A Model-Based Approach for Distributed User Interfaces

Jérémie Melchior[1], **Jean Vanderdonckt**[1], **and Peter Van Roy**[2]
Université catholique de Louvain
[1]Louvain School of Management, Place des Doyens, 1 – B-1348 Louvain-la-Neuve, Belgium
[2]Computer Science Department, Place Sainte-Barbe, 2 – B-1348 Louvain-la-Neuve, Belgium
+32 10 47 {8379, 8525, 8374} - {jeremie.melchior, jean.vanderdonckt, peter.vanroy}@uclouvain.be

ABSTRACT
This paper describes a model-based approach for designing distributed user interfaces (DUIs), i.e. graphical user interfaces that are distributed along one or many of the following dimensions: end user, display device, computing platform, and physical environment. The three pillars of this model-based approach are: (i) a Concrete User Interface model for DUIs incorporating the distribution dimensions and able to express in a XML-compliant format any DUI element until the granularity of an individual DUI element is reached, (ii) a specification language for DUI distribution primitives that have been defined in a user interface toolkit, and (iii), a step-wise method for modeling a DUI based on the concepts of distribution graph expressing a distribution scenario that can be played namely based on the distribution primitives. A distribution graph consists of a state-transition diagram whose states represent significant distribution states of a DUI and whose transitions are labeled by an even-condition-action representation. The actions involved in this format may call any distribution primitive of the DUI toolkit. In order to exemplify this model-based approach, two simple DUIs are first designed: a DUI for the Pictionary game and a DUI for the Minesweeper game. They are then incorporated into a larger DUI game of the goose where cells may trigger the two other games.

Authors Keywords
Distribution graph, primitives, and scenario, Distributed User Interface, Meta-user interface, Shared displays, Ubiquitous computing, User Interface Toolkit.

General Terms
Design, Experimentation, Human Factors, Verification.

ACM Classification Keywords
C.2.4 [**Computer-Communication Networks**]: Distributed systems – *Distributed applications*. D2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; user interfaces*. D2.m [**Software Engineering**]: Miscellaneous – Rapid Prototyping; reusable software. H5.2 [**Information interfaces and presentation**]: User Interfaces – *graphical user interfaces, user interface management system (UIMS)*.

EICS'11, June 13–16, 2011, Pisa, Italy.

INTRODUCTION

On the side of the *demand*, end users are more frequently involved in a context of use [21] where domain objects are widespread, where roles and groups are configured in dynamic fashions, thus increasing the need for User Interfaces (UIs) that support them in these multiple configurations. On the side of the *offer*, the market has disseminated a large amount of computing platforms ranging from smartphones to wall screen displays, thus offering a wide spectrum of interaction surfaces to interact with [29]. End users are puzzled by what type of platform they should choose for a particular task, especially when several tasks are distributed in time and space. For instance, when an end user delegates a task to a colleague, parts or whole of this task UI should be transferred as well to the colleague. Even at run-time, an end user may want to ask for advice for a remote colleague, thus requesting to access to the currently running UI.

All these examples strive for a global approach for designing a *Distributed User Interface* (DUI) [5,26,28,35], which is defined as any application UI whose components can be distributed across varying displays of varying platforms that are used by varying users, whether they are working at the same place (co-located) or not (remote collaboration). DUIs have been successfully used in various domains of human activity (e.g., ambient intelligence [22,23], clinical systems [5], economics [14]) and in computer science (e.g., migratory UIs [3,6,25,39], service-oriented architecture [36], ubiquitous computing [7]). DUIs are fundamental because several applications require the integration of distributed interaction devices as functional wholes. There are two main categories in which they can be important. The user needs for DUI and addition to the limited UI development. Many situations need collaboration between users, they share their computing tasks and so they should be able to share their UIs [21]. Current toolkits such as Java Swing or Microsoft Foundation Classes do not support DUIs [34].

The remainder of this paper is structured as follows: the next section reports on some significant DUI related work. Then, a specification language for UI distribution primitives is motivated, defined, and exemplified, based on a model of a Concrete UI [15]. Then, a step-wise method for modeling a DUI based on a distribution scenario represented as a distribution graph involving the abode primitives is presented. A progressive case study will then exemplify how this method can be applied on games that are intrinsically challenging and distributed by nature, first individually, then composed. Finally, a conclusion delivers the main points of this research and presents some future avenues.

RELATED WORK

DUI may be subject to distribution along many dimensions [3,4,11,28] such as user, platform (including various displays/monitors connected to the same computing platform or belonging to the same cluster). Analyzing extensively these distributions is beyond the scope of this paper. Therefore, we only focus on some significant pieces of work to examine some of these dimensions in this state of the art.

User requirements for DUIs

The "display" of the platform dimension is certainly one of the most vivid dimensions of investigation. Beale and Edmondson [4] identified several requirements for distributing or reproducing UI elements from one display to another depending on the task being carried out. They conducted user surveys in order to determine the user behavior induced by using a DUI: they identified the importance of having multiple carets and the complexity of multi-tasking and they suggest design implications for using DUIs in order to support distributed tasks. In particular, they stressed the importance of a multi-tasking model that is partially built at the local level of a single user and at the global level across users when collaboration exists. The global scenario should be also dissolved into local scenario in order to preserve the consistency between common tasks and individual tasks. This observation is fundamental here.

Tan & Czerwinski [33] found out that physical discontinuities had no effect on performance, but found a detrimental effect from separating information within the visual field, when also separated by depth. Grudin [18] highlights usability issues with multi-display such as lack of support and mobile device not used to display. This requirement is acknowledged in Personal Universal Control (PUC) [29], a system that enables controlling a primary platform (such as a desktop or a wall screen) from a secondary platform (such as a PDA or a smartphone). Berglund *et al.* [5] have set user requirements that DUI have to meet: to enable dynamic construction and distribution, to make the best possible use of the available resources, to provide graceful degradation of interaction, to negotiate the interaction resources needs to be handled dynamically and transparent configuration and UI distribution. Distribution also raises concerns for security and privacy [4,20]: when allowing the distribution of a window or application, it allows all users to control or observe all shared windows. But sometimes, users would prefer to avoid users being able to get some windows or applications. Not all users should benefit from sharing the same window, thus requiring control over distribution [12].

