targeted device context. When building UIs for mobile devices (e.g. smartphones running Android or Windows Mobile), the emulators are used during the design activities and often for early evaluations. They are clearly indispensable tools to get the design right for a particular platform. This is not different from how designers create website designs: these designs are first rendered "offline" in different web browsers for evaluation and testing purposes.

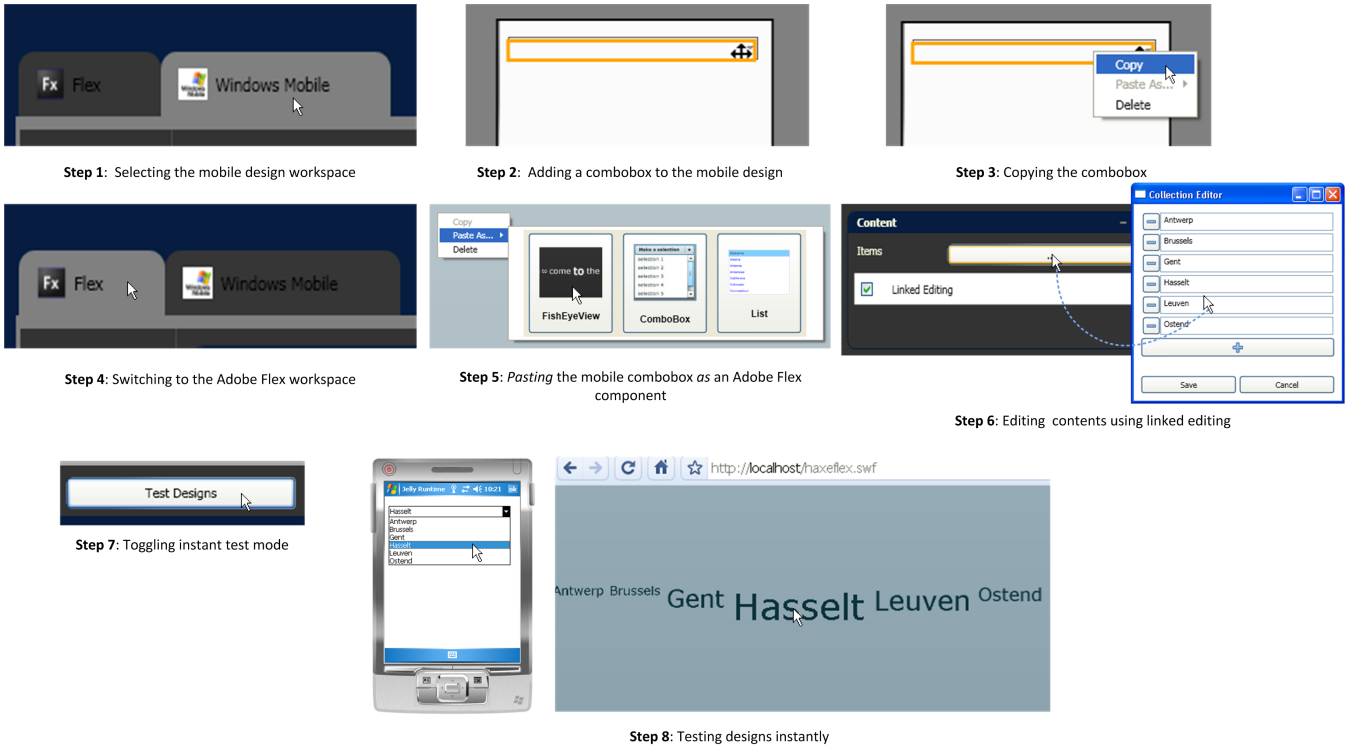
In the end, designers can use Jelly to deploy their designs to the actual target device. This way, designers can check if a design works well with device specific constraints such as touchscreen sensitivity, screen resolutions or processing power. Standard components like buttons or comboboxes have a very predictable behaviour, but complex custom components like the fish-eye view might be more difficult to handle on certain platforms.

4. ARCHITECTURE

This section describes the details for realizing a multi-device UI design environment that can target a broad set of computing platforms.

