A Reference Model for Distributed User Interfaces

Alexandre Demeure, Gaëlle Calvary, Jean-Sébastien Sottet
CLIPS-IMAG
BP53
38041 Grenoble Cedex 9, France
{alexandre.demeure, gaelle.calvary}@imag.fr

Jean Vanderdonkt
School of Management (IAG)
Catholic University of Louvain Place des Doyens, 1, B-1348 Louvain-la-Neuve, Belgium
vanderdonckt@isys.ucl.ac.be

ABSTRACT
This paper proposes a reference model for reasoning about different types of distributed User Interfaces (UI): mouldable, distributable, and migratable UIs. The reference model explicitly captures concepts subject to distribution at different levels of abstraction (tasks, concepts, abstract UI, concrete UI, and deployed UI) so as to provide designers with some guidance on deciding how to distribute the UI. Some significant types of distributed UI are uniformly expressed according to the reference model.

Author Keywords
Plasticity, .

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION

With ubiquitous computing, User Interfaces (UI) are no longer confined in a unique desktop. Instead, they may be distributed and migrate across a dynamic set of interaction resources that are opportunistically composed, borrowed and lent. As a consequence of distribution and migration, UIs must be mouldable in order to adapt gracefully to changes of the interactive space. Typically, in [14], when the user "picks" graphical tables and chairs on a vertical surface and drops them on a horizontal surface, then the rendering of the furniture switches from a 3-D to a 2-D representation. Today, mouldable, distributed and migratable (MDM) UIs are studied under the umbrella of *plasticity* [1]. But there is still no clear definition of the moulding, distribution and migration phenomena. Many terms appear, stressing the feeling of an ontological confusion: for instance, migratory UIs [1, 3], mutable [13], transformable [7], reconfigurable [6], retargetable UIs [4], composition and decomposition [18].
In addition to that confusion, a need exists for consolidating this very recent branch of research dedicated to plastic UIs. Whilst a reference framework provides a sound basis for reasoning about plasticity at design time [4], there is no tool for characterizing and comparing systems at run time. Typically, considering the *distribution* criteria, several systems exist such as I-AM [5], i-Land [16], Stanford Interactive Mural [8], Aura [15], ConnecTables [17, 18], Dygimes [18], DistriXML [7]. But so far, no state of the art of these systems has been conducted and no reference framework has been proposed to better perceive the differences between these terms and systems.

This paper proposes a reference model for reasoning about distributed UIs. It is based on A Meta-model of Interactive systems (AMIS) that describes an interactive system at different levels of abstraction ranging from tasks to deployment. AMIS is described in section 2. As explained in section 3, it provides a sound basis for reasoning about distribution. Some examples of functions and properties are elicited to characterize existing approaches in section 4. Further work includes the standardization of definitions (moulding, distribution and migration) based on AMIS (Sect. 5).

AMIS: A META-MODEL OF INTERACTIVE SYSTEMS

AMIS is a meta-model of interactive systems for describing the *morphology* of an interactive system at different levels of abstraction, ranging from the user tasks to its deployment. AMIS is a graph made of the physical and digital entities that are required for the definition and execution of the interactive system.
The *physical entities* (PhysicalEntity on Fig. 1) encompass the computing resources (CPU), output devices (called Outputers on Fig. 1, e.g, screens, loudspeakers) and input devices (called Inputers on Fig 1., e.g. keyboards, mice, cameras)) that are required for the execution of the interactive system.

The *digital entities* (DigitalEntity on Fig. 1) embody both the "components" of the interactive system and their mapping on the outputers (for their rendering) and inputers (for the user interaction). The "components" of an interactive system refer to either its functional core (DigitalFunctionalCore) or UI (DigitalUserInterface), both of them being connected through a connector (MappingUIFC). All of these elements (functional cores, UIs and mappings) are kinds of digital entities. Digital entities, inputers and outputers are managed by CPUs.

Space is a key notion in both the physical and digital worlds. It basically defines a coordinates space according to the mathematical meaning. Typically, a digital window defines a 2-D digital space which origin may be its top-left corner, and which axis x and y may be oriented to the right and bottom directions. Spaces may be confined inside boundaries, as a result defining *zones*. Typically, in Human-Computer Interaction (HCI), workspaces (also called interaction spaces or presentation units) are digital zones (DigitalZone) in which the interaction is supposed to take place. Symmetrically, outputers define physical zones (PhysicalZone). For instance, a screen may be seen as a parallelepiped which length, high and depth may be expressed in inches.

Fig. 1 presents an UML class diagram gathering the concepts that have been introduced. AMIS is based on these concepts.

