Modelling and Developing Distributed User Interfaces based on Distribution Graph

Jérémie Melchior[1], Jean Vanderdonckt[1] and Peter Van Roy[2]
[1]Louvain Interaction Laboratory, Louvain School of Management, Place des Doyens, 1
[2]Dept. of Computing Science and Engineering, Ecole Polytechnique de Louvain, Place Sainte-Barbe, 2
Université catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium
{jeremie.melchior, jean.vanderdonckt, peter.vanroy}@uclouvain.be

***Abstract*—This paper introduces, motivates, defines, and exemplifies the concept of distribution graph as a way for modelling and developing Distributed User Interfaces of interactive systems. A distribution graph consists of a state chart model enriched as follows: states represent individual states of entities involved in the distribution as well as a collective representation of their synchronization; transitions are represented by event-condition-actions where the action part consists of a distribution script. A distribution script expresses the distribution behaviour based on distribution primitives. These primitives are basic operations that manipulate parts or wholes of user interface for distribution at run-time. These primitives are themselves implemented on top of an environment for distributed computing that is implemented for four major computing platforms (i.e., Microsoft Windows, Mac OS X, Linux, and Mobile Linux). Thanks to the capabilities provided by this environment, the user interfaces belonging to these distributed systems can be run indifferently on any of these computing platforms. This paper defines the new concepts introduced for this purpose, i.e., distribution primitive, distribution script, and distribution graph, and demonstrates how they can effectively support distributed user interfaces.**

Keywords-Distributed User Interface, Human-Computer Interaction (HCI) modelling, Ubiquitous computing

I. INTRODUCTION

Significant progress has been made in the area of *multi-device* User Interfaces (UIs), where UIs are produced for several devices simultaneously, or in *migration* of UIs, where UIs are migrated from one device to another while maintaining task continuity. However, less work has been devoted towards dividing a UI across devices, displays, or platforms, where they are used by the same user or shared by different users [1]. A *Distributed User Interface* (DUI) is hereby defined as any application User Interface (UI) whose components can be distributed across different displays of different computing platforms that are used by different users, whether they are working at the same place (co-located) or not (remote collaboration). Consequently, DUIs allow for the UI to be spread out over a set of displays/devices/platforms taking advantage of each display/device/platform's unique properties instead of residing on a single display/device/platform with the interaction capabilities that are constrained on this display/device/platform [2].

DUIs have been subject to several studies that investigate further their specific characteristics that may lead to design implications. This includes use of multiple monitors on a same computing platform by a single user [3], use of multiple platforms by a single user with data synchronization between the platforms enabling continuity of tasks [4], exchange of information between platforms belonging to different users (e.g., by the Pick & Drop interaction techniques [5]), moving information between displays on a single platforms (e.g. [5]), partition of tasks across displays for a single user [6], sharing common information private on some platforms, Beale and Edmondson conducted user surveys in order to determine the user behaviour 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 for the work conducted here. Tan & Czewinsky found that physical discontinuities had no effect on performance, but found a detrimental effect form separating information within the visual field, when also separated by depth.

There is a high need of some visualization of the distribution. None of the cited works have provided a way to visualize what was distributed on each platform. The reason comes from the pre-programmed disposition and static environment provided in their examples. The dynamic of the platforms are not considered, such as platforms joining and living at run-time.

Due to the multiplicity of interaction techniques in DUIs, Nacenta *et al.* conducted a study to compare the efficiency of six techniques for moving objects from a platform (e.g., a tablet) to another one (e.g., a table top) in four different distance ranges and with three movement directions. Their study suggests that spatial manipulation of data was faster than pressure-based techniques.

On the one hand, 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 describing them in a formal way [4]. 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 is a typical example of a distributed task: one player se-

978-1-4577-1938-7/12/$26.00 ©2011 IEEE

lects 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.

A. Related Work

Wincuts [7] augment window managers by letting users acquire and interact with alternative views of arbitrary regions of existing windows. The Frisbee6 is a widget that acts as a telescope to a remote area on the display. Users manipulate remote items by interacting with their proxies within the Frisbee's main area and reposition items on the main display by moving them through specified transfer channels. Speakeasy [8] consists of a computing framework that is designed to support use of resources such as displays/devices/platforms that appear/disappear opportunistically, called *recombinant computing* [9]. In most circumstances, distributing parts or whole of a Graphical User Interface (GUI) is primarily driven by the system itself or by a predefined procedure that is rarely flexible. When there is some procedure in order to distribute some parts of a GUI, operations required to conduct this distribution are often at a high level –which is appropriate– but rarely subject to parameterization.

B. Conceptual modelling

A user working with an interactive system is represented by the context of his work. A user may often change the context in which he is while using an application. Thus, developers try to improve applications by taking into account the possible contexts of use.

There already exist lots of researches about context-aware applications [12, 13] but there is no adaptation when context switches. The adaptation proposed to context switches are either for specific or pre-programmed for applications. Thus there is no big need for visualization of the distribution state. Most of the time, platforms are co-located which enables them to see directly the result of a distribution operation on the target. DUIs are not limited to co-located platforms which lead to higher needs for feedback and visualization.

In this paper, we present concepts to improve the way applications react to a context switch. We first define the important notions as the context of use and the distribution state. Then, we introduce the concept of distribution primitives which represents the operations that can be applied to UIs. Finally, we introduce the concepts of distribution scenario and graph.