4.1 See-Through Interface

The Jelly design environment is inspired by the *see-through interfaces* [3] described by Bier et al. The canvas in Jelly's design workspace is a semi-transparant panel that is placed



Step 1: Selecting the mobile design workspace

Step 2: Adding a combobox to the mobile design

Step 3: Copying the combobox

Step 4: Switching to the Adobe Flex workspace

Step 5: *Pasting* the mobile combobox *as* an Adobe Flex component

Step 6: Editing contents using linked editing

Step 7: Toggling instant test mode

Step 8: Testing designs instantly

Figure 3: Jelly supports a parallel UI design approach across devices in combination with "linked editing" to maintain the consistency of multi-device UI content

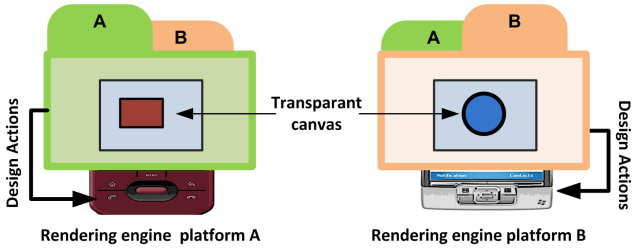


Figure 4: Jelly's see-through interface approach.

on top of a running version of the currently designed GUI that is hosted by a rendering engine (see Figure 4). When a designer makes changes to a UI design in Jelly, the design modifications are communicated to this rendering engine. The latter will then update the running GUI accordingly, which can be perceived through the semi-transparant canvas.

The see-through interface approach gives Jelly the required flexibility to target a broad set of computing devices. The rendering engine is loosely coupled to the design tool and can be implemented in any programming language, it just has to be placed behind the design tool's canvas running in a device emulator, in a webbrowser, or directly on the same PC as Jelly. Currently, we have implemented rendering engines in three different programming languages for different platforms (see Figure 5): an Action Scripting (AS3) rendering engine for producing web based Adobe Flex UIs, a .NET rendering engine for targeting desktop and Windows Mobile UIs and a Java rendering engine for designing Google Android mobile interfaces.

In Jelly, the UI deployment time is reduced dramatically. During UI design, Jelly automatically aligns the rendering engine with the design canvas using Windows API calls. When going to testing mode, Jelly brings the running GUIs to the front. This reduces the deployment time for testing a GUI to a few milliseconds (i.e. the time needed to bring a window in front). A designer can easily go back to design mode by clicking on the *design* button.

4.2 Implementation

Jelly's design-tool and rendering engine are designed as a *three-tier architecture* (see Figure 6) having a user interface, data and network layer. Before discussing the details of each layer, we describe the pipeline that is used to exchange information between the design tool and the rendering engine.

Communication pipeline. Each time a designer alters the UI design in the design tool (see Figure 6 - 1), the internal representation of this UI is adapted in the *data layer* (see Figure 6 - 2), and sent to the rendering engine over the *network layer* (see Figure 6 - 3). When receiving this message, the rendering engine updates the internal UI representation in its data layer (see Figure 6 - 4). Next, the running GUI will be updated accordingly to make this change visible (see Figure 6 - 5). Updating a running GUI implicitly reevaluates the GUI's layout managers which may change layout properties of several UI elements. These changes are stored in the rendering engine's *data layer* (see Figure 4 - 6) and sent back to the design tool through the network layer. The design tool will then update its internal data representation (see Figure 4 - 7) and finally its design workspace.