Nowadays, more user studies are available on specific DUI setups that provide us with more knowledge on design implications for such DUIs. Yet, in order to allow for the user to get the best potential of interaction capabilities offered by the various devices/displays/platforms for the current task to be carried out, we should enable designers as well as developers to provide users with the best DUI possible for a given set of devices/displays/platforms by described them in a formal way. This is discussed in the next subsection.

System requirements for DUIs

The distribution of UI elements in existing toolkits and frameworks is always at a very high level. Several works [3,6,16] have been done in the domain of migratory applications, where parts or whole of a UI moves from one computing platform to another while preserving the task continuity [3]. WinCuts [34] supports some distribution by clipping areas of interest in windows and arranging them together in order to spare screen space. This distribution is located at the visualization level, not at the UI level. In IMPROMPTU framework [7], only windows can be moved from one wall screen to another through a direct manipulation control. There are however ample possibilities for the granularity of distribution: at the level of applications, windows, widgets, and pixels [11]. Most toolkits supporting distribution support applications [6] and/or windows levels [8,16].

Distributed Programming [37] mainly focuses on the distribution of the functionalities belonging to the functional core of an interactive application [13], leaving the UI distribution aside. DUIs system support remains today only in its infancy. To be able to develop application with fully controllable application, more research has to be done [12]. DUIs could be integrated in domains such as workflows, collaborative tools and for user that would like ubiquitous computing across several devices [3].

Consequently, DUIs allow for the UI to be spread out over a set of displays/devices/platforms taking advantage of their unique properties instead of residing on a single display/device/platform with the interaction capabilities that are constrained on this display/device/platform [3,35]. DUIs have been subject to several studies that investigate further their specific characteristics that may lead to design implications. This includes the use of multiple monitors on the same computing platform by a single user [14,18,38], the use of multiple platforms by a single user with synchronization between [20], exchange of information between platforms belonging to different users [12], moving information between displays on a single platform [28], partition of tasks across displays for a single user [34], sharing common information on a common display while keeping some information private on a own platform [5].

This will allow both designers and developers to enable the underlying system to decide where different DUI portions should be placed in locations that are significant and usable for a distributed task to take place. For instance, the game of Pictionary [27] is a typical example of a distributed task: one player selects a word from a dictionary, a second player draws this word on a surface shared by other players who have to guess what this word is as quick as possible, but below a certain time threshold. The team to which the winning player belongs to receives points. Other tasks are distributed by nature. Sjölund *et al.* implemented a DUI consisting of a remote control GUI on a smartphone that controls Windows Media Player displayed on a TV. Some controls of the Windows Media Player (e.g., play, stop, volume up, volume down) are moved to the smartphone and adapted to this platform at the same time.

Shortcomings of related work

In most of the aforementioned pieces of work, there is almost *no genuine DUI* since UI elements have been developed in such a way that they simply remain in their initial context, while communicating with each other, but without any possibility to be rearranged. Where there is still some distribution of UI elements, it is mainly *predefined* and *opportunistic*: it is impossible to change the distribution of these UI elements because they have been thought in only one particular context of use. For example, the recombinant computing approach induces a small set of DUIs for mobile services such as projecting, printing or storing files [30], but they are predefined. In Sjölund [32], there is no other way to change the repartition of UI elements across the smartphone and the TV. In HyperPalette [2], the distribution of tasks across windows is also predefined, although these windows could be moved. When a distribution is anyway possible at run-time, the UI elements subject to this redistribution are mainly *containers*, such as windows or dialog boxes. For instance, WebSplitter [19] redistributes portions of web pages, WinCuts [34] redistributes window viewports, DistriXML [17] detaches and attaches toolbars depending on the current task. In other words, the *granularity* of UI distributed elements is often *very high*.

In addition, it is not *replicable*. If another user or platform comes in the context of use, it is hard to replicate or migrate on this platform the part that has already been transferred to the first smartphone. In Lightweight services [36], once a service has been selected, it can be distributed to any platform, existing or arriving, but a service can be distributed only once. On the one hand, this does not create any conflict, but on the other hand, it does not support *replication*. For this purpose, there is a need to have an underlying model that captures the various aspects of distribution, which is not the case nowadays. A platform model has been defined in order to accommodate dynamically new platforms or interaction resources coming in play [31], but there is no explicit representation of the distribution logical. Similarly, the Qtk toolkit [27] enables developing DUIs, but there is no underlying model or any representation of replication possible. Models [25] or logical UI descriptions [23] have already been successfully used for partial aspects of DUIs and could expanded with a development method based on primitives for UI distribution explicitly.

To sum up, we are looking for a model-based approach for designing DUIs that support both design-time and run-time distribution of UI with very low granularity and replicability, while being compliant with the DUI goals [35]. In order to address these challenges, we will follow the three pillars of any development method: *models* that capture relevant aspects of the future software (along with a *specification language* that expresses these models), a *step-wise approach* based on the previously defined models, and *software* supporting the step-wise approach. The next section is dedicated to the first pillar (what are the core models required for DUI), the subsequent section is devoted to the second pillar (what is the step-wise method, exemplified on three progressively more complex case studies), and the last section will be about the third pillar (i.e., how the implementation has been achieved).

CORE MODELS FOR DISTRIBUTED USER INTERFACES

The following subsections define respectively the UI model subject to distribution, the models for representing the context of use (i.e., platform, user, and environment), a set of distribution primitives that enable distribution of UI elements specified in the UI model according to the context of use, and a formal language expressing these primitives.