A distribution scenario is a sequence of operations on UIs. A distribution graph is a state diagram where states are distributed and transitions are events that trigger a new distribution of the UIs.

Thanks to these concepts, we believe that applications will be able to dynamically and automatically adapt to context switches while staying under the control of the users.

As a proof of concept, we describe an application in two variants using these concepts. A simplified and a complex Pictionary self-adapt the distribution of the UIs when role assignments change.

II. CONCEPTS

A. Assumption

As it is physically impossible for a user to be at the same place at the same time, a user is only able to directly use platforms located in the same location. If two platforms are at different places, the user is only able to use at most one of them.

B. Context

A **user** is a concept of a human person with some characteristics describing the level of use that he is able to accomplish. The set Users is a collection of users with at least one distinct attribute. Here are the different characteristics:

- Task experience: low / average / high
- System experience : low / average / high
- Motivation : low / average / high
- Complex device experience: low / average / high

The **platform** on which the user interacts with the tasks is mostly represented by the device. If the user is switching from one computer to another or change the number of devices he uses, there is a change of platform.

Depending where and when the user is accomplishing the tasks, he is evolving in an environment. For example, he can be at work, at home or traveling. Even the easiest tasks can become difficult if the environment is not appropriate for it. A change of environment happens if the user moves to somewhere else, if the environment becomes quieter or louder, and so on.

Each user has its own context of use describing the environment and the material in which he is. A context of use C is composed by a platform P, a user U and an environment E.

$$C \rightarrow (P,U,E) \quad (1)$$

A context of use is bound to only one platform, one user and one environment but it does not mean that the interactive system is only one user in a single environment using a single platform. The platform/display is the tool used by to interact with the system. The context according to this definition is displayed in Figure 1. **A simple context with only one platform/display, one user and in one environment**.

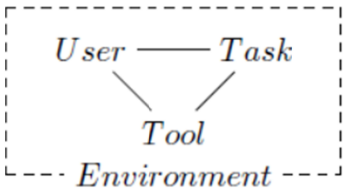

Figure 1. A simple context with only one platform/display, one user and in one environment.

There are many contexts of use to describe a multi-user interactive system. This definition is limited to a single platform, user and environment. Here we introduce the concept of a distributed context in Equation 2.

$$C_d \rightarrow (\mathbf{P}_d,U_d,E_d) \quad (2)$$

We use this notion when a system has the ability to distribute one of the components. P_d is the vector P_1, P_2, …, P_m representing m platforms on which the system may be run. U_d is U_1, U_2, ..., U_n where n users may use the system whenever and wherever they want. Finally, E_d is E_1, E_2, ..., E_p where p environments are possible environments for the users to be in. A non-distributed context has m = n = p = 1 while a distributed context has m > 1, n > 1 or p > 1.

A distributed platform P_d means that the system is running across several platforms. The most common are desktop, laptop and netbook computers as well as mobile phones. If the system runs on top of two devices with the same platform, it is also a distributed platform. Example of context with multiple platforms can be found in Figure 2.

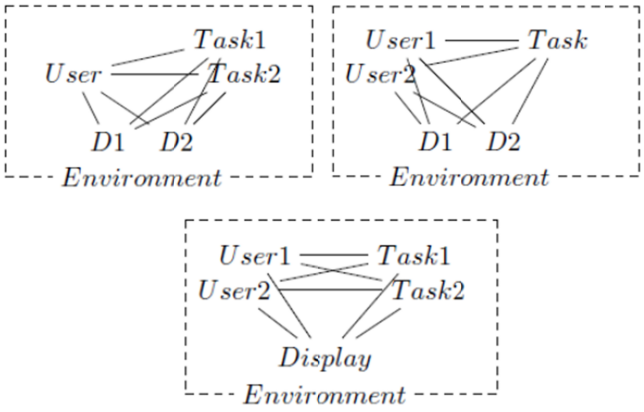

Figure 2. Example of multi-platform contexts.

These three examples can be expressed with the notation defined in this section. For the first example, there are one user, two tasks and two displays, the context is then defined as:

$$C_{d1} \rightarrow (\{D_1,D_2\}, \{U_1\}, \{E\}) \qquad (3)$$

The concept of task is not part of the context. It only describes the complexity of the goal of the system. The context of the second example is defined as:

$$C_{d2} \rightarrow (\{D_1,D_2\}, \{U_1,U_2\}, \{E\}) \quad (4)$$

As for the first example, there is only one environment because we find it already a good challenge without the need to change the environment which would make readability much more difficult. The last example is thus defined as:

$$C_{d3} \rightarrow (\{D_1\}, \{U_1,U_2\}, \{E\}) \qquad (5)$$

The system may also be distributed through different users. It may be a distribution in space, with users located in different countries or regions. Also in time, people may use the system at different periods of time. As several users may be involved, the system may be used in different environments. From the user point of view, the context may also be a distributed context. Nevertheless, it is possible that some users have a non-distributed context. It means that *m = n = p = 1*.

It is important to notice that some platforms, some users and some environments may not be attached to any context of use at a certain moment of time. For example, two users in the same room with their own platforms are in Environment E_1:

$$E_1 = \begin{cases} U_1 \rightarrow P_1 \\ U_2 \rightarrow P_2, P_3 \end{cases} \qquad (6)$$

Now, U_2 leaves environment E_1 to environment E_2. In the room E_1, he was using a projector which is not available anymore. Thus, after U_2 has left the room, the platform P_3 is not currently attached

$$\begin{cases} E_1 = \{U_1 \rightarrow P_1 \\ E_2 = \{U_2 \rightarrow P_2 \end{cases} \qquad (7)$$

As there may have many instances of each component, we are interested in studying the relationships between each component.

