Chapter 3
Distribution Primitives for Distributed User Interfaces

Jérémie Melchior, Jean Vanderdonckt, and Peter Van Roy

Abstract This paper defines a catalog of distribution primitives for Distributed User Interfaces. These primitives are classified into four categories and represents operations that the developers and/or the users can executes to distribute the UI.

Keywords Distribution primitive • Catalog • Distributed user interfaces • Distribution operation

3.1 Introduction

The domain of Distributed User Interfaces (DUI) is still in evolution and there exist no toolkit allowing the creation of DUIs. In most pieces of work, there is almost no genuine DUI. There exists toolkits to create UI such as Java Swing or Microsoft Foundation Classes, but they do not support DUIs [1]. The UI elements simply remain in their initial context, while communicating with each other, but without redistribution. There is some distribution of UI elements, but it is mainly predefined and opportunistic: no configuration of the distribution at run-time. In Sjölund [2], the repartition of UI elements across the Smartphone and the TV is fixed. It is not possible to rearrange their distribution. Some works allow distribution at run-time but with some limitations. The UI elements subject to this redistribution are mainly containers, such as windows or dialog boxes. The problem is that the granularity of UI distributed elements is often coarse-grained; it is not possible to distribute at the widget level.

J. Melchior • J. Vanderdonckt (✉)
Louvain School of Management, Université catholique de Louvain, Louvain-la-Neuve, Belgium
e-mail: jean.vanderdonckt@uclouvain.be

P. Van Roy
Louvain School of Management, Université catholique de Louvain, Louvain-la-Neuve, Belgium
Department of Computing Science and Engineering, Université catholique de Louvain, Louvain-la-Neuve, Belgium
e-mail: peter.vanroy@uclouvain.be

J.A. Gallud et al. (eds.), *Distributed User Interfaces: Designing Interfaces for the Distributed Ecosystem*, Human-Computer Interaction Series,
DOI 10.1007/978-1-4471-2271-5_3, © Springer-Verlag London Limited 2011

In addition, they do not support replicability, i.e. when another platform comes in the context of use, it is hard to migrate on this platform parts that have already been transferred to other platforms.

In Luyten [3], there are already attempts to model the distribution. The granularity is however limited to tasks that are predefined before the application starts.

To sum up, we are looking for a way to support distribution at both design-time and run-time with very fine and coarse-grained granularity and to support replicable distribution while being compliant with the DUI goals as in [4]. This paper tends to help understanding and managing DUI.

3.2 Catalog of Distribution Operations

We propose a catalog of distribution operations and a toolkit based on this catalog.

3.2.1 Toolkit

A toolkit has been developed upon the catalog. It creates application with the UI separated in two-parts: the proxy and the rendering. A command line interface is provided to allow manual redistribution at run-time, see Fig. 3.1.

In Fig. 3.2, the proxy is represented as a separate part of the application than the rendering. The first keeps the state of the application and ensures the core functionalities, while the second displays the user interface. Applications that support DUI allow the rendering part to be distributed on other platforms while the proxy stays on the platform where the application has been created. The toolkit works in Mozart environment supported by Microsoft Windows operating systems (XP and newer), Apple Mac OS X, Linux and Android. We are currently working on the full support for Apple iOS. As Mozart is a multi-platform environment, the applications created with this toolkit are also multi-platform. Each graphical component is described as a record containing several keys and values. It ensures compatibility with the XML format because the keys/values become the name/value pairs for the XML markup. The DUI can be controlled by a command line interface, a meta-UI or even by the applications themselves using distribution scenario. A model-based approach closely related to the toolkit has already been described [5]. The definitions and examples presented in this paper come from this paper.