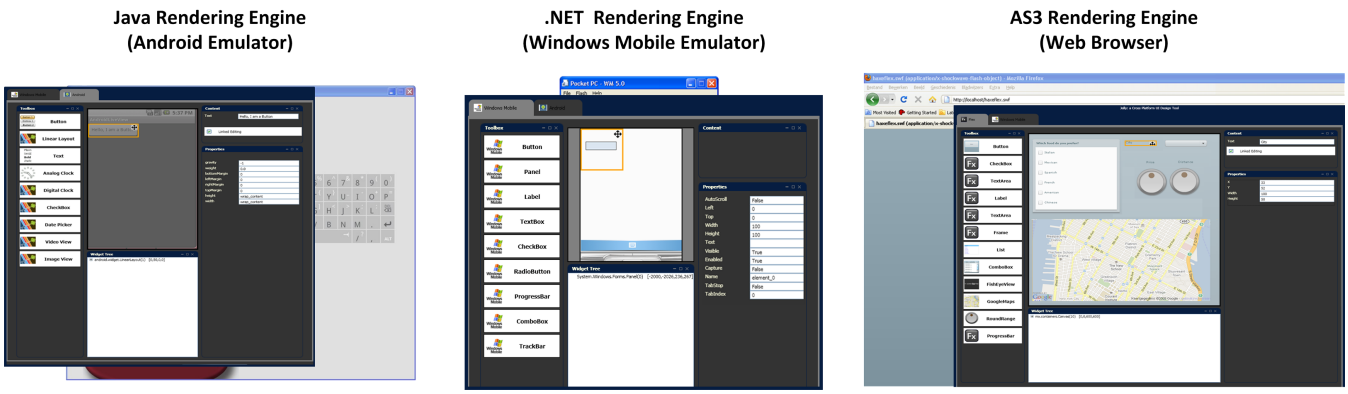


Figure 5: Jelly's canvas is a semi-transparant window that is placed on top of a UI rendering engine.

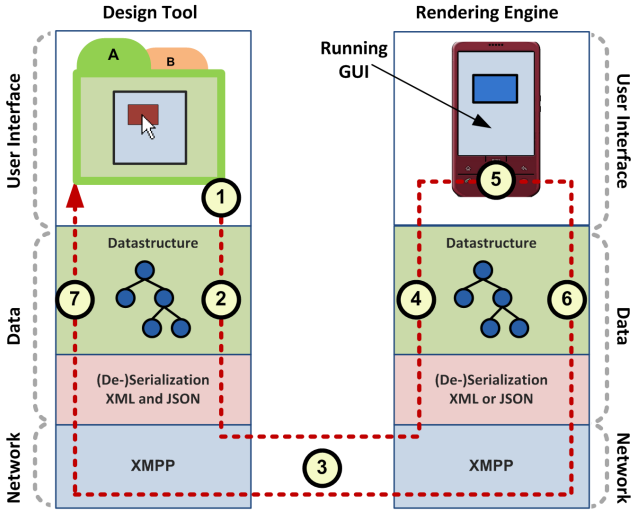


Figure 6: Detailed view on the communication between the design tool and the runtime GUI, hosted by a rendering engine.

The **user interface layer** contains the design tool's UI implemented in .NET using the Windows Presentation Foundation (WPF) toolkit. On the rendering engine side, the user interface layer consists of the running version of the currently designed GUI. This running GUI is located behind the transparant canvas of the design tool.

The **data layer** employs a tree datastructure to describe the currently designed UI. The design tool uses the UI datastructure to build up its treeview, properties- and content panel. In the rendering engine, this datastructure is used to update and visualise the running GUI. This tree is inspired by the User Interface Markup Language [11] and describes a UI in a toolkit- and device-independent way. Each node in the tree corresponds to a UI element and describes the element's name, toolkit mapping, content, properties and bounding box. The properties and the element's content are described by their names, toolkit mappings and values. Mentioning the toolkit mappings explicitly enables Jelly to extend the number of supported components dynamically with new custom components. Figure 7 shows a snippet of Jelly's internal tree datastructure for an Adobe Flex UI.

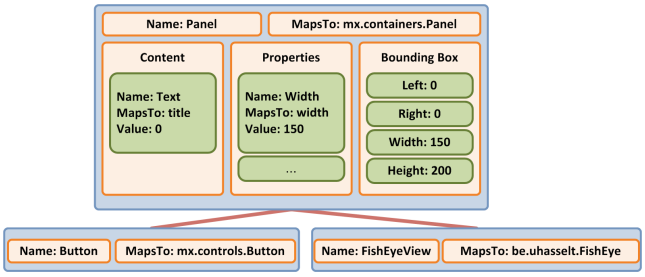


Figure 7: Jelly's internal representation of an Adobe Flex UI.