C. Distribution state

The first concept we needed to introduce in order to visualize the actual distribution is a state for the system. Each platform needs to give information about what it is currently displaying. The state of the whole system is called a *distribution state*. A distribution state is a snapshot of a system at a certain moment of time in which the system is stable. It means that a state represents the context of use of the whole system.

Each user has a different context of use which is an individual distribution state. The distribution state for the user is the distribution of his UI across several platforms available to him. There are two kinds of platforms, the individual platforms which are not distributed and shared platforms which are distributed with other users. An example of a shared platform is a large screen display in a room.

The **distribution shared state** is the sum of all individual states taking into account the different inter-state relations. For example, Adrien is working in his room with his desktop computer. The context for this situation can be:

$$C1_a \rightarrow (desktop, adrien, quiet) \quad (8)$$

Here we have a distribution state for Adrien. We call this the distribution individual state of Adrien. The representation of this state is:

$$S1_{adrien} = \{C1_a\} \qquad (9)$$

At the same time, his friend named Bastien is playing with him on a distributed application. The distribution individual state for Bastien may be:

$$C1_b \rightarrow (\{desktop, laptop\}, bastien, quiet) \quad (10)$$

Thus, the system is composed by two individual states. For the example, the distribution shared state is:

$$S1 = \{C1_a, C1_b\} \qquad (11)$$

Both types of state are distributed. Bastien in context $C1_b$ has distributed the application on his two platforms.

D. Distribution primitives

The modifications that could happen in applications after a context change may follow some well-formed rules. To adapt the UIs, the application may apply several distribution primitives. The simplest statements are the Display and Undisplay. The first allows one or several graphical elements to be dis-

played on the screen while the second hide them. For example, the Display statement is defined by:

display(object:UI, site:Site)
Pre: The site has already a window ready to display the object.
Post: The object is display.

Then we have the Move and Copy. It allows to move and copy one or more elements from a platform to another. The definition for the copy is:

copy(object:UI, source:Site, target:Site)
Pre: The target site has already a window ready to get the object.
Post: The object is copy and inserted in the target.

The context of use associated with the source is C1 → (P1, U1, E1). The target one is C2 → (P2, U2, E2). There is at least one of the elements in C_2 that is distinct from C_1. There are also two other basic operations: Insert and Switch. While the first inserts an element in a container, the second switch two elements. Complex operations such as merging and splitting apart UIs are also statements.

E. Distribution scenario

With the distribution primitives, it is possible to adapt the UI. The power of these statements is the ability to be automatically triggered when some events happen on the system. For example, when a new user connects to the application, he may wish to collaborate and thus receive a copy of some parts of the UI. Several statements may be triggered to allow the user to take part of the application. In this section, we introduce the notion of distribution scenario. It consists of automation in order to simulate events in the application. Concretely, the distribution scenario is a sequence of distribution primitives leading to the whole execution of the application.

Let us define Scenario 1 which will start and exit an application with a single window and button:
Scenario 1:

- button(text:"Button1" name:B1)
- display(object:B1 site:A)
- wait(5)
- undisplay(object:B1 site:A)

This simple example has two distributions primitives and two commands. It first creates a button with the command button(text:"Button 1" name:B1). Then, the statement display put this button in an already created window. The wait command will wait 5 seconds before executing the next statement. The statement undisplay hides the button which lead to the end of the scenario.

F. Distribution graph

A state diagrams as used in [11] is a combination of states and transitions into the same diagram. The state represents the system at a certain period of time. The system evolves from a state to another by using transition links. Each transition represents an event. A *distribution graph* is the same notion as a state diagram. The difference comes from the definition of state and transition. A distribution graph is a state diagram where states are distribution state as defined in 2.3 and transitions are sequence of distribution primitives.

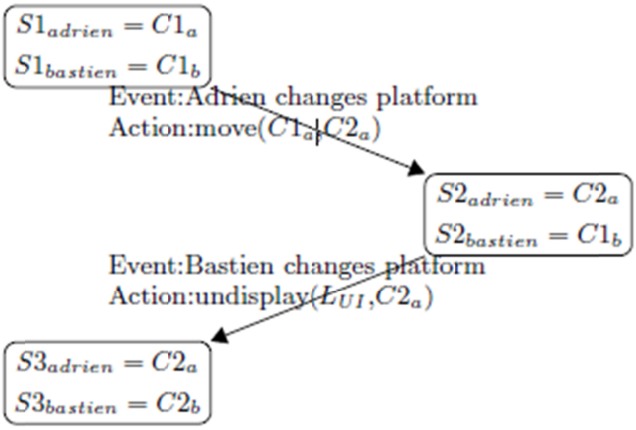

Figure 3: Example of distribution graph for Adrien and Bastien contexts.

In Figure 3, we see an example of distribution graph. It starts in S1 defined in Definition 8. Then Adrien decides to switch from his desktop to his laptop, this leads to Adrien being in a new context. The effect of this change is that the UI displayed on the desktop is now moved to the laptop. We are now in state S2. Lastly, Bastien turns off his laptop entering a new context for state S3. The UI displayed on the laptop is now undisplayed.

III. CATALOG OF DISTRIBUTION OPERATIONS

This section introduces a summary of the catalog for the distribution operations. For complete descriptions, please refer to www.usixml.org.