Concrete User Interface (CUI) Model

There has been considerable research and development in defining a *Concrete User Interface* (CUI) model that is independent of any computing platform and implementation language (programming or markup) and their associated User Interface Description Language (UIDL). It is interesting to notice that this is now subject to a common effort under the umbrella of the W3C Incubator Group on Model-Based UIs (). Since the main goal of this paper is to introduce a model-based approach for developing DUIs, we will be using for the purpose of this paper a simple representation of a CUI model. A *CUI model* is hereby defined as recursive hierarchy of containers (e.g., windows, tabbed dialog boxes, group boxes) and individual widgets (e.g., check boxes, push buttons, list boxes, etc.). Widgets are laid out in their containers according to the Box & Glue mechanism [1]: a window is structured into horizontal and vertical boxes, that are in turn, decomposed into boxes, until individual widgets are defined. Each widget is defined as a vector $W=(P_i, V_i)$ where P_i denotes the i^{th} property of the widget and V_i denotes the value of this property (e.g., the widget identifier, the background color of a push button is grey).

An identifier serves as a handle for referring to any particular widget. But it may be convenient to designate a series of widgets that satisfy a certain condition. For this purpose, we introduce the concept of selector. A *selector* consists of a selection of UI element types of a particular CUI model that satisfy a first-order predicate logical formula. In this way, it will be possible to apply a *template* for a selector instead of a (potentially long) sequence of widgets. Four types of selector are defined that will be later on used for addressing UI elements and for specifying source and targets elements of distribution primitives:

- universalSelector: applies the template to all UI elements belonging to a particular CUI, whatever they are.
- elementTypeSelector: applies the template to all UI elements belonging to a particular CUI which correspond to the selector′s type (e.g., all containers, all list boxes).
- classSelector: applies the template to all UI elements belonging to a particular CUI which correspond to the selector′s type whose definition makes them part of the class (e.g., all containers having an id greater or equal to 10, all list boxes having more than 10 items).
- idSelector: applies the template to only one UI element belonging to a particular CUI: the one whose id property matches the string contained in the parameter.

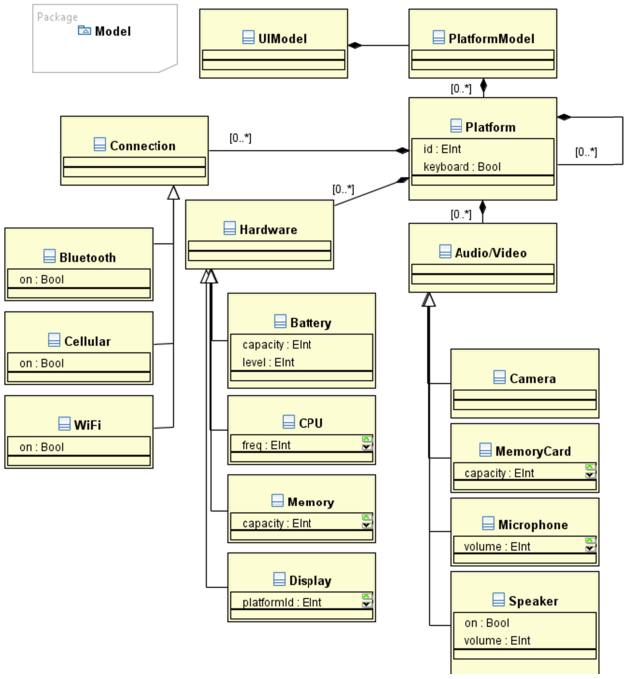

Figure 1. The platform model used for distributed user interfaces.

Platform Model

Again, considerable efforts have been devoted to defining and using models for representing the computing platform operated by the end user. In this work, the user is no longer considered in isolation as a single person carrying out a single task on a single computing platform in a unique environment, which has been the prevalent representation for year. Rather, we consider that a *UI distribution* [5,35] concerns the repartition of one or many UI elements from one or many DUIs in order to support one or many users to carry out one or many tasks on one or many domains in one or many contexts of use, each context of use consisting of users, platforms, and environments. Therefore, the context of use is hereby considered as a cluster of individual components. In order to represent this cluster, we adopted the *Delivery Context Ontology* (DCO) standardized by W3C [10], a subset of which is depicted in Fig. 1. According to DCO, a platform model is divided into one or many platforms. For example, a laptop itself consists of three platforms: the laptop, the display and the keyboard. Each platform has three main categories of components: a *connection* category representing the input and output connections to other devices or to the internet, the *hardware* category that defines the main components such as the CPU, the memory and if the platform has a display or not, and any component linked to the medias (e.g., audio and video).

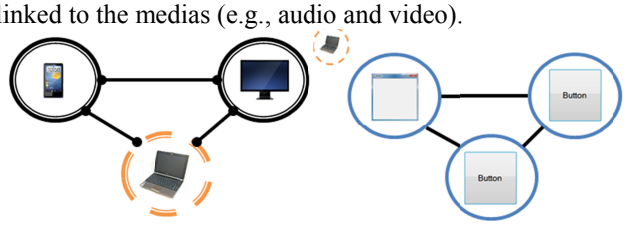

Figure 2. A distribution scene made up of: a cluster of three platforms (a) and an associated graph of CUI models (b).

Devices/ Feature	Mobile Phone	Laptop	Monitor
Battery	YES : 100%	YES : 100%	NO
Bluetooth	YES : OFF	YES : OFF	NO
Camera	YES : 5MP	YES : 1MP	NO
Cellular	YES : HSD-PA	NO	NO
CPU	YES : 1Ghz	YES : 2Ghz	NO
Display	YES : WVGA	YES : SXGA+	YES : HD 1080
Keyboard	YES	YES	NO
Memory Card	YES : 6Go	NO	NO
Microphone	YES	YES	NO
Speaker	YES	YES	NO
WiFi	YES : OFF	YES : ON	NO
Memory	512 MB	2 GB	NO

Table 1. Some properties and their values of the cluster of Fig. 1.