Before sending a piece of this tree through the network layer, it is serialised in the Javascript Object Notation (JSON) or XML. The former facilitates the development of web-based rendering engines (e.g. the Adobe Flex rendering engine), while the latter is mostly employed by native platform rendering engines such as for the Android or .NET toolkits.

An important point in our architecture is the **network layer** that enables the rendering engine and design tool to exchange information. This requires a communication protocol that is available in multiple programming languages and on multiple platforms. We chose for the IETF standard Extensible Messaging and Presence Protocol (XMPP). Pierce et al. [22] have shown that this instant messaging protocol provides the required functionalities and flexibility to build cross-platform distributed applications. Several open-source XMPP libraries exist in different programming languages, which simplifies the development of new rendering engines. In our rendering engines, we used the XIFF [1] library for AS3, AGSXMPP [2] in .NET and Smack [3] in Java.

5. REMAPPING UI ELEMENTS

When copying a UI element from one device to another, Jelly needs to *remap* the copied UI element to a suitable element in the other device's toolkit. First, Jelly computes all elements in the new device toolkit that are *similar* to the original UI element and allows the designer to select one of these elements through the *"paste as..."* menu. Jelly

[1] `http://www.igniterealtime.org/projects/xiff/`
[2] `http://www.ag-software.de/agsxmpp-sdk.html`
[3] `http://www.igniterealtime.org/projects/smack/`

adds an instance of the selected element to the current design workspace and gives it the same content as the copied element.

In order to compute elements similar to the copied element, Jelly adopts a similar approach as the user interface semantic networks described by Demeure et al. [6] and Vermeulen et al. [25]. These UI semantic networks describe the relationship between concrete UI components and Abstract Interaction Objects (AIOs). AIOs describe UI components independently of any platform and interaction modality (e.g. graphical, vocal, tactile, ...) [24]. When copying an element from one device to another, the corresponding elements on the other device can be computed by asking the semantic network for components that are linked to the same AIO as the copied element on the first device.

Jelly loads its UI semantic network from a custom XML format. Storing the network in an external XML file allows adding new (custom) controls to the network at any time. Our network employs the same set of high-level AIO types as described by Vermeulen et al. [25]. These AIOs are differentiated according to the functionality they offer to the user: (1) *input* components allow users to enter or manipulate content; (2) *output* components present content to the user; (3) *action* components allow users to trigger an action; and finally (4) *group* components group other components into a hierarchical structure.

When querying our UI network to find components similar to a given component, we look for all components that have the same AIO type (i.e. input,output,action or group) and have the same *content datatype* as the given component. Currently, we support the five primitive types of XML Schema (e.g. decimal, string, void, etc.), a number of datatypes that are often used in user interfaces (e.g. Image, Colour, etc.) and container datatypes that group content items of a certain type together (e.g. a list of strings, a tree of images, etc.).

For example, assume we are querying our network for all Adobe Flex components that can represent a Windows Mobile combobox. Since the combobox is linked to an input AIO, the network is first searched for all Adobe Flex components that are linked to an input AIO. This returns a huge list of controls such as checkbox, spinbox, listbox, combobox, fish-eye view, etc. Secondly, this list is searched for all components that support the same datatype as the Windows Mobile combobox (i.e. a list of strings). This finally results in three Adobe Flex components: listbox, combobox and fish-eye view. These components are then displayed in Jelly's *"paste as..."* menu (see Figure 3, step 4).

6. EVALUATION

We conducted a first informal use study of Jelly in order to gain feedback about the usefulness of the tool and the usability of the basic interactions, such as instant UI testing, copying and pasting elements across devices and linked editing. Eight lab members participated in the study, all having a reasonable level of experience with commercial user interface design tools. Five of them had experience with UI design for other devices than desktop PCs such as multi-touch tables or mobile devices.