Name	**Effect**
Display	Display an item in one or more UIs
Undisplay	Hide an item from one to many UIs
Move	Move an item from a UI to another one
Copy	Copy an item of an UI to another one
Insert	Insert an item in a container of a UI
Switch	Exchange two components in the same or different UIs
Merge	Merge two UIs together
Separate	Explode a UI in two or more separated UIs

Table 1: Catalog of distribution operations.

IV. CASE STUDY

In this section, the different notions previously introduced are instantiated in a simple example. The case study presented here is a Pictionary game with different variants. The first one

is a small example of the different concepts while the other one is a complete and more complex case study. A more complex case study is also presented; it is a board game like the Game of the Goose and Snake and Ladders.

A. *Pictionary*

Pictionary is a game where players have to guess words helped by drawings. We have chosen this case study because it is easy to create some variants, is an easy to use and easy to understand game and brings fun to players. The game may be played with and without teams depending on the number of players. Security aspects are not handled here as we consider this game as familial and friendly. The way the game is distributed is to prevent players cheating easily but no guarantee is provided against cheaters. There are several roles in the Pictionary: the drawer, the guesser and observers. Each role is associated to a UI to enable the task allowed with the role. For example, the drawer has to be able to draw things, the guesser to see what is the current drawing and observers should be able to verify that the game is going right. The distribution of the UI will be automatically rearranged when roles change. The minimal information is the remaining time and the game status. Each UI has this information in common plus other specific information. Compared to collaboration games, where each platform would run its own separate client of the game, a game based on DUIs does only need to be part of the system. Thus, it allows the system to start on a single platform and to extend with platforms arriving into the system.

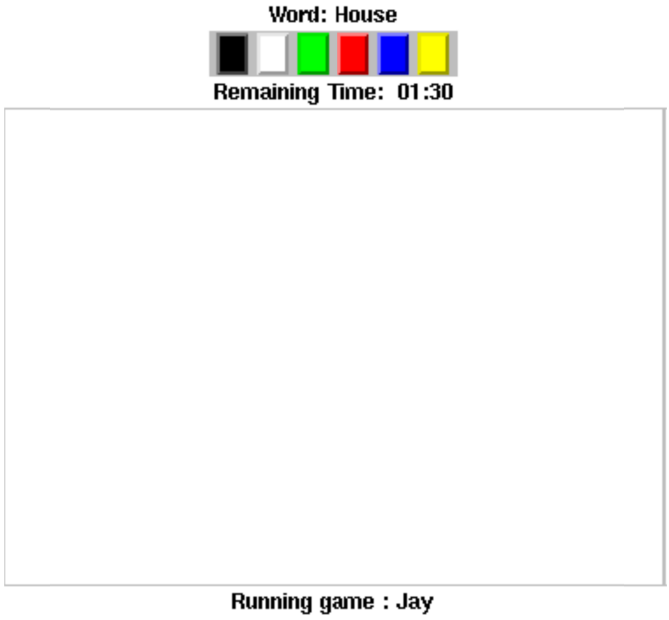

Figure 4: User Interface of the drawer.

Here are the key roles and their Graphical User Interfaces (GUI):

- **Drawer:** a person that helps finding the word to guess by drawings. The minimum UI needed for this role as in Figure 4 is the drawing tools and information about the game. The user sees the word he has to draw about and has the tool to draw on the shared area.
- **Guesser:** a person that has to guess the word. If there are teams, guessers are in the same team as the drawer. The drawer may not be a guesser and his associated UI is in Figure 5.
- **Observer:** a person that is not currently trying to guess the word. This role is for opponents to the playing team. When there are only two players, the drawer will also have the observer UI. For observers, the UI is mainly the ability to start and end a game as in Figure 6.

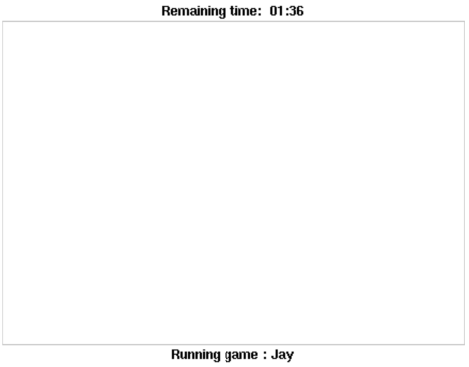

Figure 5: User Interface of the guessers.

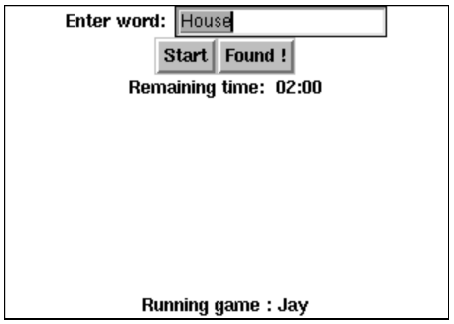

Figure 6: User Interface of the observers.

Depending on the role associated to a player, he will get the appropriate GUI for his role. When roles change, the distribution of the UIs is reprocessed to keep a coherent state.

B. *Simplified Pictionary*

The first case study is a simplified variant of the Pictionary, where there is no team. Observers are players waiting for next game to begin. Each person is a single player as the drawer or a guesser. There is only one drawer at the same time but there may be several guessers. The game starts with an initial state where the application is not started, the current state is empty. When the first user starts the application, he needs to create a room for the game. Other players will then join this room to play. The UI for this first player allows him to create a new game as in Figure 7. The pseudo-code to create this UI is reproduced in Figure 8.