Based on this, a *cluster* is defined by a graph $G=(N_j,R_j)$ which is a set of nodes N_j connected together through R_j relationships. Each *node* consists of a DCO-compliant platform model representing any kind of device or components able to interact with the system (e.g., computers, displays, keyboards and mice are representative examples). Each relationship represents a communication channel (e.g., a Wi-Fi network or a BlueTooth connection) between nodes. Fig. 2a denotes a cluster composed of three platforms: a laptop connected to a flat monitor and a mobile phone, whose descriptions are partially reproduced in Table 1. In order to properly express DUIs, to operate them and to reason about them, it is required to know at time what DUI is residing on which platform of the cluster [21]. For this purpose, we hereby define a *distribution scene* as a cluster in which each node is associated to a CUI model, all CUI models connected to each other by a graph (Fig. 2b). Any cluster node contains a reference to a particular CUI model that could evolve over time. Consequently, a distribution scene holds a two-layer structure: (1) a cluster representing the physical setup of interaction elements and devices and (2) an associated graph of CUI models attached to any element in this cluster that supports some interaction. Not all platforms run a UI at any-time. To depict this, full circles in Fig. 2a represent that no DUI exist for those two platforms at some point (e.g., the starting time). The dashed circle around the laptop means that it holds some DUI.

User and Environment Models

A *User model* is hereby defined as recursive hierarchy of user profiles (e.g., average user, novice, expert) where each user profile is defined as a vector $U=(P_i, V_i)$ where P_i denotes the i^{th} property of the user (e.g., system experience, motivation, task experience, age, background) and V_i denotes the value of this property. An *Environment model* is hereby defined as recursive hierarchy of working places (e.g., ranging from a whole organization to an individual organization cell) where each working place is defined as a vector $U=(P_i, V_i)$ where P_i denotes the i^{th} property of the place (e.g., location, level of stress, light, temperature, organizational responsibilities) and V_i denotes its value.

All models defined so far are defined consistently so as to allow an internal representation for the implementation as a tree of nodes, each node having a vector of properties and values, thus facilitating the handling of models and model elements. This internal representation is not restrictive.

A Catalogue of Distribution Primitives

Definition. 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 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 for generalization.

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 Elments 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, "*" 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 Fig. 3 and 4 (excerpt only). The definitions could be extended later to support more operations or features.

```
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 ;
```

Figure 3. EBNF grammar for distribution primitives (partim).

Examples. In order to illustrate how distribution primitive could behave, we hereby provide a series of progressively more complex examples. In Fig. 4, a display of the platform by default has been modified in the following way: DISPLAY "pushButton_1" AT 5,5 ON "defaultPlatform" followed by SET "pushButton_1.label" TO "B", thus creating a CUI model attached to this platform.

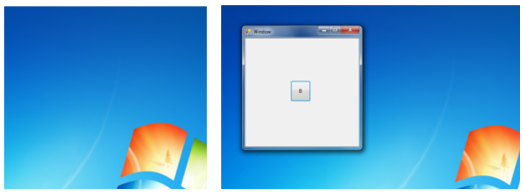

Figure 4. Example of a simple display primitive.

Obviously distribution operations could be more complex than the example provided in Fig. 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

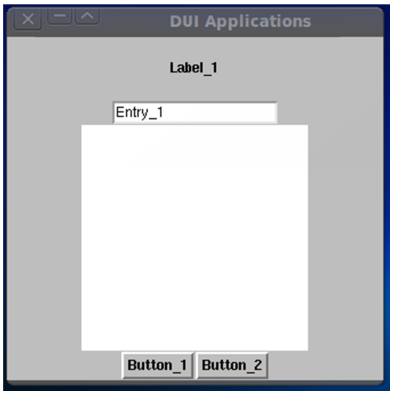

Figure 5. Source CUI for the COPY examples.

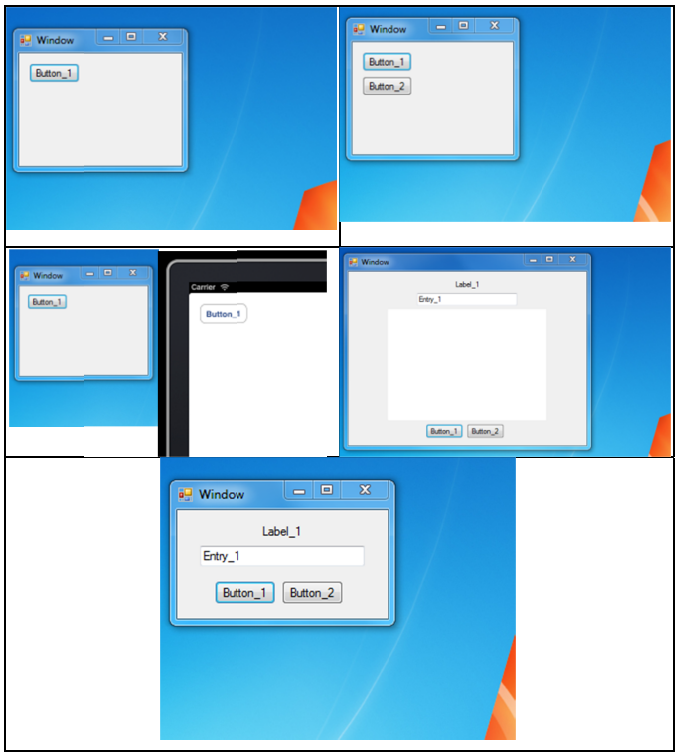

Figure 6. Results of examples 1, 2 & 3 (top left), 4, 8 (top right), 5 & 6 (middle left), 7 (middle right), 9 (bottom).

The source CUI associated to these examples is reproduced in Fig. 5, while the results of the nine above copy operations are reproduced respectively in the different regions of Fig. 6.

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

1. *Interactively* through a meta-UI [9] 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 a "errors and trials" process until the right statement is reached. Fig. 7 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.

