A Toolkit for Peer-to-Peer Distributed User Interfaces: Concepts, Implementation, and Applications

Jérémie Melchior[1], Donatien Grolaux[1,2], Jean Vanderdonckt[1], Peter Van Roy[2]
Université catholique de Louvain, – B-1348 Louvain-la-Neuve (Belgium)
[1]Louvain School of Management, Place des Doyens, 1
[2]Dept. of Computing Science and Engineering, Place Sainte Barbe, 2
nediar@gmail.com, {jeremie.melchior, jean.vanderdonckt, peter.vanroy}@uclouvain.be

ABSTRACT
In this paper we present a software toolkit for deploying peer-to-peer distributed graphical user interfaces across four dimensions: multiple displays, multiple platforms, multiple operating systems, and multiple users, either independently or concurrently. This toolkit is based on the concept of multi-purpose proxy connected to one or many rendering engines in order to render a graphical user interface in part or whole for any user, any operating system (Linux, Mac OS X and Windows XP or higher), any computing platform (ranging from a pocket PC to a wall screen), and/or any display (ranging from private to public displays). This toolkit is a genuine peer-to-peer solution in that no computing platform is used for a server or for a client: any user interface can be distributed across users, systems, and platforms independently of their location, system constraints, and platform constraints. After defining the toolkit concepts, its implementation is described, motivated, and exemplified on two non-form based user interfaces: a distributed office automation and a distributed interactive game.

Categories and Subject Descriptors
C2.4 [**Distributed systems**]: Distributed applications. D1.3 [**Concurrent Programming**]: Distributed programming. D2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; user interfaces*. D2.m [**Software Engineering**]: Miscellaneous – *Rapid Prototyping; reusable software*. D4.7 [**Organization and Design**]: Distributed systems. H.1.2 [**Information Systems**]: Models and Principles – *User/Machine Systems*. H5.2 [**Information interfaces and presentation**]: User Interfaces – *Prototyping; user-centered design; user interface management systems (UIMS)*. I.6.5 [**Model Development**]: modeling methodologies.

General Terms
Design, Experimentation, Human Factors, Verification.

Keywords: Distributed User Interfaces, Multi-Device Environments, Multi-platform user interfaces, Multi-user user interfaces, Peer-to-peer, User Interface Toolkit, Ubiquitous computing.

1. INTRODUCTION
The division of labor in corporate environments requires more and more to allocate tasks to users in a flexible, dynamic, and opportunistic way. For instance, a task that can no longer be ensured by a worker is delegated to another one or offered to single / multiple resources in order to be achieved. Interactive tasks are frequently reallocated or (re)distributed across workers in an organization and the User Interfaces (UIs) should support these interactive tasks. For instance, a particular worker may want to get some advice from a colleague for solving a problem. For this purpose, she can share some part of the information by redistributing parts or whole of her UI, here by duplicating the UI to another user using another computing platform and located in another environment or context of use.

Workers may distribute UIs for several reasons beyond this distribution of task allocations: in order to follow task allocation, to exchange information with co-workers, to balance private and public information [15], to partition the working space into different parts [12] and several displays [22]. They all want to distribute their UI in order to match requirements of dynamically distributed tasks and to keep the same usability quality in the distributed UIs as they had before the distribution. The UIs of these applications are generally unable to accommodate such changes, thus forcing end users to switch from one application to another or to rely on workflow management systems (if any) to reach their goals. In general, the following situations may arise:

- *Multi-monitor usage*: a single user using a single computing platform may want to distribute her UI across various monitors connected to the same platform [12,14]. For instance, a dual display if the graphic card allows it or an external monitor via an external port.
- *Multi-device usage*: a single user may use several different devices together, whether they are running the same operating system or not [15]. For instance, a user may control a music player running on a media center using a remote control running on a handheld device.
- *Multi-platform usage*: a single user may user heterogeneous computing platforms, perhaps running different operating systems [21]. Note that a multi-device usage implies a multi-platform usage (since there are different machines) but the reciprocal does not hold: a user could use several computers (hence, multi-platform) that are similar (hence, no multi-device).
- *Multi-display usage*: we hereby define multi-display as a combination of multi-monitor and multi-device usages [22]. A single user may distribute a UI across multiple monitors and devices simultaneously.
- *Multi-user*: it represents an extension of the previous usages to multiple users concurrently [5]. In this case, one or many users may want to distribute parts or whole of their UI across several monitors, devices, platforms, or displays. For instance, in

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EICS'09, July 15–17, 2009, Pittsburgh, Pennsylvania, USA.
Copyright 2009 ACM 978-1-60558-600-7/09/07…$5.00

a control room setup, users may want to direct portions of a UI to other displays of others users depending on the context of use. When a multi-user interface is of concern, it is also typically used for supporting tasks that are allocated or deallocated from one user to another one, such as in task delegation, task suspension and resuming.

A *Distributed User Interface* (DUI) consists of a UI having the ability to distribute parts or whole of its components across multiple monitors, devices, platforms, displays, and/ or users. Hence, a DUI should support any of the aforementioned usages, the multi-monitor being the minimal.

This paper presents a toolkit with the following properties. Given n application processes and m display processes, the toolkit allows the GUI of each application to be partitioned arbitrarily and dynamically over the m display processes. A single display process can therefore combine parts of the GUI of each application, in one or several windows with no restriction. No other dependency exists beyond those implied by the n+m processes, i.e., if an application process crashes, its GUI disappears and if a display process crashes, the GUI parts that it hosts 'return' to the application process and can be migrated to another display process. Each application has its own point of failure which means that if an application process crashes, other applications still work.

During the past two decades, a lot of work has been dedicated to addressing some of the aforementioned usages at a time, such as multi-device or multi-platform only. With the advent of all these usages, the time has come to enable designers and developers to engineer their interactive applications in a way that they do not need to take care about the functions required to support these usages.

In other words, the user interface of such applications should become completely agnostic, independent from any underlying technology that would allow these usages. Until now, many works addressed these usages explicitly, but in a way that forces designers and developers to think and develop user interfaces in a way that is constrained by distribution. For instance, the underlying software architecture may influence the way the user interface is programmed if it should be migratory [3,26] or multi-user [5].

In this paper, we relax this important constraint by offering to designers and developers a toolkit that would enable them to design and develop user interfaces that support all the aforementioned usages. For this purpose, the remainder of this paper is structured as follows: Section 2 reports on some significant pieces of related work in order to characterize a brief, yet accurate, state of the art in the domain of distributed user interfaces. Section 3 summarizes the benefits brought by the toolkit. Section 4 immediately shows two case studies of interactive applications exhibiting user interfaces covering the above usages that are hard to develop otherwise. It then introduces, describes, and motivates the software architecture of the underlying toolkit that was used for those two case studies and finished with some final examples. It also discusses some selected aspects and properties of this toolkit. Section 5 concludes the paper by presenting some future avenues of this work.