3.2.2 Definitions

As observed in the related work section, the distribution logic of DUIs is often hard-coded and is not represented explicitly, which prevents us from reasoning on how

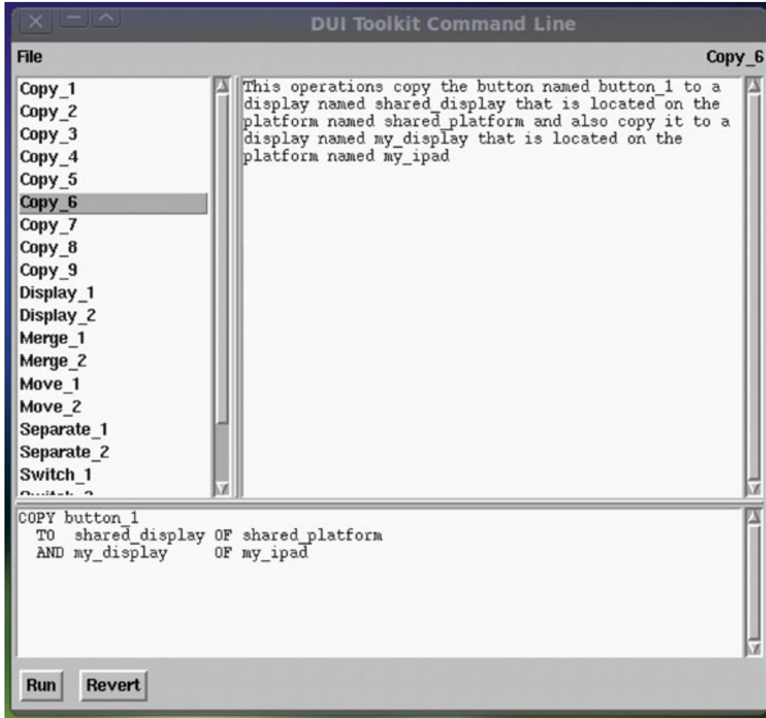

Fig. 3.1 Command line interface provided by the toolkit

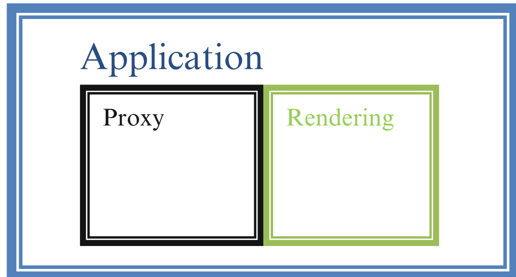

Fig. 3.2 Structure of a DUI application

distribution is operated. In order to address this issue, we now provide a catalogue of distribution primitives that will operate on CUI models of the cluster. We first define these distribution primitives in natural language, then in an Extended Backus-Naur Form (EBNF) format. In this notation, brackets indicate an optional section, while parentheses denote a simple choice in a set of possible values. In the following definitions, we use only one widget at a time for facilitating understanding. In the EBNF, we will use the four selector mechanism standard from W3C for CSS for generalization to all widgets.

SET <Widget.property> TO {value, percentage} [ON <Platform>]: assigns a value to a CUI widget property or a percentage of the actual value on a platform identified in a cluster. For instance, SET "pushButton_1.height" TO 10 will size the push button to a height of 10 units while SET "pushButton_1.height" TO +10 increases its height by 10%. Note that the platform reference is optional: when it is not provided, we assume that the default platform is used.

DISPLAY <Widget> [AT x,y] [ON <Platform>]: displays a CUI widget at a x,y location on a platform identified in a cluster, where x and y are integer positions (e.g., in characters or pixels). For instance, DISPLAY "pushButton_1" AT 1,1 ON "Laptop" will display an identified push button at coordinates 1,1 on the laptop. UNDISPLAY <Element> [AT x,y] [ON <Platform>] is the inverse operation. DISPLAY <Message> [AT x,y] [ON <Platform>] displays a given message on a designated platform in the cluster (mainly for user feedback in an optional console).

COPY<Widget>[ON<SourcePlatform>]TO[<Widget>][ON<TargetPlatform>]: copies a CUI widget from a source platform identified in a cluster to a clone on a target platform, thus creating a new identifier. This identifier can be provided as a parameter to the primitive or created automatically by the primitive to handle it.

MOVE <Widget> TO x,y [ON <TargetPlatform>] [IN n steps]: moves a CUI widget to a new location indicated by its coordinates x and y, possibly in a fixed amount of steps, on a target platform in the cluster.