One evaluation session was conducted per participant, and each evaluation session consisted of three parts. First, the participant was asked to read a written tutorial about the Jelly design tool and its basic functionalities. The second part was to get the participant used to interacting with Jelly. We asked the user to add a component to one design workspace, to copy this element to another workspace and to adapt the contents of this element using linked editing.

The final task was a design task, where we asked the participants to create a music player GUI for a Windows Mobile device and an Adobe Flex website. We chose a music-player since this is a well-known application that exists on many devices with a reasonable variety in the different device specific UIs. The designers were provided with two mockup sketches that gave them an idea of what we expected from the music-player GUIs. The desired music player features were for example selecting a song in a playlist, controlling the volume of the music and navigating through a song.

During the experiment, a think aloud protocol was used to understand the actions and decisions of the participants. Additional questions were asked by the observers if necessary. After the participants finished their task, they filled out a questionnaire asking about Jelly's core features such as testing UI elements, copying UI elements and linked editing in terms of usefulness and ease-of-use. The participants rated their usefulness and ease-of-use responses on 5-point Likert scales ranging from "not very useful" to "very useful", and from "very difficult to use" to "very easy to use".

6.1 Results and Feedback

Figures 8 summarises the results of the post-test questionnaire. From all features we tested in Jelly, the participants ranked *copying elements* across devices as Jelly's most useful feature (mean=4.38, median=4.50, $\sigma = 0.74$). On the other hand, the participants rated this feature as the most difficult to use (mean=3.38, median=3.00, $\sigma = 0.92$). This seems to indicate that users felt that this technique's conceptual idea was on target, but that its current implementation can be improved.

How useful was...	mean	median	σ
Testing UI Designs	4.13	4.00	0.64
Linked Editing	4.13	4.00	0.83
Copying Elements	4.38	4.50	0.74

How difficult was...	mean	median	σ
Testing UI Designs	4.25	4.00	0.71
Linked Editing	4.25	4.00	0.46
Copying Elements	3.38	3.00	0.92

Figure 8: Post-experiment questionnaire results.

Two participants suggested to distinguish between *"trivial copy operations"* that copy a UI element to a very similar element (e.g. copying a Windows Mobile button to an Adobe Flex button) and *"complex copy operations"* where an element is pasted as a very different widget (e.g. migrating a Combobox as a fish-eye view). They would like to do the straightforward copy operations for multiple items at the same time, while doing complex copy operations one-by-one. Another participant wanted to have more visual feedback about the component that is currently stored on the clipboard: *"it would be great to have a 'paste X as Y or Z' option, where 'X' is the component on the clipboard and 'Y' or 'Z' are candidate widgets to represent 'X' "*. The current implementation of Jelly only migrates a component's con-

tent when it is copied from one device to another. Several users wanted to have the option to copy other properties as well. One designer mentioned that *"copying would be more useful if the spatial layout would remain more or less intact"*.

The participants also gave high marks to the usefulness of Jelly's *instant testing* feature (mean=4.13, median=4.00, $\sigma = 0.64$) and gave this feature a fairly high usability rating (mean=4.25, median=4.00, $\sigma = 0.71$). In the Adobe Flex design, we noticed that almost each participant used a custom round range slider as a volume control for the music player. Six participants switched to the testing mode after adding this custom control and tested its behaviour in a web browser. Most of the participants liked the instant switch between test and design mode.

Jelly's *linked editing* was also rated as useful (mean=4.13, median=4.00, $\sigma = 0.83$) and easy to use (mean=4.25, median=4.00, $\sigma = 0.46$). During the post-test survey, several participants noticed that this technique would be great during software maintenance or updates of larger multi-device projects. They also mentioned that inconsistency between UIs mostly appears during software maintenance, where updates are done in one version of the UI but not in the other and vice versa.

7. RELATED WORK

Jelly draws on related work in two areas of research: automatic user interface generation and multi-device UI design tools. We discuss each area in turn.