2. RELATED WORK

In this section, we compare the advantages and disadvantages of the major related work in order to identify the specific aspects of our toolkit.

Probably the first DUI ever was developed as a system that distributed a UI over many workstations connected to the same network and running the same operating system [3] thanks a to a connector mechanism. In [4], a program is dynamically changing between centralized, replicated and a hybrid collaboration architecture. There is a notion of masters using a replica of the program and slaves using one of the replicas provided by a master in a centralized architecture. It allows mobile devices with lack of computing power to avoid running the program when it can communicate with a more powerful device.

In [13], a web page is split in partial pages which will be replicated to all the users. The framework supports multi-device and multi-user Web browsing where clients connect to a server which delivers the page. A proxy split the pages in respect to the device and user constraints. Each page is in a XML file with specific tags to configure how the Web page will be split among the different users and devices.

In Luyten & Coninx [17], it is shown how an interactive system can be distributed among several peer devices. Their approach relies on the fact that nowadays most computing resources are network-enabled and publish their device profile like in UAProf or CC/PP. It raises the opportunity for supporting collaborative tasks with the same user interface with little or no extra effort from the user interface designer.

In [2,24], a part or whole of a DUI can be migrated from one platform to another at run-time. The underlying architecture is a client-server architecture that maintains in a central position the internal state of the DUI.

In this paper, the toolkit that will be presented is significantly different from this previous work in that it provides a unique combination of the following features:

- Any DUI developed in the toolkit may benefit transparently from facilities provided to support any of the aforementioned usages.
- The DUI is not restricted to form-based applications as it is possible to distribute any graphical UI, such as from a game, a spreadsheet, a graphic application. Such programs contain native widgets.
- The DUI is not restricted to web applications. The only condition to distribute a UI is to have the platforms connected via a LAN or a wireless connection, but the applications are not necessarily web applications in markup languages. There is no such language restriction like having a UI in HTML or another markup language.
- The toolkit relies on a genuine peer-to-peer architecture in the sense that there is no client and no server: every platform can send and receive any part of a DUI depending on its distribution rights. No single platform maintains the DUI internal state.
- The granularity of distribution can range from the application level to the widget level: an entire application can be distributed across platforms for instance, but also the different components of any widget. For instance, even a radio button, consisting of a circle to be checked and a label can be split across platforms. Even the label could be distributed, although it does not make sense in this case.

3. SOLUTION BENEFITS

The toolkit does not rely on a client-server architecture which in the WebSplitter[13] would allow each device to create a Web page

and to share it with others. There is no need for a server to store the Web page and provide it to the proxy. Each peer may create a user interface and share it through the network. The exchange of data is values passed through a message passing mechanism; there is no static representation of the application in files. The toolkit combines the ability to distribute a part or whole of a user interface among devices without needing to completely share the application. It is possible to design a lot of various applications at a finer granularity than in [3,4,13]. Here is how the dimensions are possible in the toolkit:

- *Multiple displays*: the system may be a single device connected to multiple displays or multiple devices. For a single device with many displays, many rendering engines on the same device might be used. For multiple devices, each device will create at least one rendering engine.
- *Multiple users*: the user interface can be distributed in part of whole for each user independently of the situation.
- *Multiple operating systems*: the toolkit is developed in the Mozart environment which supports a lot of different operating systems.
- *Multiple platforms*: it is supported by combining the multiple operating systems dimension and the distribution of parts of the user interface. The smallest form factors which are not able to render the whole user interface might just get what they need and are able to display.

In sort, any computing platform, regardless its operating system (i.e., Linux, MacOs or Windows), can trigger a distribution of a user interface to other platforms provided that they are connected together through any local or wireless network. Each computing platform can be used by a single user or by multiple users. The distribution of the UI is not governed by a single machine since any portion of a UI can be, for instance, forwarded to another one, and vice versa.

4. DISTRIBUTED USER INTERFACES

4.1 Case Studies

Two case studies have been chosen to evaluate the power of the toolkit. The evaluation of a toolkit is never easy because there are always a lot of different and complex applications and it is not possible to test every single feature and his behaviors. The choice of case studies is thus quite important and difficult.

The *first case study* is a distributed office suite. The goal is to show that even a complex application with many toolbars and buttons, such as those found in office suites, can be distributed over multiple devices and many users. The interface can be decomposed in a lot of components. Going further in the decomposition of the interface, the work space can also be divided into a lot of different migratable regions. Distributing components of any commercially-available application belonging to an office suite cannot be achieved today.

The *second case study* is a distributed Pictionary. This is a multiplayer game where each player has his own role. To prevent players from cheating, the toolkit provides some distinctions between the users. Players trying to guess the word cannot see it and the player who has to describe the word by drawings should see the word and draw in an area. The drawing area then is seen by the other players but none of them can edit it.

Pictionary is naturally distributed because it needs at least a drawer, a player trying to guess the word the other is trying to draw. This game is task-driven distributed, the choice of by whom and where the task is realized depends on the task itself. A player has to pick a word and then another have to guess that word. These two tasks have to be realized by two different users, otherwise the player will know the word she has to guess. These two tasks should also happen at different places to prevent the players seeing the chosen word. The game is distributed across players.

Contrarily to Pictionary, the distributed office suite is not naturally distributed. Someone may write a text or draw a picture but nobody is able to write in the text at the same time or to draw on the same picture. One of the objectives of the first case study is to distribute an application that is not naturally distributed. Many users will be able to draw on the same place, to write on the same text at the same time and to work on the same spreadsheet. Another objective is to be able to distribute atomic elements of the user interface. Buttons on the drawing toolbar shall be detachable and distributable. A paragraph or a line of the text shall also be distributable without the need to distribute the whole text. A task like writing a text may be realized by more than one user, which is different than usual office suites.

4.1.1 The Distributed Office Case Study

The office suite is a bundle of applications running separately in order to write text, to draw something, to make a presentation, and to do other typical office tasks.

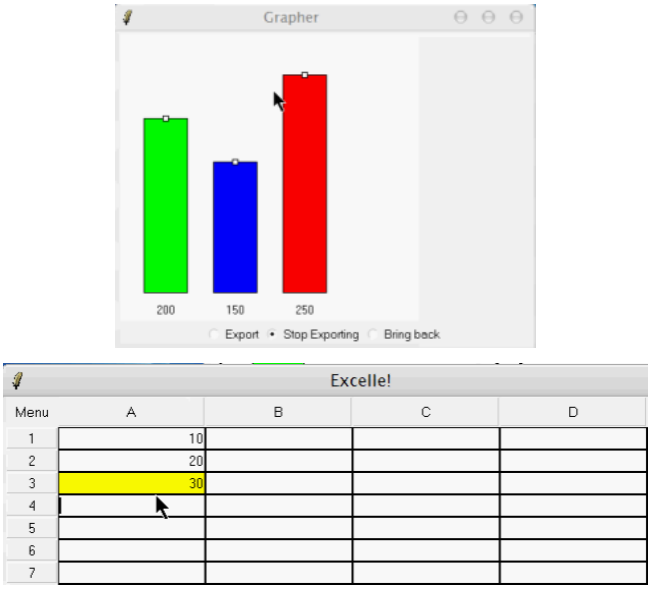

Figure 1. A graphing and a spreadsheet application.