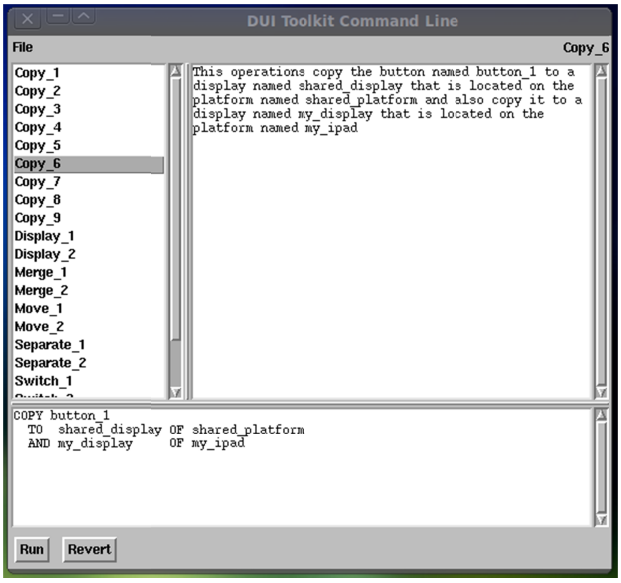

Figure 7. Command Line Interface for primitives.

IMPLEMENTATION

A toolkit has been developed upon the model-based approach. It creates application with UI separated in two-parts: the proxy and the rendering. In Fig. 8, 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. Application supporting DUI allows the rendering to be distributed on other platforms while the proxy stays where the application has been created. The toolkit works in an environment supported by Microsoft Windows operating systems (XP and newer), Apple Mac OS X, Linux and Android. And we are currently working on the full support for Apple iOS. The applications created with this toolkit are multi-platform. Each graphical component is described as a record containing several keys and values. It ensures compatibility with XML because the keys/values become the name/value pairs of the XML markup. The DUI can be controlled by a command line interface, a meta-UI or even by the applications themselves.

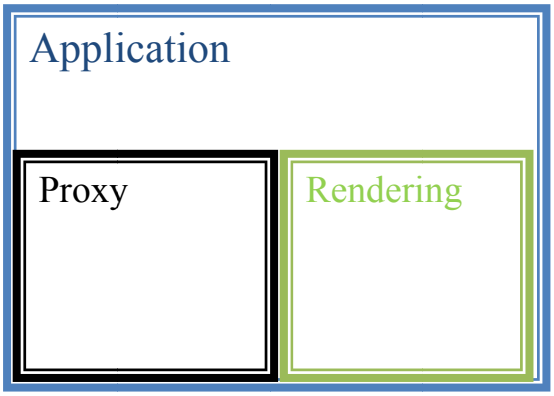

Figure 8. Structure of a DUI application.

A MODEL-BASED APPROACH FOR DUIs

Using the distribution primitives defined and exemplified in the previous section is an appropriate starting point for designing DUIs at a higher level of expression that simple code. After having defined the different models and these primitives, we outline a model-based approach for designing a DUI based on the aforementioned concepts:

1. Build a cluster model of the platforms.
2. Build a CUI model for each platform
3. Assemble models in the distribution scene
4. Write a distribution scenario based on distribution primitives.
5. Develop the distribution scenario

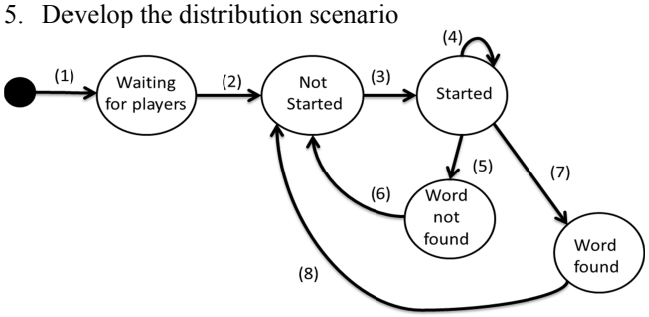

Figure 9. State-machine diagram of the Pictionary.

Pictionary

We now illustrate this method on some examples. A Pictionary game exemplified the mechanisms and evaluates the difference between a local Pictionary and a distributed Pictionary. In Figure 9, the STM shows that to start a game, the Pictionary needs several steps. The players will each have to assign or be assigned to a role. There are three roles: the player, the guessers and the observers. Two players are at least needed because the game needs one drawer and at least one guesser. The distribution does not appear in the states of the Pictionary. The transition can be of two types: with and without distribution. As in any STM, the transitions may have some guarding conditions. For readability issue, we put it apart from the STM. Also, the final state is not display on the diagram. In order to clearly state the transitions, they are all numbered from 1 to 8 and here are the transitions in the form IF condition THEN action:

1. IF new_player_event
   THEN DISPLAY pictionary_UI TO new_player
2. IF nb_player > 1
   THEN DISPLAY assign_UI TO Pictionary_UI OF players
3. IF drawer != null && guesser != null
   THEN UNDISPLAY assign_UI
   DISPLAY draw_UI TO Pictionary_UI OF drawer
   AND guesser_UI TO Pictionary_UI OF guessers
   UPDATE observe_UI TO observers
4. UPDATE draw_UI
5. IF timer <= 0 & !found
   THEN UPDATE draw_UI, guesser_UI
6. UNDISPLAY draw_UI, guesser_UI, observer_UI
   DISPLAY assign_UI TO players

7. IF timer > 0 & found
   THEN UPDATE draw_UI, guesser_UI
8. UNDISPLAY draw_UI, guesser_UI,observer_UI
   DISPLAY assign_UI TO players

The drawer UI (Fig. 10) enables the drawing area. The guessers and observers are able to watch the drawing area but are unable to draw on it. The guessers can try words. A timer runs down until the word is found or until it reaches 00:00. This UI can be translated into a local distribution graph for the device on which the Pictionary is started. For example, a user with a computer and a mobile phone will have the following distribution scene as in Fig. 11 and 12.

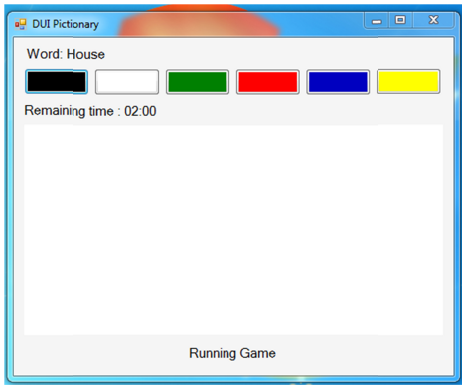

Figure 10. draw_UI enabling drawing area.