REPLACE <Widget1> BY <Widget2>: replaces a CUI widget Widget1 by another one Widget2. Sometimes the replacement widget could be determined after a (re-)distribution algorithm, thus giving the following definition: REPLACE <Widget1> BY <Algo:>. This mechanism could be applied to contents and image transformations: images are usually transformed by local or remote algorithms (e.g., for resizing, converting, cropping, clipping, repurposing), thus giving the following definition: TRANSFORM <Image1> BY <ImageAlgo:URL>.

MERGE <Widgets> [ON <SourcePlatforms>] TO [<Widget>] [ON <TargetPlatform>]: merges a collection of CUI widgets from a source platform identified in a cluster into a container widget on a target platform, thus creating a new identifier. Again, when source and target platforms are not provided, we assume that the default platform is used. SEPARATE is the inverse primitive. SEPARATES <Widgets> [ON <SourcePlatforms>] TO [<Widgets>] [ON <TargetPlatforms>]: splits a collection of CUI widgets (typically, a container) from a source platform identified in a cluster into CUI widgets on one or many target platforms.

SWITCH <Widget> [ON <SourcePlatforms>] TO [<Widget>] [ON <TargetPlatform>]: switches two CUI widgets between two platforms. When the source and target platforms are equal, the two widgets are simply substituted.

DISTRIBUTE <Elements> INTO <Containers> [BY <DistribAlgo:URL>: computes a distribution of a series of UI Elements into a series of UI Containers, possibly by calling an external algorithm, local or remote.

EBNF Grammar. In order to formally define the language expressing distribution primitives, an Extended Backus Naur Form (EBNF) grammar has been defined. EBNF only differs from BNF in the usage of the following symbols: "?" means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional,

statement = operation , white_space , source , white_space , ”TO” , white_space , target ;

operation = "SET" | "DISPLAY" | "UNDISPLAY" | "COPY" | "MOVE" | "REPLACE" | "TRANSFORM" | "MERGE" | "SWITCH" | "SEPARATE" | "DISTRIBUTE";

source = selector ;

target = displays | selector , white_space , “ON” , white_space , displays ;

displays = display_platform , { “,” , display_platform}

display_platform = display , [white_space , “OF” , white_space , platform] ;

selector = identifier , { “,” , identifier } | universal ;

display = identifier ;

platform = identifier ;

Fig. 3.3 EBNF grammar for distribution primitives (excerpt)

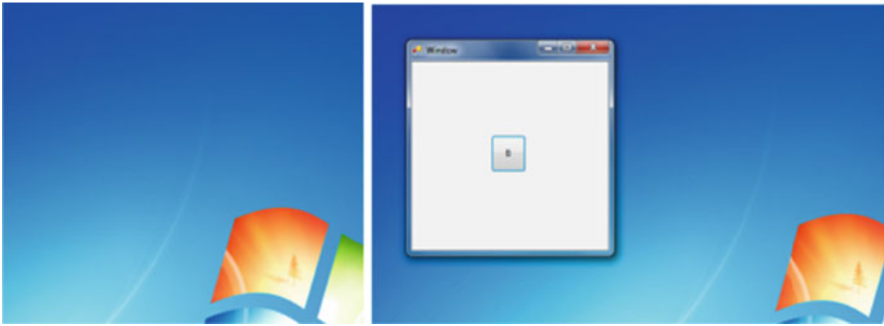

Fig. 3.4 Example of a simple display primitive

“*” means that something can be repeated any number of times, and “+” means that something can appear one or more times. EBNF has been selected because it is widely used to formally define programming languages and markup languages (e.g., XML and SGML), the syntax of the language is precisely defined, thus leaving no ambiguity on its interpretation, and it is easier to develop a parser for such a language, because the parser can be generated automatically with a compiler (e.g., YACC). Instances of distribution primitives are called by statements. The definitions of an operation, a source, a target, a selector and some other ones are defined in Figs. 3.3 and 3.4 (excerpt only). The definitions could be extended later to support more operations or features.