Figure 1 reproduces a screenshot of two typical office applications: a graphing that shows a histogram with 3 columns. The value of each column can be edited by height and a spreadsheet. Any component of either application can be marked for export and exported to another display, device, platform or user. In particular, a whole UI like the one of graphing application itself is entirely migratable from one platform to another. In the second case, every cell is migratable from one platform to another. The value of a cell, even if it is the result of a formula, will remain consistent with the formula, no matter where the cell is displayed.

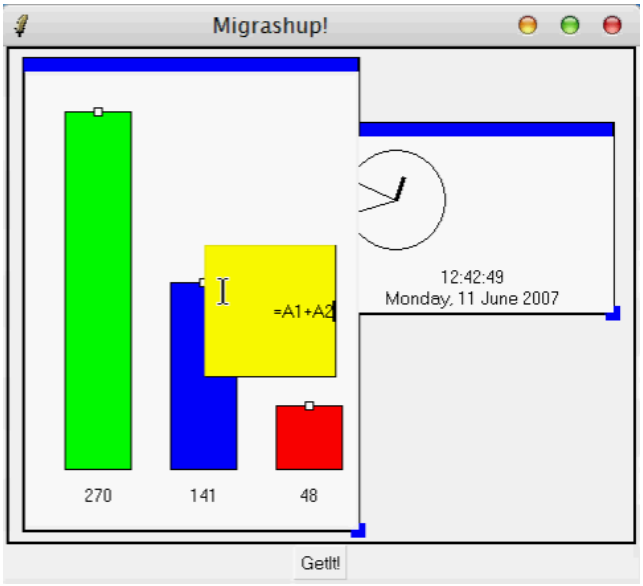

Figure 2. Example of migration realized from the two applications.

The result of a possible migration of components exported from both applications is illustrated in Figure 2. For each application, parts or whole can be marked for export, exported to another platform where it continues to run its own life, and re-imported back from where it was initially exported. This migration allows a user to enter some data in the spreadsheet while another user is looking at the result. The other user may adapt the values of the graphing thanks to the values of the cells he gets from the spreadsheet. The work is concurrently achieved and many users are working together no matter where they are.

Applications with migratable abilities allow many users working together on the same application and on the same document. The sequential work is thus converted to a concurrent multi-task system. A full video of a typical session demonstrating this case study is provided attached on the PCS system for this submission.

4.1.2 The Pictionary Case Study

The Pictionary is a multi-player game with a board. Players are separated into teams that will have to guess words. Teams are represented by their position on the board. They win the game when they reach the last square of the board. In each team, a player has to make his team guess a word by drawing some clues or the word itself but without talking and without writing it. Here, the game is a multi-user application with two teams. Both teams have to play alternatively. The needs of this case study are some devices able to do the tasks. The first task is the selection of a word on a computer by a team, let's call it Team 2. This computer may be a PDA, a laptop or a desktop. The other team, let's call it Team 1, has to guess the word selected by Team 2. A member of Team 1, denoted Player, has to draw something to help his teammates find the word. This task has to be realized on a PDA to allow drawing like on a paper sheet. In order to see what Player is drawing, another device is set up to display the drawing. A computer with a big screen or a video projector can be used to realize this task. An example of architecture able to run the application is illustrated in Figure 3.

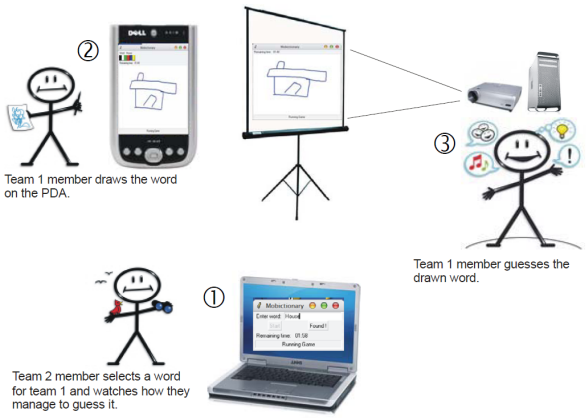

Figure 3. Example of architecture for the Pictionary.

To start the game, the application must be run on the three devices. Once the connection is established, Team 2 goes to a computer to enter a word. Once entered, a countdown starts to prevent Team 1 using too much time to find the word. The PDA is given to Player and the word appears on the screen. He may now draw whatever he wants. The video projector displays the drawing for everyone. These different steps are reproduced in Figure 4. In this case, the different screen shots are taken from the different devices running the same operating system, but nothing would prevent this application to run between devices running different operating systems.

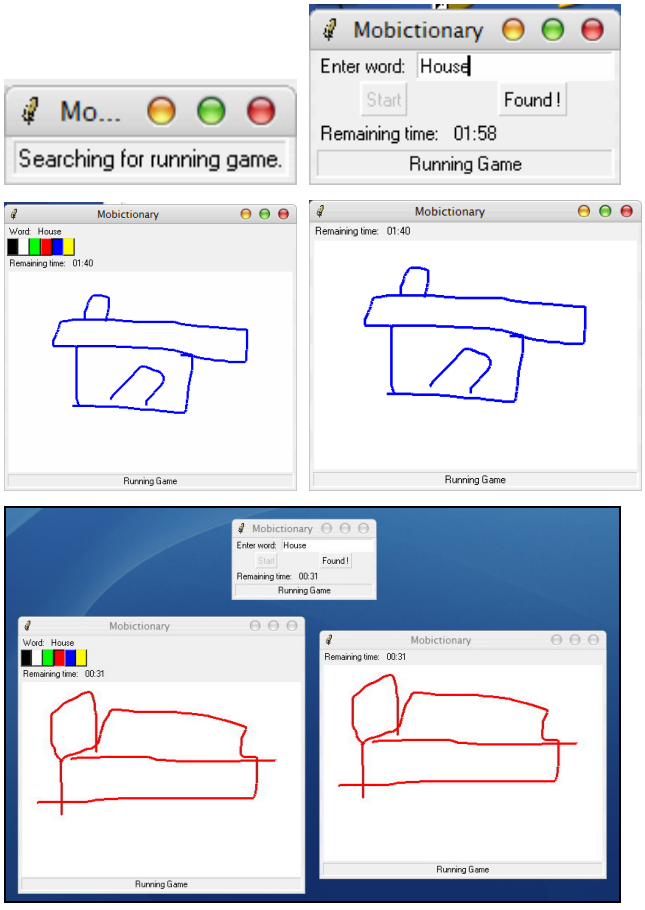

Figure 4. Screenshots of the Pictionary.