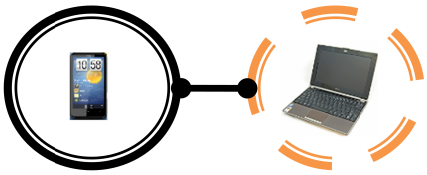

Figure 11. The cluster model of the Pictionary.

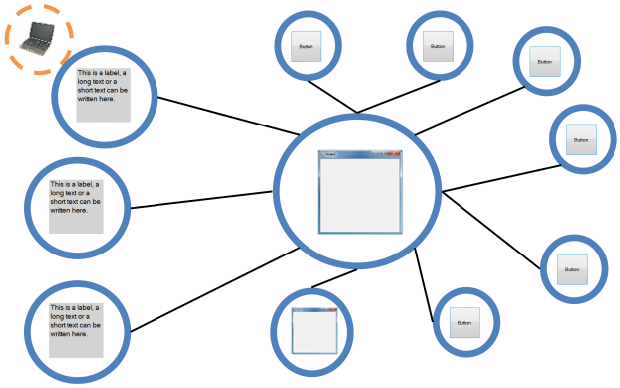

Figure 12. Associated graph of CUI models.

Game of the Goose

We also present an idea of a game that will be a combination of several games as the Game of the Goose. A basic example would be to use the Pictionary, a Minesweeper and a Snake as games to combine. See Figure 13 for the Minesweeper examples of UI. The DUI combination game increases the use of distribution. When a player reaches a case on the board, the game of the case will be loaded and automatically display to its platforms. We may also redistribute the UI through its platforms at his own ease of use.

Figure 13. Example of a Minesweeper game.

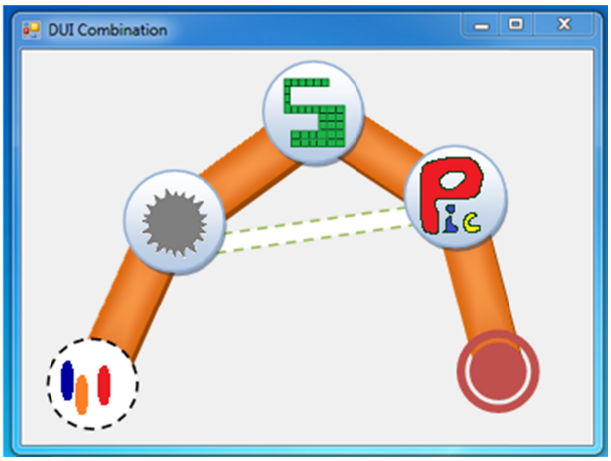

Figure 14. Example of UI for the combination game.

There is a bridge between the Minesweeper and Pictionary games. A win in the Minesweeper game will allow the player to go to the Pictionary while a loss on the Pictionary will send the player back to the Minesweeper. The example shown here is a simple 3-cases game, but the number should be higher for a real game. In the Game of the Goose, there are 63 cases. In order to demonstrate the use of the distribution, we start with a configuration of three devices. The global distribution graph is represented in Figure 15.

Two users, Lucas and Nathan, share Lucas's laptop on which they are currently running the Game of the Goose. This laptop is represented by the dashed circle laptop on the graph. Lucas has connected his mobile phone, while Nathan has connected his own laptop. The Game of the Goose is based on the toolkit and will automatically distribute the UI. Two players have been created in the game and are each linked to a platform. The game starts with the first player, Lucas, and will use Lucas' mobile phone to display the Minesweeper game. The game triggers a scenario of the distribution to distribute the UI allowing Lucas to play and preventing Nathan to access to the game. At this level, the global distribution graph has been changed and is represented in Fig. 16. The scenario used for the distribution is:

DISPLAY minesweeper_UI TO lucas ON mobile_1

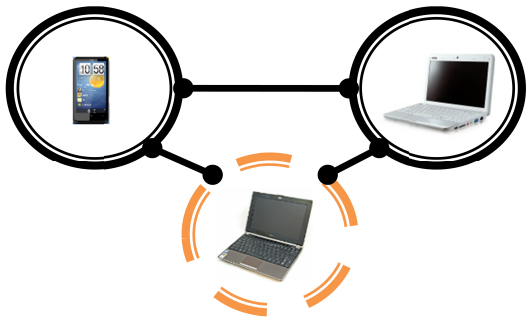

Fig. 15. Global distribution graph for the Game of the Goose.

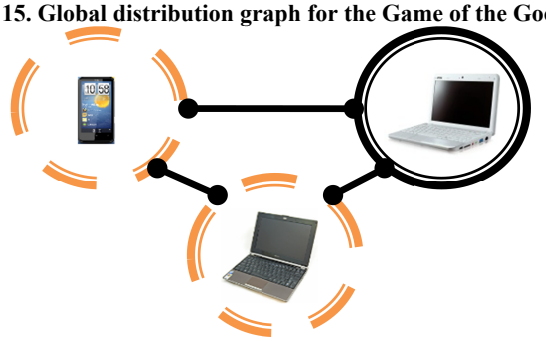

Figure 16. Global distribution graph for the Game of the Goose when Lucas is playing Minesweeper.