Figure 7: Creation of a game when no game is already started.

```
{Display
  td(name:p1
     button(name:b_c glue:e text:"Create:")
     )
}

{Display
  entry(name:e_c glue:w bg:white
        init:"Own game"
        handle:HEntry return:R)
#p1}
{Display lr(name:create_game b_c e_c)#p1}
{Display
   td(name:p1
      create_game
      td(name:join)
      label(name:status glue:swe
            text:"Waiting for a game"
            bg:white)
      )
}
for I in {DiscoverGames} do
   {Display
      button(name:b#I glue:nswe
             text:"Join "#I}
#join}
end
```

Figure 8: Pseudo-code for creating initial UI.

The first steps create and display a button and an entry in a new window. These two widgets will then be associated in a new widget arranging them from left to right. The last step is the creation of the window with all widgets arranged in the desired order. The td widget created by the main Display contains the widgets create_game, status and zero to several buttons. The name used when we create widgets is the key to use the widget later. As we can see, the button named bc appears in the creation of the lr widget. As there is currently only one player, the game cannot be started. The application still needs a player before being able to start. The next state is the connection of a second player. Two players are now connected as Observers waiting for the game to begin. Figure 9 shows the current state of the application for the different roles. To create the UI of Player 2, it only needs to copy Player 1 UI with code in Figure 10.

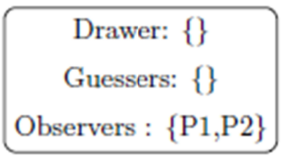

Figure 9: Second state. Player 1 and 2 are connected

```
{Copy p1 td(name:p2)}
```

Figure 10: Pseudo-code for updating UI after game creation.

As the game is created, both players got an update with their UI looking like in Figure 11.

Figure 11: Creation of a game when no game is already started.

The code to update the UI is presented in Figure 12. Now, both players are waiting for the game to begin. Their UI is the same until one of the players chooses to start the game. As the minimum number of players is reached, the game can start and each player will now be assigned to a role. In Figure 13 Player 1 becomes the drawer and Player 2 a guesser. When Player 1 becomes the drawer, the UI has to be redistributed to this new role assignment. The result of this redistribution appears in Figure 14. This UI slightly differs from Player 2 because Player 1 has to stay an observer. As Player 2 is the guesser, he is not allowed to have the 'Start' and 'Found!' buttons. This role should be assigned to observers, if there were any. The only solution is to assign this role to the drawer itself.

Thanks to the UI he is already playing with, the adaptation of the UI is only a small piece of code. In three statements, the UI is adapted. The code can be found in Figure 15. The UI of the second player also has to be adapted to the new role assignment. He now has the guesser UI as in Figure 16. The code for this adaptation can be found in Figure 17. Here, the transition triggered by the connection of a new player can be a loop from the current state. The only modification happening when a new player connects is an update in the observer list. It is exactly the same if Player 1 leaves the game.

```
{Undisplay create_game#p1}
{Update status "Running game: "#Name}
{Display
  td(name:observer
     lr(name:enter_word glue:nwe bg:white
        label(bg:white text:"Enter word: ")
        entry(glue:w bg:white
              init:"House"
              handle:HEW)
        )
        lr(name:start_found bg:white glue:nwe
           button(glue:e text:"Start")
           button(glue:w text:"Found !")
        )
        lr(name:remaining_time bg:white glue:swe
           label(glue:e bg:white
                 text:"Remaining time: ")
           label(glue:w bg:white text:"02:00")
        )
  )
#p1}
{Display td(name:p1 create_game observer status)}
```

Figure 12: Pseudo-code for updating UI after game creation.

Every time the word to guess is found, the winner becomes the drawer while the drawer becomes a guesser. If the word is not found in the time let for the game, the current drawer wins and stays the drawer. The game needs at least two players.
In the state where Player 1 is the drawer, two transitions can be triggered. If P1 wins, the system stays in the same state. Otherwise, the system has to redistribute the UIs. In Figure 18, the winner is Player 2. As he won, he becomes the new drawer and Player 1 becomes a guesser. The merging of the two first states is represented by a dashed red loop. The last step remaining is the redistribution of the UI when role are exchanged. This means that we want to switch the upper part of Player 1's UI to Player 2's UI. This can be done with the code in Figure 19. Here we introduce the ability to choose the position of the widget. The drawing_tool widget will be placed first in widget p2.

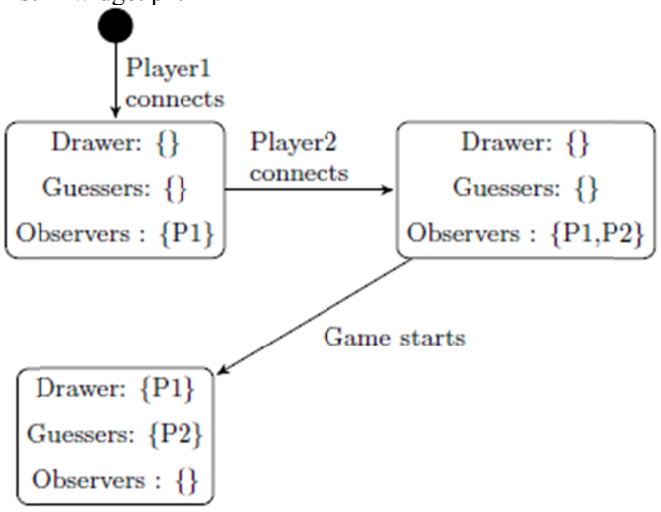

Figure 13: State diagram of the current system.