Indeed, the toolkit used for this purpose allows deploying DUI on top of three operating system: Linux Ubuntu distribution, Mac OS X and Windows XP/Vista any version.

The toolkit allows designing and developing the DUI of this application and supports the multi-user aspects of the game. To prevent cheating, some widgets are created and migrated to the PDA as well as the toolbar which allows drawing. The application can also be run on a single computer in multiple windows but we recommend using at least three devices: one to enter the word, one for the drawings and one to find the word.

Let us design this application with our toolkit, using the simple multi-user aspects of the toolkit. Even if three different devices are used in a distributed fashion, we must think of this application as a single process application, with transparently distributed user interfaces. The widgets used are:

- A text field for entering the text (Laptop)
- A label for displaying this text (PDA)
- A start button for accepting the text input and starting the countdown (Laptop)
- A clock (PDA, Laptop, and PC)
- A free drawing area (PC and PDA)
- A toolbar for selecting the drawing tools (PDA)
- A "Win" button to click if the word is guessed on time (Laptop)

We end up with seven different components, some of them present only on one device, others at different devices simultaneously. We use the proxy-renderer relationship: the underlying principle is that the proxy serves as the reference for the state of the widget while the renderer follows the instructions of the proxy for updating its incarnation. This principle allows the existence of several renderers; each of them following the same instructions. Basically they all act as mirror views of the same proxy. Note that this is against the principle of not disrupting the stationary behavior as we introduce multi-user capabilities. New complexity is introduced because of concurrency and coherency problems. The toolkit itself provides a very basic way of dealing with this complexity: each widget can be configured so as to have at most one renderer at a time (the default), or to let an arbitrary number of renderers be connected at the same time. For the Pictionary application, we have the following functionalities:

- The application can be run from any device, including the PC, the laptop, the PC or even another computer. For our demonstration, we use the laptop.
- The text field, Start and Win buttons are created and migrated to the laptop
- The clock is created, configured to support multiple renderers and displayed on all the devices
- The label and the toolbar are created and migrated to the PDA.
- The canvas is created, configured to support multiple-renderers and displayed on the PDA and the PC.

When the start button is clicked, the content of the text entry is placed into the label, and a countdown thread is created: each second the clock is updated to reflect the remaining time. The toolbar chooses the active drawing tool, while the clicks on the drawing area issue commands to apply it at that place. And that's it; we have a functional simple multi-player game! Note that the three processes find each other using the Discovery module of Mozart [6], which uses a broadcasting message on a LAN to find providers. The Pictionary is inspired by the HyperPalette [1], but in a distributed way that can be tailored at run-time.

4.2 SOFTWARE TOOLKIT FOR PEER-TO-PEER DUIS

4.2.1 Software architecture

Each application using the toolkit relies on a three-layer model as in Figure 5. The Application layer describes the different services offered, the toolkit for migration and adaptation and the distribution layer for peer-to-peer network. At the top, the Application layer is the part developed for the application itself. It differs between applications depending on their needs. They support migratable and adaptable user interfaces. This part is independent from the toolkit and the distribution layer. The application has a standard graphical user interface. The middle layer is the toolkit supporting the different features for migration and adaptation. It extends Tcl/Tk which is a toolkit for graphical user interfaces supporting several platforms. An extension Ext is added to this toolkit to provide the needed features. The lowest layer is the base for the toolkit. Mozart [6] implements the Oz programming language [25]. It supports several paradigms and distributed applications [20]. It relies on a distribution layer which relies on TCP/IP protocol.

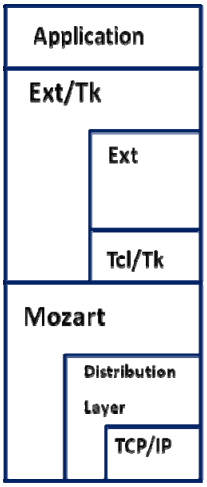

Figure 5. Software architecture of applications using the toolkit.

4.2.2 Granularity of migration & adaptation

A running application could have its UI migrated and/or adapted at different levels of granularity:

1. Whole screen containing the UI of the application (which may also contain the UI of other running applications).
2. Whole UI of the application, typically contained in a single window.
3. Subset of widgets of the application:
   a. Limited to a single widget.
   b. Limited to widgets that respects some placement constraints (for example respecting a rectangular shape).
   c. Any arbitrary selection of widgets.
4. Arbitrary pixel area.

Not all these levels are interesting for our purpose. In level 1, we lack information regarding the remaining of the screen which makes virtually impossible to provide interesting adaptation. In level 4, the arbitrary nature of the area also makes it virtually impossible to provide interesting adaptation. Level 2 is a particular case of level 3, where the whole UI of the application is used instead of a particular subset of it.

Consequently **Ext/Tk provides migration and adaptation support at the widget level** [11]. We want a maximum of flexibility: any widget can be migrated to any platform at any time. Two widgets from the same running application can be migrated to the same platform, or to two different ones. As the migration is independent for each widget, covering 3a is enough to also cover 3b and 3c, by executing several migrations at the same time.

4.2.3 Orthogonal migration & adaptation

To be useful, a graphical toolkit with migration and adaptation support must still offer a functionality equivalent to a graphical toolkit with no such support. In other words the migration and adaptation are new functionality on top of the pure graphical toolkit functionality. We argue that this new functionality is important enough to be isolated from the pure toolkit functionality. Ext/Tk is consequently designed to provide the migration and adaptation functionality orthogonally to the pure graphical one

1. The migration functionality is provided as a capability of the widget. This capability is a value which can be passed along freely to another process, on another computer: the migration is a distributed operation between different computers connected over the Internet (local migrations on the same computer/process are of course supported). Once a migration capability has been passed to another process, it can be used to trigger the migration of the widget, like a PULL mechanism. To achieve this, the capability serves two purposes: 1) it contains the authority to migrate the widget, and 2) it is a reference to the home site of the widget over the Internet, like a URL for a web page. Because of 2), we often call the migration capability of the widget the reference of the widget. It is the responsibility of the application to pass the capabilities to interested parties: it has the complete control on who receives them. However it does not have the control on when these capabilities are used by the remote peers, i.e. when the migration really occurs. Consequently the application should be as impermeable to the migration process as possible. The only observable effect is a temporary blocking of the threads interacting with the migrated UI. For this reason we say that the migration is transparent to the application. Note that the application can register its interest for migration events if it wants to be notified of the process. Note also that the application has a direct access to the capabilities of the widgets it has created, and consequently can use them to migrate the widget back into its original place.