When Lucas has finished playing the Minesweeper game, he is now either staying at Minesweeper case if he lost the game or going to the Pictionary if he won. It Is now Nathan's turn, and the UI is now redistribute through the following scenario:

```
UNDISPLAY minesweeper_UI TO lucas ON mobile_1
DISPLAY minesweeper_UI TO nathan ON laptop_2
```

After Nathan's part, it's Lucas turn. They will both play Pictionary. As Lucas is playing, he will be the one that choose the word and that try to make Lucas guess the word. The game is now triggering a new scenario:

```
UNDISPLAY minesweeper_UI TO nathan ON laptop_2
DISPLAY pictionary_UI TO laptop_1
DISPLAY draw_UI TO nathan ON laptop_2
DISPLAY guesser_UI TO lucas ON mobile_1
```

As Nathan wants to choose the word on his laptop, he will manually ask for pictionary_UI to be displayed on his laptop with the following primitive:

```
COPY pictionary_UI TO nathan ON laptop2
```

He is now able to choose the word from pictionary_UI and then draw on the draw_UI. Lucas sees what Nathan is drawing on his mobile phone and has the ability to propose word. If Lucas finds the word, the game stops and Lucas wins. If the time is up and Lucas has not been able to find the word, Nathan wins. The game finishes when one of the player reaches the red dot. Here, Lucas has found the word and has won the Pictionary game. He then won the Game of the Goose as it was the last game before the red dot. Here was an example of a more interesting distribution application.

CONCLUSION AND FUTURE WORK

In this paper we have introduced a model-based approach for designing applications with Distributed User Interfaces. These applications enable the distribution of their UI components in both static and dynamic way. We have first introduced a distribution graph in order to model the distribution space. The distribution graphs help modeling the distribution of the UI and describing the virtual distributed environment of the application. We then introduced a distribution language to describe the distribution statements. These statements can be used in distribution scenarios for automatic and manual distribution. This latter can be through a command line interface or a Meta-UI. No tool has been developed in order to support the distribution graph representation. A toolkit supporting the creation of DUIs and the distribution operations is currently in development and will be introduced in the future. Based on this toolkit, we have exemplified the model-based approach on the example of a Pictionary as basic case-study. Then, we have introduced an example of application that is a composition of games. It exemplifies the use of distribution primitives and scenarios. There is still work to do before releasing the toolkit. We keep working on this model-based approach in order to make it compatible with several approach designing applications. A catalog of the distribution primitives will be released with a complete description of each primitive.

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 Wallonne

REFERENCES

1. Avrahami, G., Brooks, K.P., and Brown, M.H. A Two-View Approach to Constructing User Interfaces, *ACM Computer Graphics 23*, 3 (July 1989), pp. 137–146.
2. Ayatsuka, Y., Matsushita, N., and Rekimoto, J. HyperPalette: a hybrid computing environment for small computing devices. In: *Proc. of CHI'00* (The Hague, April 1-6, 2000). ACM Press, New York (2000), pp. 133–134.
3. Bandelloni, R. and Paternò, F. Migratory user interfaces able to adapt to various interaction platforms. *Int. Journal of Human-Computer Studies 60,* 5-6 (2004), pp. 621–639.
4. Beale, R. and Edmondson, W. Multiple carets, multiple screens and multi-tasking: new behaviours with multiple computers. In: *Proc. of HCI'07* (Lancaster, September 3-7, 2007). British Computer Society, Swinton (2007), pp. 55–64.
5. Berglund, E. and Bång, M. Requirements for distributed user interface in ubiquitous computing network. In: *Proc. of ACM Conf. on Mobile and Ubiquitous MultiMedia* MUM'02 (Oulu, December 11-13, 2002). ACM Press, New York (2002).
6. Bharat, K. A. and Cardelli, L. Migratory applications. In: *Proc. of UIST'95* (Pittsburgh, November 15-17, 1995). ACM Press, New York, 1995, pp. 132–142.
7. Biehl, J. T., Baker, W. T., Bailey, B. P., Tan, D. S., Inkpen, K. M., and Czerwinski, M. IMPROMPTU: a new interaction framework for supporting collaboration in multiple display environments and its field evaluation for co-located software development. In: *Proc. of CHI'08* (Florence, April 5-10,