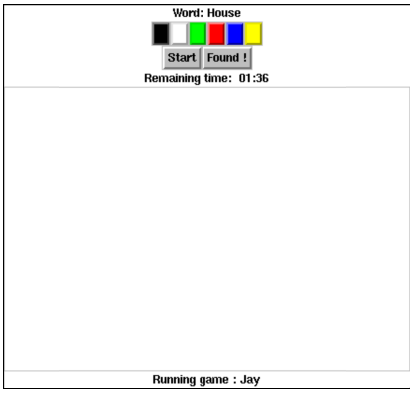

Figure 14: Player 1 becomes the drawer and stays observer.

```
{Undisplay enter_word#observer}
{Display
   td(name:drawing_tool
      label(name:word bg:white
            text:"Word: House")
      {Record.adjoin CD lr(name:colors
                           glue:n
                           relief:sunken
                           bg:white)}
      )
#p1}
{Display
   td(name:p1
      drawing_tool
      observer
      canvas(name:drawing area
      bg:white glue:nswe)
      status
   )
}
```

Figure 15: Pseudo-code for updating UI from Observer to Observer-Drawer.

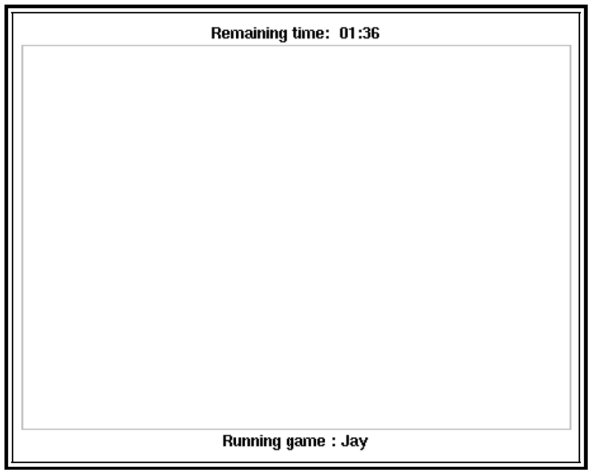

Figure 16: Player 2 becomes the guesser.

```
{Undisplay enter_word#p2}
{Undisplay start_found#p2}
{Display canvas(name:drawing_area
                bg:white glue:nswe)#p2}
```

Figure 17: Pseudo-code for updating UI for Player 2 becoming a guesser.

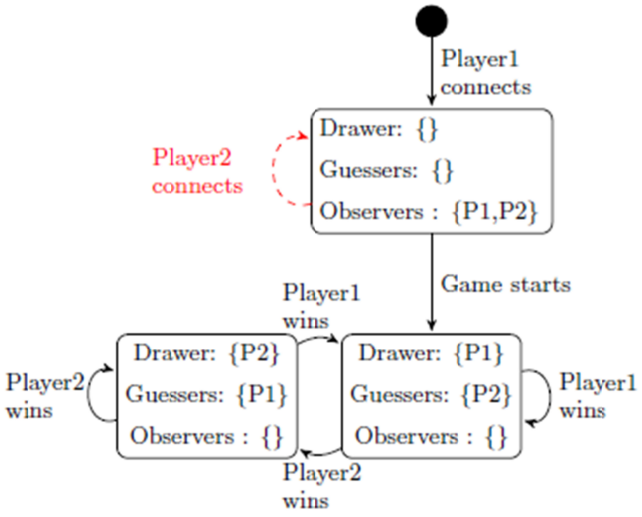

Figure 18: Complete diagram of the whole system.

```
{Move drawing_tool p2 pos:first}
```

Figure 19. Pseudo-code for switching player's role.

C. *Extended Pictionary*

In this section, we introduce a dynamically extended version of the Pictionary as a real case study. As the number of player increases, the simplified version may be extended to support teams. The minimum required for this variant is four players separated in two teams with two players.

A team is composed by at least two players. The team that is currently drawing needs a drawer and at least one guesser. The members of the other teams are observers. The distribution graph for two teams is presented in Figure 23 based on the one in Figure 18.

If a guesser finds the word within the guessing time, he becomes the drawer and the team stays playing. If the guessing time is passed and no guesser found the word, another team is given the ability to play the same word. If the team finds the word, this team becomes the new team playing. If not, another team takes the turn until every team has played with the word. Every time a word is found, it increases the points of the team currently playing.

D. *Game of the Goose / Snakes and Ladders*

Another kind of game we introduce as a case study for this modelling with distribution graph is inspired from the game of the goose and snakes and ladders games. In both games, you have a board filled with squares. Each square is a different step with some action associated to it. The game we created has a virtual board with squares associated to games. As each game has its own UI, the user is providing with a different UI at each step during the game. The distribution appears at each step, because the game will distribute the right UI to the involved users. Here are three different games:

- **Single-player**: Minesweeper is a game in solo where the user has to clear the board without detonating a mine.
- **Two-player**: chess is a game where two players are opponents; one is in white and the other in black. The goal of the game is to checkmate the opponent's king.
- **Multi-player**: the Pictionary as introduced in the previous sections is a game where each player has his own role. There is always someone who has to guess what another one is drawing.

This game is even more complex than the Pictionary variants introduced in the previous sections. The game has to provide a UI to each player for the main game, but also to each player for the current action.