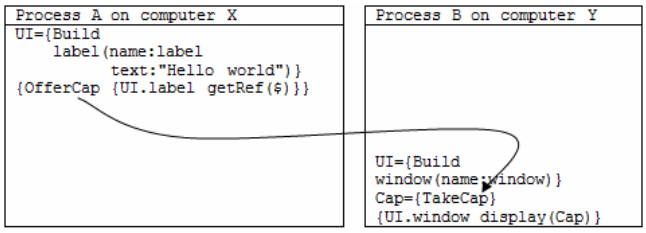

Figure 6. Migration from one platform to another.

This example illustrates the computer X offering the migration capability of a label widget, and the computer Y creating a window, getting the capability, and using it to migrate the label into its local window. Note that the `OfferCap` and `TakeCap` functions are not specified here and can be implemented in numerous different ways. This example assumes that they are able to get in touch with each other, and then exchange the piece of information given to `OfferCap`. A possible implementation is to have a shared file between the processes. Another possible implementation is to rely on emails: `OfferCap` sends an email with the capability attached to it to a mail box that is then read by `TakeCap`. Still another possible implementation is to use a DHT (distributed hash table) based P2P (peer to peer) network, and use a shared name between the two processes to place the value into the network, and obtain it back. And many more implementations are still possible.

2. The adaptation of the widgets consists in changing its representation (presentation and/or interaction) while keeping a useful level of usability. In that sense, the simple reconfiguration of a visual parameter of a widget like its background color is already an adaptation of that widget. Ext/Tk pushes this view forward by introducing a special adaptation parameter to every widget. When this special configuration parameter is changed, it is the whole way the widget is displayed that is changed. Once again, this process is impermeable to the application; the only observable effect is a temporary blocking of the threads interacting with the adapted UI. For this reason we say that the adaptation is *transparent* to the application. Because of this transparency, the application is independent of the representation currently used for a particular widget. For example the target device of a migration could provide its own representation of the widget, adapting it on the fly to its own specifics. Figure 7 illustrates a selector widget that supports different representations. Switching between these representations is achieved by calling the `setContext` method. It is up to the application to define why and when the widget should change the representation. The representation is just changed at different points during the execution unknown to the application.

```
UI={Build window(selector(name:selector
                          text:"Car Model"
                          items:["Ford" "Peugeot" "Renault"]))}

{UI.selector setContext(list)}

...

{UI.selector setContext(default)}
```

Figure 7. Migration from one platform to another.

4.2.4 Distributed structure of a widget

Desktop applications are often centralized applications running the functional core and the UI inside a single process of a computer. Some of them have a distributed functional core (voice over IP applications for example), but that is not what interest us in this work. Once we let parts of the UI migrate from site to site, several devices become involved in the running of the application, and we also shift from a centralized environment into a distributed one. The way Ext/Tk introduces distribution is motivated by two choices:

- Any widget of a running application can be migrated at any time (transparent distribution). Consequently Ext/Tk widgets are distributed entities. At any time they may be situated at the application's process, a remote process, or even nowhere if they are not currently displayed. Later we will see that it is also possible to have several renderers connected to a single proxy, replicating the UI of the widget at several places simultaneously.
- As for the functional core of the application, Ext/Tk does not dictate if it should be distributed or stationary, nor does it offer any support for distribution.

Ext/Tk provides specific distribution support for all widgets, allowing them to dynamically migrate from one site to another. But the widgets are also used by the functional core of the application that interacts with the UI, so part of them should behave in a stationary way. The distribution scheme of widgets is composed of:

- A part that is stationary to the process that created the widget. That part is returned to the functional core of the application so that it can interact with the widget. This part is called the proxy of the widget.
- A part that is distributed, and runs on the site actually displaying the widget. That part is the one the user can interact with. This part is called the renderer of the widget.

In fig. 8, three sites are running on three different computers. Site A creates two widgets that are migrated into Site B. Site C creates one widget that is also migrated into Site B. Each gray area covers a widget in its distributed execution. The proxies stay at the site that created them forever. However the renderers are running at the site the widgets are migrated to. The proxies and the renderers are connected together over the Internet, so as to be synchronized.

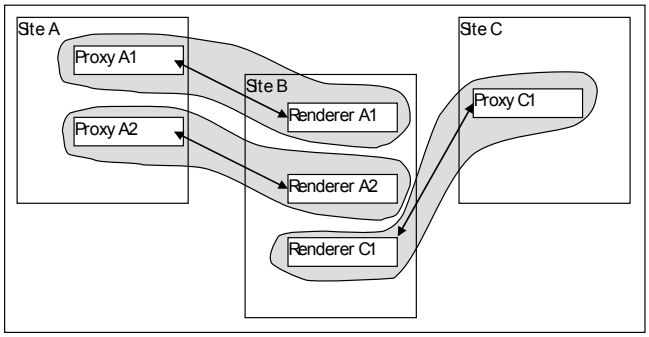

Figure 8. Distributed architecture example.

4.2.5 Runtime architecture

At runtime, each widget is split in two parts: the stationary part that stays at the creator site (the proxy), and the migratable part that is run at a remote site to actually display the widget (the renderer). A notable exception is the top level window widget: the migratable part stays at the creator site; it is created immediately along with the proxy and cannot migrate away. The renderer part of a widget needs a window to be displayed inside, so it can only run at a site were a window proxy is running. Note that the content of a window is a separate widget that can be migrated away. In other words, top level windows provide the physical hook where widgets can be displayed. Also Note that there is no dependency on an external server for this architecture to work. Widget proxies act as servers for their renderers. This is based on the distribution layer of Mozart.

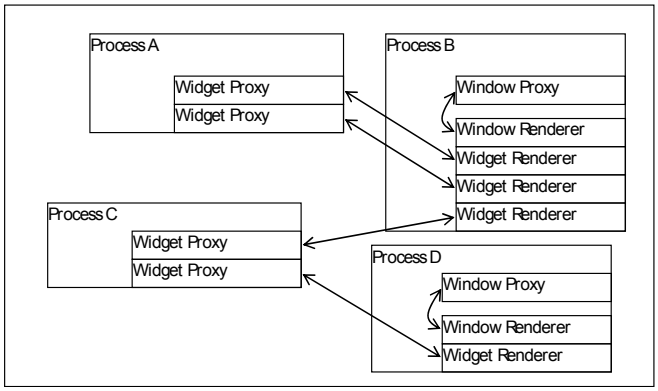

Figure 9. Overview of the runtime architecture.