2008). ACM Press, New York (2008), pp. 939–948.
8. Chung, G. and Dewan, P. Towards dynamic collaboration architectures. In: *Proc. of CSCW'04* (Chicago, November 6-10, 2004). ACM Press, New York (2004), pp. 1–10.
9. Coutaz, J. Meta-User Interfaces for Ambient Spaces. In: *Proc. of TAMODIA'2006* (Hasselt, October 23-24, 2006). LNCS, Vol. 4385. Springer, Berlin (2006), pp. 1–15.
10. Delivery Context Ontology (DCO), W3C, Geneva, 2010. http://www.w3.org/TR/2009/WD-dcontology-20090616/.
11. Demeure, A., Sottet, J.S., Calvary, G., Coutaz, J., Ganneau, V., and Vanderdonckt, J. The 4C Reference Model for Distributed User Interfaces. In: *Proc. of ICAS'2008* (Gosier, March 16-21, 2008), IEEE Comp. Soc., (2008), pp. 61-69.
12. Dewan, P. and Shen, H. Controlling access in multiuser interfaces. *ACM Trans. Comp.-Hum. Interact. 5, 1* (1998), 34-62.
13. Distributed Programming in Mozart–A Tutorial Introduction, chapter 3: Basic Operations and Examples, accessible at http://www.mozart-oz.org/documentation/dstutorial/node3.html#chapter.examples. Visited on Nov. 21st, 2010.
14. Econometric Modeling & Computing Corporation (EMCC). 2010. Accessible at: http://www.speakeasy.com
15. Eisenstein, J., Vanderdonckt, J., and Puerta, A. Applying model-based techniques to the development of UIs for mobile computers. In: *Proc. of IUI'01* (Santa Fe, January 14-17, 2001). ACM Press, New York (2001), pp. 69–76.
16. Grolaux, D., Van Roy, P., and Vanderdonckt, J. Migratable User Interfaces: Beyond Migratory User Interfaces. In: *Proc. of 1st IEEE-ACM Annual Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services MOBIQUITOUS'04* (Cambridge, August 22-26, 2004). ACM Press (2004), pp. 422–430.
17. Grolaux, D., Vanderdonckt, J., and Van Roy, P. Attach me, Detach me, Assemble me like You Work. In: *Proc. of INTERACT'05* (Rome, September 12-16, 2005). LNCS, Vol. 3585, Springer-Verlag, Berlin (2005), pp. 198–212.
18. Grudin, J. Partitioning digital worlds: focal and peripheral awareness in multiple monitor use. In: *Proc. of the ACM Conf. on Human Factors in Computing Systems CHI'01* (Seattle, 2001). ACM Press, New York (2001), pp. 458–465.
19. Han, R., Perret, V., and Naghshineh, M. WebSplitter: a unified XML framework for multi-device collaborative Web browsing. In: *Proc. of CSCW'00* (Philadelphia, December 2-6, 2000). ACM Press, New York (2000), pp. 221–230.
20. Hutchings, D. R., Smith, G., Meyers, B., Czerwinski, M., and Robertson, G. Display space usage and window management operation comparisons between single monitor and multiple monitor users. In: *Proc. of AVI'04* (Gallipoli, May 25-28, 2004). ACM Press, New York (2004), pp. 32–39.
21. Hutchings, H.M. and Pierce, J. S. Understanding the whethers, hows, and whys of divisible interfaces. In: *Proc. of the Working Conf. on Advanced Visual Interfaces AVI'06* (Venezia, May 23-26, 2006). ACM Press, New Y. (2006), pp. 274–277.
22. Loeser, C., Mueller, W., Berger, F., and Eikerling, H.J. Peer-to-Peer Networks for Virtual Home Environments. In: *Proc. of HICSS'03* (Big Island, January 6-9, 2003). IEEE Computer Society, Los Alamitos (2003), p. 282.
23. Lorenz, A. Research directions for the application of MVC in ambient computing environments. In: *Proc. of the 1st Int. Workshop on Pattern-Driven Engineering of interactive Computing Systems PEICS'10* (Berlin, July 20, 2010). ACM Press, New York (2010), pp. 28–31.
24. Luyten, K. and Coninx, K. Distributed User Interface Elements to support Smart Interaction Spaces. In: *Proc. of the 7th IEEE Int. Symposium on Multimedia ISM'2005* (December 12-14, 2005). IEEE Computer Society (2005), pp. 277–286.
25. Luyten, K., Vandervelpen, C., and Coninx, K. Migratable User Interface Descriptions in Component-Based Development. In: *Proc. of DSV-IS'2002* (Rostock, June 12-14, 2002). Lecture Notes in Computer Science, Vol. 2545. Springer-Verlag, London (2002), pp. 44–58.
26. Luyten, K., Van den Bergh, J., Vandervelpen, Ch., and Coninx, K. Designing distributed user interfaces for ambient intelligent environments using models and simulations. *Computers & Graphics 30*, 5 (2006), pp. 702–713.
27. Melchior, J., Grolaux, D., Vanderdonckt, J., and Van Roy, P. A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications. In: *Proc. of the 1st ACM Symposium on Engineering interactive Computing Systems EICS'09* (Pittsburgh, July 15-17, 2009). ACM, pp. 69–78.
28. Molina, J.P., Vanderdonckt, J., González, P., Fernández-Caballero, A., and Lozano, M.D. Rapid Prototyping of Distributed User Interfaces. In: *Proc. of CADUI'2006* (Bucharest, 6-8 June 2006). Springer-Verlag, Berlin (2006), pp. 151–166.
29. Myers, B. A. 2001. Using handhelds and PCs together. *Commun. ACM 44*, 11 (November 2001), pp. 34–41.
30. Newman, M. W., Izadi, S., Edwards, W. K., Sedivy, J. Z., and Smith, T.F. User interfaces when and where they are needed: an infrastructure for recombinant computing. In: *Proc. of the 15th ACM Symposium on User interface Software and Technology UIST'02* (Paris, October 27-30, 2002). ACM Press, New York (2002), pp. 171–180.
31. Qiu, X. F. and Graham, T.N. Flexible and efficient platform modeling for distributed interactive systems. In: *Proc. of the 1st ACM Symposium on Engineering interactive Computing Systems EICS'09* (Pittsburgh, July 15 - 17, 2009). ACM Press, New York (2009), pp. 29–34.
32. Sjölund, M., Larsson, A., and Berglund, E. Smartphone Views: Building Multi-Device Distributed User Interfaces In: *Proc. of MobileHCI'2004* (Glasgow, 13-16 September 2004). LNCS, Vol. 3160. Springer, Berlin (2004), pp. 507–511.
33. Tan, D.S. and Czerwinski, M. Effects of Visual Separation and Physical Discontinuities when Distributing Information across Multiple Displays. In: *Proc. of INTERACT'03* (Zurich, September 1-5, 2003). IOS Press (2003), pp. 252–260.
34. Tan, D.S., Myers, B. and Czerwinski, M. 2004 WinCuts: Manipulating Arbitrary Window Regions for More Effective User of Screen Space. In: *Proc. of CHI'2004* (Vienna, April 24-29, 2004). ACM Press, New York (2004), pp. 1525–1528.
35. Vanderdonckt, J. Distributed User Interfaces: How to Distribute User Interface Elements across Users, Platforms, and Environments. In: *Proc. of XIth Congreso Internacional de Interacción Persona-Ordenador Interacción'2010* (Valencia, 7-10 September 2010). AIPO, Valencia, 2010, pp. 3-14.
36. Vandervelpen, Ch., Vanderhulst, G., Luyten, K., and Coninx, K. 2005. Light-Weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments. In: *Proc. of the 5th Int. Conf. on Web Engineering ICWE'2005* (Sydney, July 27-29, 2005). Springer, Berlin (2005), pp. 197–202.
37. Van Roy, P. and Haridi, S. Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.
38. Xiaojun, B. and Balakrishnan, R. Comparing usage of a large high-resolution display to single or dual desktop displays for daily work. In: *Proc. of the 27th Int. Conf. on Human factors in Computing Systems CHI'09* (Boston, April 4-9, 2009). ACM Press, New York (2004), pp. 1005–1014.
39. Yanagida, T. and Nonaka, H. 2008. Architecture for Migratory Adaptive User Interfaces. In: *Proc. of the 8th IEEE Int. Conf. on Computer and Information Technology CIT'2008* (Sidney, July 8-11, 2008). IEEE (2008), pp. 450–455.