For a single-player game, the other players may have no UI at all or be observers. For a two-player game, two players need to get a UI which allow them to see what they can do and to partially see what he can from the other player.

The players that are not opponents in the game may join the game as observers.

In a multi-player game, each player will have his own role and the UI must be distributed in a way that enable all the players to have access to their role functionalities but without giving them information that they should not have.

The distribution scenario for this game is much more complex than for the Pictionary. An addition to this case study is the ability to add new games on the fly, or change the association between squares and games. This game is thus a dynamically evolving environment with several users with several platforms. At least one platform is used for the system, but to allow hiding information, it is recommended to have at least one platform by user/player.

There exist three main states for this game as respectively reproduced from Figure 20 to Figure 22.

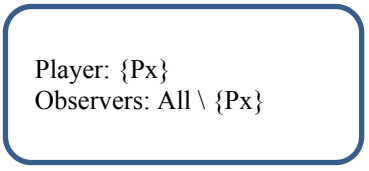

Figure 20. Single-player game, one player (Px) only while other are observers.

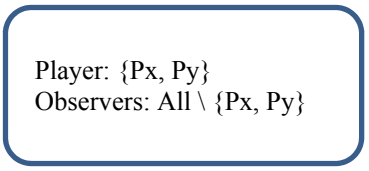

Figure 21. Two-player game, two players, Px is the first player and Py is the opponent.

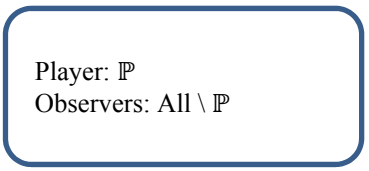

Figure 22. Multi-player game, player in □ will have a role but also Observers may have on.

In the first case, we have games like the Minesweeper which does only involve one player. Px will be the one playing the game and is the current player of the main game. Other examples of single-player games are a Rubik's cube and Pinball.

The two-player games will involve the two players (Px and Py). The actual player (Px) will have to face the opponent (Py). This can happen when two players are in the same square. The player arriving in the square (actual player of the main game) will have to defeat the one that was already there (the opponent). Chess is a good example of a two-player game.

The last case is multi-player games such as the Pictionary. All the players may be part of the game. The one that are in the same square will have the main roles (drawer and guesser for the Pictionary) while the others will act as observers or passive players.

V. IMPLEMENTATION

A. Distribution primitives

The distribution operations have been implemented in Mozart framework in OZ language [14]. The advantages of this framework are the support of the most used operating systems (Microsoft Windows, Unix-based distributions such as Linux and Mac OS X), the network layer allowing transparent migration. The work described in this paper goes further than a prototype. The main goal is to develop a complete toolkit for creating DUI as simple as designing UIs with programming tools. We use distribution operations to control the way the DUI are distributed across several devices on various platforms. These operations can be triggered manually by code, by command line scripts or an interpreter, or by meta-GUI controlling the whole distribution.

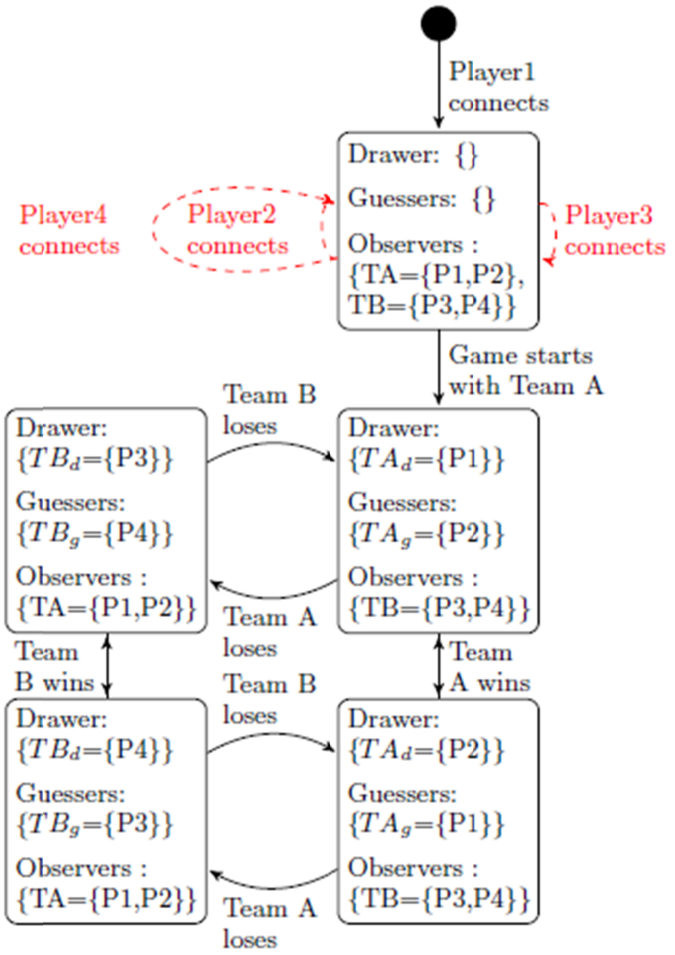

Figure 23: Complete diagram of the whole system.

B. Implementation of the primitives

The display operation is the first to use for creating DUI. Because of space limitations, this section only describes the implementation of this operation for the example 5:

```
{Display td(name:p1
            button(name:b_c glue:e text:"Create"))}
```

Figure 24: Simple example of a distribution operation.