4.2.6 Trajectory of a universal reference

The universal reference is a capability the creator of the widget can give to a remote site. Typically, an intermediate discovery service allows the sites to exchange these values. Fig. 10 is a typical scenario (dashed arrows are actual connections, plain arrows are the trajectory of the universal reference):

1. Process A running on computer X creates a widget and asks for its migration capability.
2. Process A stores this capability at the discovery service.
3. Process B running on computer Y asks the discovery service for the capability of the widget it wants to display.
4. Process B receives the answer
5. Process B passes it to the proxy of a container widget, here a window.
6. The proxy forwards the capability to its renderer.
7. And lastly the renderer opens a connection with the proxy corresponding to the capability. In section Migration protocol we show how this connection is used for creating a new renderer for this proxy.

Figure 11 displays a more complex scenario where the process B migrates the widget inside a container that is currently displayed at the process C. The migration capability follows the same route as in Figure 10, except that the container proxy forwards the capability to its renderer at process C and not locally anymore.

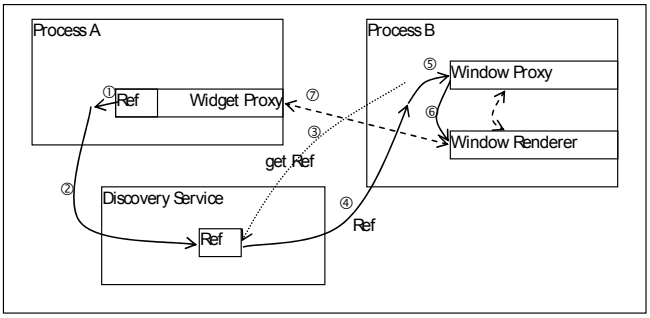

Figure 10. Universal reference trajectory.

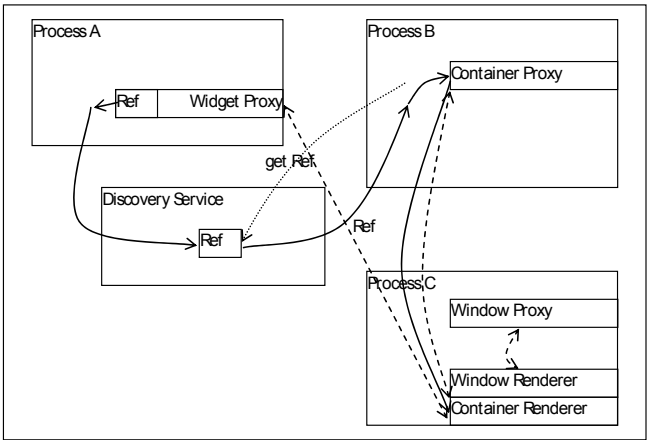

Figure 11. Complex trajectory.

4.2.7 Migration protocol

The migration protocol is a negotiation between the proxy of the receiving container (PC), the proxy of the migrated widget (PM), the renderer of the container (RC) and the renderer of the migrated widget (RM). First, the migration capability of PM has to be given

to PC somehow ①. The migration starts at PC, by using the `importHere` method of its manager using the reference given by PM (the second PI parameter is further placement instructions for example the row/column coordinates of a table container). This method stores this new child; stored children are automatically given to RC ② (either at the child's creation or at the RC creation). RC connects to PM using the reference contained in the capability ③, and in returns PM sends the class definition of the widget renderer ④. RC creates an environment and then asks Ext/Tk to create RM using the class definition just received. If RC fails to create RM (due to PM not responding, or an error while creating RM), RC tells PC to drop this particular child.

In order to create RM, Ext/Tk first creates its manager, connects back to PM ⑤, gets the actual state of all stores ⑥, and then creates the RM object with the manager as parameter ⑦. The initialization of RM should create the actual widget, and update its state according to the current content of the store ⑧ (parameters & event bindings). Once initialized, Ext/Tk automatically calls the methods of RM according to the updates of the store ⑨. If the migrated widget is itself a container, the information necessary to restore its content is in the store it receives from PM, and RM reacts to it like RC after step ②. As a result its content is also migrated along.

Negotiation phase. The step ③ of the protocol above asks a class definition for the renderer and is returned the one currently selected by PM. Indeed there can be different renderers possible for this widget, and the process running PM has selected one of them in particular (using the `setContext` method). However we can extend this protocol further by adding a negotiation phase where RC uses the knowledge of its own available resources (keyboard/mouse presence, screen size...) to hint PM so that it is able to override the current selection for the renderer with another one that is more fit to the device.

The scheme would require:

- A model for describing the platform running the UI.
- Introspection capabilities for renderers determining their level of compatibility with specific platforms.

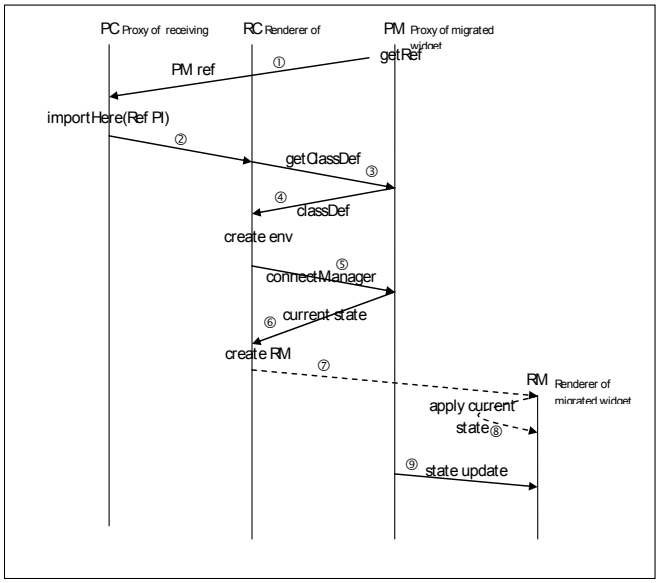

Figure 12. Migration protocol.

Another option is for RC to use its own renderer definition, ignoring the one sent by PM. This may result in an incorrect renderer that is unable to behave correctly with its proxy, however this would open up the possibility of having a target device that adapts the UIs it receives even if the process running those UIs do not know how to adapt them.

Fault tolerance. Network failures can happen at any time, between any of the sites:

- Between PC and PM. There is no direct connection between these two sites: the capability of PM can be brought to PC by a third site.
- Between PC and RC. If message ② cannot be sent be cause of a network failure or because there is currently no RC, then the migration cannot be executed. Nevertheless, the migration instruction is now part of the store of the widget. When a new RC comes in, it will then proceed with the migration of PM. As a result, there may be an arbitrary time between the application command to migrate a widget, and when this command is really executed. If message ② was sent, and there is a network failure between PC and RC then RC eventually disappears. This can happen while the migration protocol is still running, or afterwards. In all cases, the disappearance of RC will result in a disconnection with either PM or with RM. In both situations the migration of RM is cancelled, and it is destroyed if it exists.
- Between RC and PM. The only time this network connection matters is between messages ③ and ⑥. If there is a network failure there, then the migration of PM is aborted. Also RC removes PM from the migration store shared with PC, so that PM is no more considered as a contained widget of PC.
- Between PM and RM. This is the same as between PC and RC.