7.1 Automatic User Interface Generation

Automatic user interface generation originates from *model-based* systems such as XWeb [21], PUC [19], Teresa [16] and Supple [7]. Rather than specifying a visual UI design from scratch for each computing platform, these systems allow designers to specify a UI only once by means of an abstract model. The platform-specific UIs are then generated automatically from this abstract description. Even though model-based UI design can simplify the development of UIs for multiple devices in a few specific application domains [19, 8], this technique can be daunting for designers. Designers have to master a new language to specify the high-level models and cannot control the look and feel of the resulting UI [17]. In Jelly, this problem is solved since designers can directly alter the visual design instead of abstract UI descriptions.

7.2 Multi-Device User Interface Design Tools

For creating multi-device UIs, designers can rely on design tools for cross-platform UI toolkits such as AWT [1] or Qt [2]. UIs created with these toolkits can run on every platform that is supported by the underlying toolkit. This is benificial for targeting a wide variation of platforms, but most of the cross-platform toolkits are subject to some restrictions (e.g. speed of execution, user experience, availability of certain widgets, only available for a subset of platforms...) [20]. Jelly takes it one step further: it can produce consistent UIs for different platforms while using different toolkits. Jelly is a design environment that supports designing for a set of arbitrary computing platforms, regardless of the availability of a cross platform toolkit that covers this set.

Damask [14] is a UI design system that allows designers to prototype multi-device UIs and employs *layers* to manage the consistency of these UIs across devices. When adding a component to the *all devices* layer, Damask uses an automatic transformation to add a version of this component to all devices. Jelly differs in two ways from this work. First, Jelly does not use layers but enables designers copy components across devices. Each time an element is copied from one device to another, designers can select from a list of available widgets how this element should look on the other device. This provides more fain grained control over the transformation process. Second, Jelly focuses on creating running UIs on top of existing toolkits instead of sketching low fidelity prototypes.

Existing design environments that can produce running GUIs for multiple devices and toolkits are SketchiXML [4], the mixed fidelity framework [5] and Gummy [15]. Each of these systems provides designers with a *GUI builder workspace* to shape UIs for multiple devices. Since Jelly also targets the creation of one design environment for many computing platforms, its contributions are complementary. Jelly goes beyond these multi-device design environments in two ways: it contributes a novel design tool architecture that can target a very broad set of computing platforms, and it introduces a linked editing technique to keep the content of UIs consistent across these different platforms.

Linked editing was first introduced by Toomim et al. in the Codelink programming environment to edit duplicated code fragments as one [23]. Hartmann et al. extended this technique to maintain UI design alternatives in the Juxtapose programming environment [10]. Jelly shares the motivation of the two aforementioned approaches to edit duplicated items in concert. While Codelink and Juxtapose focus on *textual* programming, Jelly integrates this technique in a *graphical* design environment to edit the content of duplicated UI elements across different platforms.

8. CONCLUSION AND FUTURE WORK

This paper presented a set of techniques to design and manage UIs for multiple devices integrated in a single multi-device UI design environment: Jelly. In Jelly, UIs can be designed for multiple computing platforms in parallel. The tool allows designers to copy elements from one device design canvas to another while preserving the consistency of their content across devices using linked editing. Jelly's underlying architecture is designed to cover a broad set of computing devices and facilitates instant UI testing using device emulators. In an informal study with eight participants, we found that they were enthusiastic about Jelly's techniques and would like to use such a system in their work.

Future research will explore how Jelly can be used for designing interface behaviour that goes with the graphical interface design. We will evaluate how designers and programmers can benefit from a multi-device design environment to add behaviour to their cross-platform UI designs. This allows for the creation of more complex and complete user interfaces.

Based on our current findings we continue the development of Jelly and release it as open source software[4]. Thorough validation is still needed to estimate the value of Jelly in long-term multi-device UI design projects. Especially the role of Jelly during the software maintenance phase seems to be an important area for future research.

[4] `http://research.edm.uhasselt.be/~jmeskens/jelly/`

Acknowledgments

This research was funded by the AMASS++ (Advanced Multimedia Alignment and Structured Summarization) project IWT 060051, which is directly funded by the IWT (Flemish subsidy organization).