Here, we need to display some elements arranged in a top-down (td) panel. The widgets placed here is a simple button with text Create. The procedure display receives a record describing the widgets to display as in Figure 24. Here the top-level widget can only be a top-down or left-right widget. The code of the Display function is in Figure 25.

```
proc {Display Param}
  case {Record.label Param}
  of td then {Dis td Param}
  [] lr then {Dis lr Param}
  end
end
```

Figure 25: Part of code of the procedure Display.

The Dis function of Figure 26 is called by the display procedure. As no window was created before, a new window is created according to the widgets placed in the records. Each widget is stored with his name to be managed later.

```
proc {Dis Wid Param_Old}
   Title Widget
   Param = {Record.subtract Param_Old name}
in
   Widget = {Process Param {Arity Param} Wid()}
   if {HasFeature Param title}
   then Title= Param.title
   else Title= "No name" end
   Win = {QTk.build
            {Adjoin Wid(title:Title)
                    Widget}}
    if {HasFeature Param_Old name}
       then Names := {Append @Names
      [Param_Old.name#Win]}
   end
  {Win show}
end
```

Figure 26: Part of code of the procedure Dis.

The last important procedure is Process from Figure 27. Each widget will be created in two parts. The first is the object representing the widget and manage by its name. The second is the value which is used for the GUI.

In Figure 8, we presented the scenario to create the UI displayed in Figure 7. No matter where the scenario is triggered, it allows programming the action resulting to an event in a distribution graph. Each event is bound to an action. When an event occurs, the action corresponding will be triggered. Several distribution operations will then redistribute the system across the different contexts of use.

```
fun {Process Param List Wid}
   fun {ProcessHelper Param List Wid}
      case List of nil then Wid
      [] I|R
      then NewWid Lab={Label Wid} in
        if {Value.type I} == int
        then
           Wid_I= Param.I Lab_I= {Label Wid_I}
```

```
      in
        NewWid = {Process Wid_I
         {Arity Wid_I}
         Lab_I()}
      else
         case I
         of glue then
           NewWid = Param.glue
          [] name
            then Names := {Append
             @Names
             [Param.name#NewWid]}
          [] text then NewWid = Param.text
         end
      end
    if {Value.isFree NewWid} then
       if I == name then
         {ProcessHelper
           Param
           R
           {Adjoin Wid Lab(handle:NewWid)}
         }
       else
         {ProcessHelper Param R Wid}
       end
     else
       {ProcessHelper
         Param
         R
         {Adjoin Wid Lab(I:NewWid)}
       }
       end
     else
       error
     end
   end
in
   {ProcessHelper Param List Wid}
end
```

Figure 27: Code of the Process function. Processing every widget one by one.

VI. Conclusion and Future Work

This paper introduced the notion of distribution graph in order as a way for modelling and developing Distributed User Interfaces. The graph is a state diagram where states represent the current distribution of the system across the different contexts of use. It also describes the implementation of three case studies, a simple and an extended Pictionary, and a complex board game. This new methodology needs to be validated, i.e. with these case studies applied as real game.

Acknowledgment

The authors 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, Direction générale opérationnelle de l'Economie, de l'Emploi et de la Recherche (DGO6).

References

[1] J.T. Biehl, W.T. Baker, B.P. Bailey, D.S. Tan, K.M. Inkpenl, M. Czerwinski, IMPROMPTU: A New Interaction Framework for Supporting Collaboration in Multiple Display Environments and Its Field Evaluation for Co-located Software Development Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems , pp. 939-948; 2008

[2] E. Berglund, M. Bång. Requirements for distributed user interface in ubiquitous computing networks. In Proc. of Conf. on Mobile and Ubiquitous MultiMedia, 2002.

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

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

[5] J.Rekimoto. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments Proceedings of UIST'97, pp. 31-39, 1997.

[6] R. Beale, W. Edmonson. Multiple Carets, Multiple Screens and Multi-Tasking: New Behaviours with Multiple Computers. In Proc. of HCI'2007, Bristish Computer Society, pp. 55-64, 2007.

[7] D. Tan, B. Myers, M. Czerwinski. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In Proc. of CHI'2004, ACM Press, New York, pp. 1525-1528, 2004.

[8] EMCC : Econometric Modeling & Computing Corporation http://www.speakeasy.com/

[9] M.W. Newman, S. Izadi, W.K. Edwards, J.Z. Sedivy, T.F. Smith. User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing. In Proc. of UIST'2002. ACM Press, new York, pp. 171-180, 2002.N.

[10] X.F. Qiu and N. Graham. Flexible and Efficient Platform Modeling For Distributed Interactive Systems. Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 29-34, July 2009.

[11] C. Damas, B. Lambeau, P. Dupont and A. van Lamsweerde. Generating Annotated Behavior Models From End-User Scenarios IEEE Transactions on Software Engineering, Special Issue on Interaction and State-based Modeling, Vol. 31, No. 12, pp. 1056-1073, 2005.

[12] S. Zaidenberg, O. Brdiczka, P. Reignier, J.L. Crowley. Learning context models for the recognition of scenarios IFIP International Federation for Information Processing of Artificial Intelligence Applications and Innovations pp. 86-97, 2006.

[13] O. Brdiczka, J.L. Crowley, P. Reignier. Learning Situation Models for Providing Context-Aware Services Lecture Notes in Computer Science pp. 23-32, 2007.

[14] The Mozart Programming System. http://www.mozart-oz.org/