4.2.8 Final examples

With this toolkit, all the widgets automatically have a migration capability. This capability is controlled by the universal reference of the widget. This universal reference is a simple text string encoding the information needed to find the widget on the Internet. As long as the widget exists, this reference implements the migration capability of the widget. Typically, passing a reference from an application A to an application B is achieved by a discovery service. This service can be implemented in many different ways:

- By human beings, dictating the reference over the phone.
- By email, sending the reference inside an email.
- By using an Internet server, where A registers the reference and B gets it back. This server can be a Web server, an FTP server, or the simple socket server provided by Ext/Tk itself.
- By broadcasting messages over a LAN, allowing B to find A and get the reference. This can be implemented by the Discovery module of Mozart.
- By registering to a peer to peer network and using its functionality to get the reference. This can be implemented by the P2PS module for Mozart.

Figure 13 graphically depicts an example where every DUI component can be distributed: the clock, the buttons, the labels, the calendar, the agenda or even any entry of the calendar and the agenda. Figure 14 represents a case where the clock has been distributed.

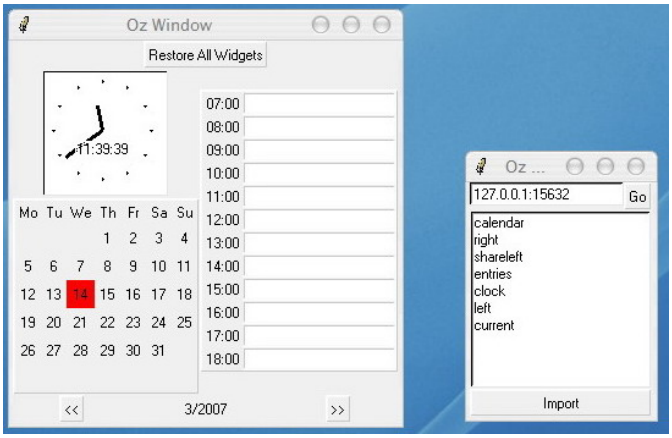

Figure 13. DUI before distribution.

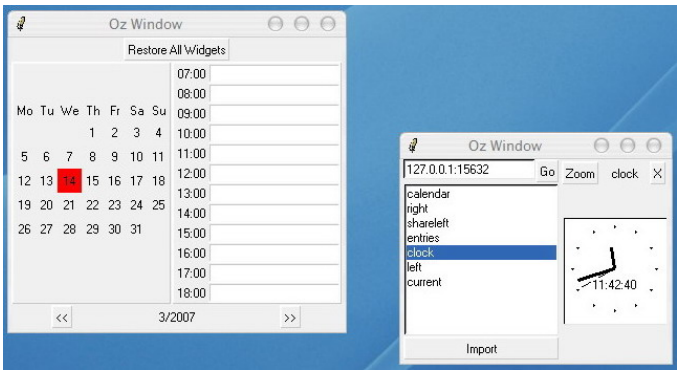

Figure 14. DUI after distribution.

Figures 15 to 17 present another example of distributing portions of a graphical user interface, in this case, a vectorial drawing application. This application is again non-form based and is considered hard to distribute due its tight synchronization between the drawing operations and its effects. Figure 15 presents a screen shot of this application before distribution.

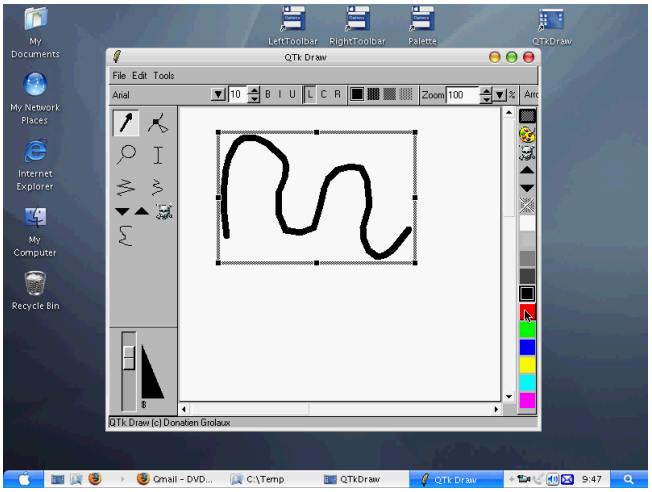

Figure 15. Drawing Application DUI before distribution.

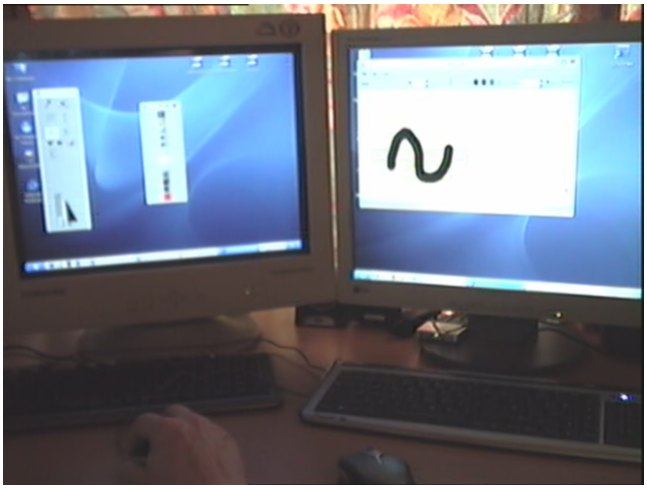

Figure 16. Drawing Application DUI after distribution with another desktop platform.

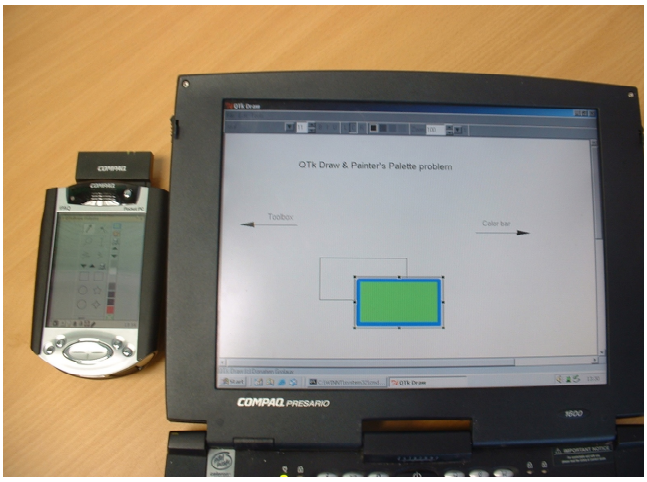

Figure 17. Drawing Application DUI after distribution with another mobile platform.