9. REFERENCES

[1] AWT - Abstract Window Toolkit. `http://java.sun.com/products/jdk/awt/`.

[2] Qt - Cross-platform application and UI framework. `http://qt.nokia.com/`.

[3] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose. Toolglass and magic lenses: the see-through interface. In *Proc. SIGGRAPH '93*, pages 73–80, New York, NY, USA, 1993. ACM.

[4] A. Coyette, S. Kieffer, and J. Vanderdonckt. Multi-fidelity prototyping of user interfaces. In *Proc. INTERACT'07*, 2007.

[5] M. de Sá, L. Carriço, L. Duarte, and T. Reis. A mixed-fidelity prototyping tool for mobile devices. In *Proc. AVI '08*, pages 225–232. ACM, 2008.

[6] A. Demeure, G. Calvary, J. Coutaz, and J. Vanderdonckt. The comets inspector. In *Proc. CADUI'06*, pages 167–174, 2006.

[7] K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *Proc. IUI '04*, pages 93–100. ACM, 2004.

[8] K. Z. Gajos, J. O. Wobbrock, and D. S. Weld. Improving the performance of motor-impaired users with automatically-generated, ability-based interfaces. In *Proc. CHI '08*, pages 1257–1266. ACM, 2008.

[9] V. Grigoreanu, R. Fernandez, K. Inkpen, and G. Robertson. What designers want: Needs of interactive application designers. In *Proc. VL/HCC'09*. IEEE Computer Society, 2009.

[10] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proc. UIST '08*, pages 91–100. ACM, 2008.

[11] J. Helms and M. Abrams. Retrospective on ui description languages, based on eight years´ experience with the user interface markup language (uiml). *IJWET'08*, 4(2), 2008.

[12] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Proc. of VLHCC '04*, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society.

[13] J. Lin. *Using Patterns and Layers to Support the Early-Stage Design and Prototyping of Cross-Device User Interfaces*. Dissertation, University of California, Berkeley, 2005.

[14] J. Lin and J. A. Landay. Employing patterns and layers for early-stage design and prototyping of cross-device user interfaces. In *Proc. CHI '08*, pages 1313–1322. ACM, 2008.

[15] J. Meskens, J. Vermeulen, K. Luyten, and K. Coninx. Gummy for multi-platform user interface designs: Shape me, multiply me, fix me, use me. In *Proc. AVI'08*. ACM, 2008.

[16] G. Mori, F. Paterno, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Eng.*, 30(8):507–520, August 2004.

[17] B. Myers, S. E. Hudson, and R. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.

[18] B. Myers, S. Y. Park, Y. Nakano, G. Mueller, and A. Ko. How designers design and program interactive behaviors. In *Proc. VLHCC '08*, pages 177–184, Washington, DC, USA, 2008. IEEE Computer Society.

[19] J. Nichols, D. Horng Chau, and B. A. Myers. Demonstrating the viability of automatically generated user interfaces. In *Proc. CHI'07*. ACM, 2007.

[20] E. G. Nilsson. Combining compound conceptual user interface components with modelling patterns - a promising direction for model-based cross-platform user interface development. In *Proc. DSV-IS'02*, 2002.

[21] D. R. Olsen, Jr., S. Jefferies, T. Nielsen, W. Moyes, and P. Fredrickson. Cross-modal interaction using xweb. In *Proc. UIST '00*, pages 191–200, New York, NY, USA, 2000. ACM.

[22] J. S. Pierce and J. Nichols. An infrastructure for extending applications' user experiences across multiple personal devices. In *Proc. UIST '08*, pages 101–110. ACM, 2008.

[23] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Proc. VLHCC '04*, pages 173–180. IEEE, 2004.

[24] J. Vanderdonckt. Advice-giving systems for selecting interaction objects. In *Proc. UIDIS '99*, page 152. IEEE Computer Society, 1999.

[25] J. Vermeulen, Y. Vandriessche, T. Clerckx, K. Luyten, and K. Coninx. Service-interaction descriptions: Augmenting services with user interface models. In *Proc. EIS'07*. Springer, March 2007.