3.2.3 Examples

In order to illustrate how distribution primitive could behave, we hereby provide a series of increasingly complex examples. In Fig. 3.4, a display of the platform by default has been modified in the following way: DISPLAY “pushButton_1” AT 5,5

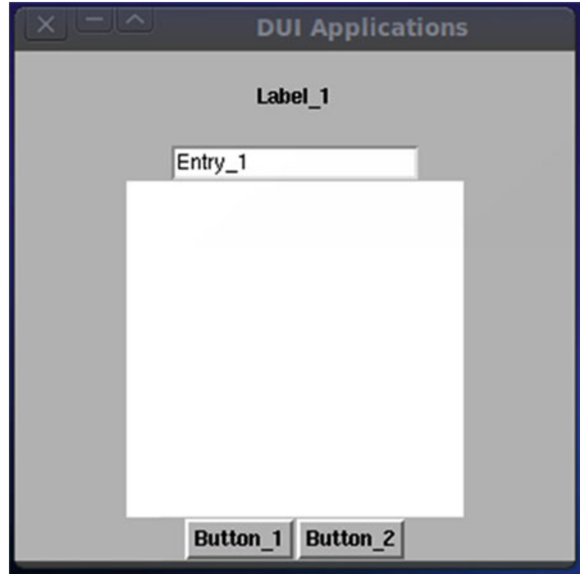

Fig. 3.5 Source CUI for the COPY examples

ON "defaultPlatform" followed by SET "pushButton_1.label" TO "B", thus creating a CUI model attached to this platform.

Distribution operations can be more complex than the example provided in Fig. 3.4. Here is a series of examples for the COPY primitive:

1. COPY button_1 TO shared_display: simple copy of button_1 sent to shared_display without specifying neither an identifier nor a container
2. COPY button_1 TO button_2 ON shared_display: copy button_1 on shared_display and identify it as button_2
3. COPY button_1 TO button_2 ON shared_display of shared_platform: the same but we specify the shared_platform to avoid searching through all the platforms
4. COPY button_1, button_2 TO shared_display: copy button_1 and button_2 to shared_display in a single operation
5. COPY button_1 TO shared_display, my_display: copy button_1 to shared_display and also to my_display
6. COPY button_1 TO shared_display OF shared_platform AND my_display OF my_ipad: copy button_1 to both shared_display and my_display, specifying on which platform is each display
7. COPY * TO shared_display: copy all the graphical components from the current UI to shared_display
8. COPY ALL buttons TO shared_display: copy all buttons to shared_display
9. COPY individuals TO shared_display: copy any individual CUI widgets to shared_display

The source CUI associated to these examples is reproduced in Fig. 3.5, while the results of the nine above copy operations are reproduced respectively in the different regions of Fig. 3.6.