Figures 16, respectively 17, presents the results of DUI distribution after the toolbars and palettes have been migrated to another desktop, respectively another mobile platform. In this case, the various platforms were used by the same user, but we can also assume that these platforms are used by different users provided that they are connected together via the same network, local or wireless.

5. CONCLUSION

This paper presents a toolkit for peer-to-peer distribution of any graphical UI that supports the following usages: multi-monitor, devices, platform, display, and users. It also presents a detailed description of the software architecture developed for this toolkit. Contrarily to model-based design of multiple UIs [7,8,9, 10], this approach does not rely on any model per se, although each user interface is stored in a distributed way through its properties.

ACKNOWLEDGEMENTS

This work is supported by the SELFMAN (Self Management for Large-Scale Distributed Systems based on Structured Overlay

Networks and Components) European project of the 6th Framework Programme (FP6-IST-2005-034084). We also acknowledge the support of the UsiXML (User Interface extensible Markup Language – http://www.usixml.org) project.

REFERENCES

[1] Ayatsuka, Y., Matsushita, N., and Rekimoto, J. 2000. HyperPalette: a Hybrid Computing Environment for Small Computing devices. In *Proc. of CHI'2000*. ACM Press, New York, pp. 133-134.

[2] Bandelloni, R. and Paternò, F. Migratory user interfaces able to adapt to various interaction platforms. *Int. J. Human-Computer Studies 60*, 5-6 (2004), pp. 621-639.

[3] Bharat, K.A. and Cardelli, L. 1995. Migratory Applications Distributed User Interfaces. In *Proc. of UIST'95* (Pittsburgh, Nov. 1995), ACM Press, New York, pp. 132-142.

[4] Chung, G. and Dewan, P. 2004. Towards Dynamic Collaboration Architectures. In *Proc. of the ACM Conf. on Computer Supported Cooperative Work CSCW'2004*, pp. 1-10.

[5] Dewan, P. and Choudhary, R. Coupling the User Interfaces of a Multiuser Program. *ACM Transactions on Computer-Human Interaction 5*, 1 (1998), pp. 34-62.

[6] 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

[7] Eisenstein, J., Vanderdonckt, J., and Puerta, A. 2001. Model-Based User-Interface Development Techniques for Mobile Computing. In *Proc. of IUI'01* (Santa Fe, January 14-17, 2001), ACM Press, New York, pp. 69-76.

[8] Griffiths, T., Barclay, P.J., Paton, N.W., McKirdy, J., Kennedy, J., Gray, P.D., Cooper, R., Goble, C.A., and Pinheiro, P. Teallach: a Model-based User Interface Development Environment for Object Databases. *Interacting with Computers 14*, 1 (December 2001), pp. 31-68.

[9] Grolaux, D., Van Roy, P., and Vanderdonckt, J. 2004. 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*, pp. 422-430.

[10] Grolaux, D., Vanderdonckt, J., and Van Roy, P. 2005. Attach me, Detach me, Assemble me like You Work. In *Proc. of INTERACT'05*, pp. 198-212.

[11] Grolaux, D. 2007. *Transparent Migration and Adaptation in a Graphical User Interface Toolkit*, Ph.D. dissertation, Department of Computing Science and Engineering, Université catholique de Louvain, 2007.

[12] Grudin, J. 2001. Partitioning digital worlds: focal and peripheral awareness in multiple monitor use. In *Proc. of CHI'01*, ACM Press, New York, pp. 458-465.

[13] Han, R., Perret, V., and Naghsineh, M. 2000. WebSplitter: A Unified XML Framework for Multi-Device Collaborative Web Browsing. In *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, pp. 221-230.

[14] Hutchins, R., Meyers, B., Smith, G., Czerwinski, M., and Robertson, G. 2004. Display Space Usage and Window Management Operation Comparisons between Single Monitor and Multiple Monitor Users. In *Proc. of AVI'04*, ACM Press, New York, pp. 32-39.

[15] Hutchings, H.M. and Pierce, J.S. 2006. Understanding the Whethers, Hows, and Whys of Divisible Interfaces. In *Proc. of AVI'06*, ACM Press, New York, pp. 274-277.

[16] Loeser, C., Mueller, W., Berger, F., and Eikerling, H.-J. 2003. Peer to peer networks for virtual home environments, in *Proc of HICSS-36*, IEEE Computer Society Press.

[17] Luyten, K. and Coninx, K. 2005. Distributed User Interface Elements to support Smart Interaction Spaces. In *Proc. of the 7th IEEE Int. Symposium on Multimedia*, IEEE Comp. Society, Washington, DC, pp. 277-286.

[18] Luyten, K., Vandervelpen, Ch., and Coninx, K. 2002. Migratable User Interface Descriptions in Component-Based Development, in *Proc. of DSV-IS'2002*, pp. 44-58.

[19] Luyten, K., Van den Bergh, J., Vandervelpen, Ch., and Coninx, K. 2006. Designing distributed user interfaces for ambient intelligent environments using models and simulations. *Computers & Graphics 30*, 5 (2006) 702-713.

[20] Mesaros, V., Carton, B., and Van Roy, P. 2004. P2PS: Peer-to-Peer Development Platform for Mozart. In *Proc. of Second International Mozart/Oz Conference MOZ'04*. LNCS, Vol. 3389, Springer, Berlin, pp. 125-136.

[21] Myers, B.A. Using Handhelds and PCs Together. *Communications of the ACM 44*, 11 (2001), pp. 34-41.

[22] Tan, D.S. and Czerwinski, M. 2003. Effects of Visual Separation and Physical Discontinuities when Distributing Information across Multiple Displays. In *Proc. of INTERACT'03,* IOS Press, pp. 252-260.

[23] Vanderdonckt, J., Furtado, E., Furtado, V., Limbourg, Q., Silva, W., Rodrigues, D., and Taddeo, L. 2001. Multi-model and Multi-level Development of User Interfaces, in "Multiple User Interfaces - Cross-Platform Applications and Context-Aware Interfaces", John Wiley & Sons, pp. 193-216.

[24] Vandervelpen, Ch., Vanderhulst, G., Luyten, K., and Coninx, K. 2005. Light-Weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments. In *Proc. of ICWE 2005*, pp. 197-202.

[25] Van Roy, P. and Haridi, S. 2004. Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge.

[26] Yanagida, T. and Nonaka, H. Architecture for Migratory Adaptive User Interfaces. In *Proc. of CIT'2008*, pp. 450-455.