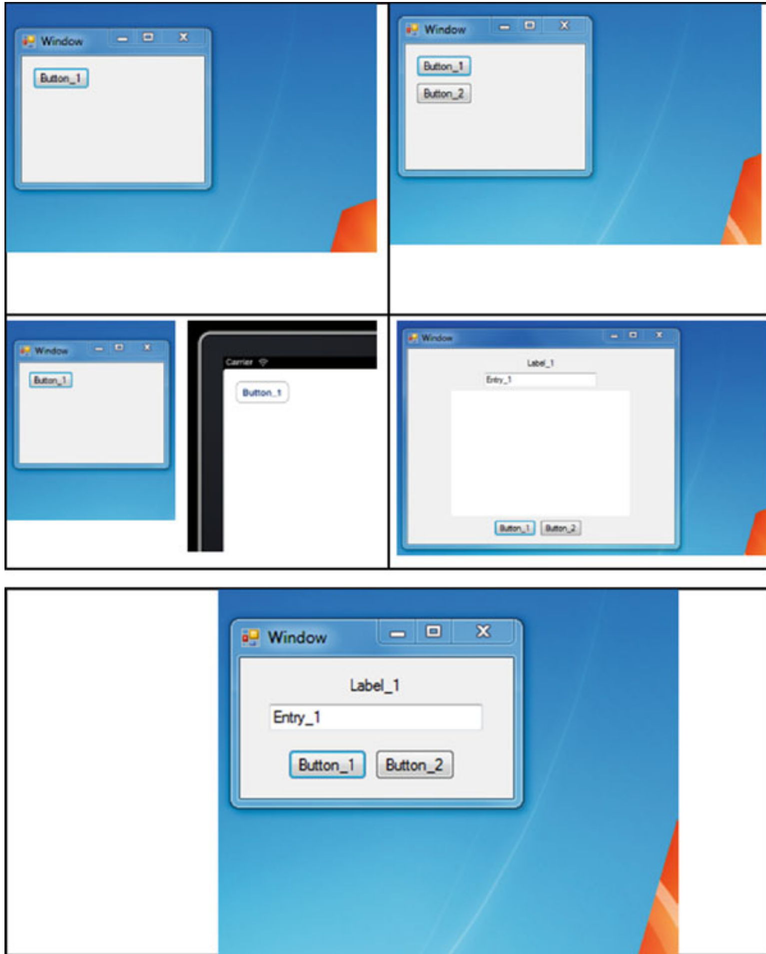

Fig. 3.6 Results of examples 1, 2 & 3 (*top left*), 4, 8 (*top right*), 5 & 6 (*middle left*), 7 (*middle right*), 9 (*bottom*)

Meta-User Interface for Distribution Primitives. The distribution primitives defined in the previous subsection can be called in two ways:

1. Interactively through a meta-UI providing a command line equipped with a command language: in this way, one can type any distribution primitive through statements that are immediately interpreted and provide immediate feedback. This meta-UI adheres to usability guidelines for command languages (such as consistency, congruence, and symmetry), but does not provide for the moment any graphical counterpart of each statement or graphical representation of the platforms of the cluster. Actually, each platform is straightforwardly addressed at run-time. It is of course possible to see the results of a distribution primitive immediately by typing it by an “errors and trials” process until the right

statement is reached. Figure 3.1 reproduces a screen shot of this meta-UI, which also serve as a tutorial to understand how to use the distribution primitives. Indeed, any statement type in the command language can be stored in a list of statements that could be recalled at any time.
2. Programmatically: each statement representing an instance of a distribution primitive can be incorporated in an interactive application in the same way since the parser will be called to interpret it. It is therefore no longer needed to program these primitives.

3.3 Future Work

These distribution primitives have been defined in such a way that it allows distributing any graphical widget without focusing on how the value is shared when distributed. There are two important aspects in a User Interface: the *presentation* and the *behavior* of the UI. The presentation part is the graphical representation of a widget. If the widget is moved to another platform, it means that the same widget will be displayed on the destination platform. This leads to an ambiguous solution about the behavior. Is the widget still acting like it was on the other platform? Is the action taking place locally (to the new platform) or globally (in the same way than before being moved, like a remote)?

The behavior is an important future work to keep in mind.

3.4 Conclusion

In this paper, we have introduced the concept of distribution primitives and a toolkit based on them. The goal is to provide a catalog of distribution primitives as a common base for researchers on DUIs. They now have the same set of primitives. It allows them to share the same possibilities independently of the UI implementation.

A toolkit based on this catalog has also been introduced and allows developers to see the powerfulness of this catalog.

Acknowledgments The author would like to acknowledge the support of the ITEA2-Call3-2008026 USIXML (User Interface extensible Markup Language) European project and its support by Région Wallone.

References

1. Tan, D.S., Czerwinski, M.: Effects of visual separation and physical discontinuities when distributing information across multiple displays. In: Proceedings of INTERACT'03, pp. 252–260. IOS, Zurich (2003)

2. Sjölund, M., Larsson, A., Berglund, E.: Smartphone views: building multi-device distributed user interfaces. In: Proceedings of MobileHCI'2004, LNCS, pp. 507–511. Springer, Berlin (2004)
3. Luyten, K., Van den Bergh, J., Vandervelpen, C., Coninx, K.: Designing distributed user interfaces for ambient intelligent environments using models and simulations. Comput. Graph. **30**(5), 702–713 (2006)
4. Vanderdonckt, J.: Distributed user interfaces: how to distribute user interface elements across users, platforms, and environments. In: Proceedings of Interacción'2010, pp. 3–14. AIPO, Valencia (2010)
5. Melchior, J., Vanderdonckt, J., Van Roy, P.: A model-based approach for distributed user interfaces. In: Proceedings of EICS'11, pp. 11–20. ACM, Pisa (2011)