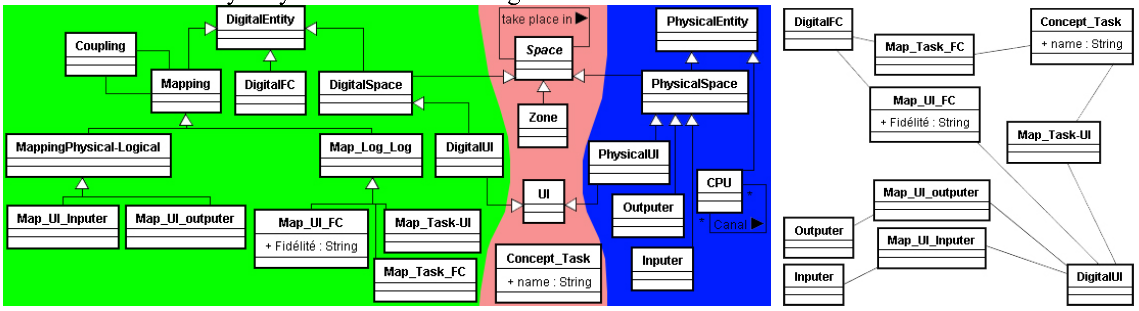


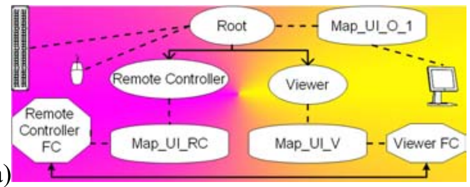
Figure 1 : Basic concepts useful for meta-modeling the reference model. Left side is about the digital entities, right side about the physical entities and the middle about conceptual entities. The far right side focuses on the mappings, making explicit the entities they are based on.

AMIS is a *graph* which:

- nodes are the physical and digital entities useful for the definition and execution of the interactive system. For a better legibility, we propose graphical notations (i.e., a concrete syntax) for differentiating entities: functional cores, UIs and mappings are respectively depicted by octagons ⯃, ovals ⬭ and cylinders ▭ . Outputers are represented by flat screens, inputers by keyboards or mice, CPUs by a surrounding box and other physical nodes by rectangles.
- the relations may be of different types: *"managed by"* for expressing the fact that a digital entity (DE) is managed by a CPU (in that case, the DEs are represented inside a colored box denoting the CPU) ; *"is defined with respect to"* for expressing the fact that a space *A* is defined with respect to a space *B* ($\boxed{A}\rightarrow\boxed{B}$) ; "*is functionally dependent of*" for expressing the fact that a DE is functionally dependent of another one (dependentEntity ▶ referenceEntity) ; "is mapped on" for expressing the fact that a DE is mapped on an other DE ($\boxed{}$ ----- $\boxed{\text{Mapping}}$ ----- $\boxed{}$) ; *"is replicated"* for conveying multiple instanciations of a same DE ($\boxed{\text{Numeric Entity}}$ -·-·- $\boxed{\text{Numeric Entity}}$).

This meta-model is instantiated in Fig. 2, on the CamNote case study. CamNote is a powerpoint-like software made of a slides viewer and a controller. The controller is able to run both on PC and PDA. PDA is an interesting option for controlling the presentation in a remote way. The graph of CamNote is modeled in two configurations: (a) the slides viewer and the controller on a same PC; (b) the slides viewer on PC and the controller on PDA.

- In both configurations, the functional core is made of two parts: the slides viewer (Viewer) and the remote controller (RemoteController FC). The RemoteController FC is either managed by the PC (a) or the PDA (b),
- The functional cores and their UIs are linked together through mappings (Map_UI_FC),
- In a), the desktop PC (Root) contains the slides viewer and the remote controller UIs. The desktop (root) is directly mapped on the PC screen: as a result, both of the components (slides viewer and remote controller) are displayed on the PC screen. In b), the desktop PC is limited to the slides viewer UI (Viewer), the remote controller UI being mapped on the PDA screen. As a result, the slides viewer appears on PC, whilst the remote controller is displayed on PDA.
- In b), PC and PDA are connected through a network.

(a)


(b)

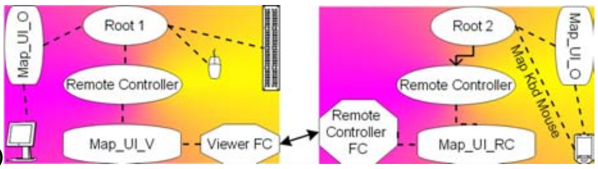


Figure 2 : The graph of CamNote when (a) both the slides viewer and the remote controller run on PC; (b) the slides viewer runs on PC whilst the remote controller runs on PDA. The models are instance diagrams of the class diagram presented in Fig. 1. The proposed graphical notations are used to improve the diagram legibility.

In practice, the INTERACTIVE SYSTEM can be analyzed in a finer way. Considering the UIs, the desktop may be seen as a digital space in which the content of the UIs is expressed.

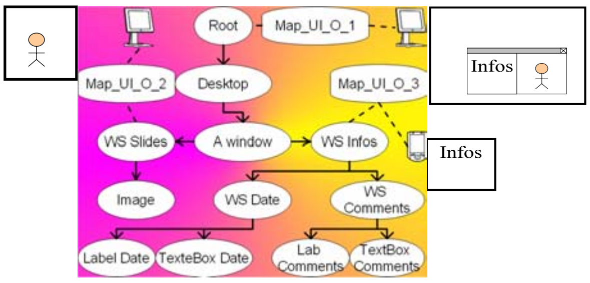


Figure 3 : The reference model is a powerful notion for reasoning about distribution. The way the UI is mapped on outputers defines the locations where the UI appears.

In Fig 3. the UI is made of a unique window (A Window) structured into two workspaces: "WS Slide" for the slides presentation and "WS Infos" for a notes editor. Recursively, WS Slide and WS Infos define digital spaces for expressing their content. They are expressed with regard to the window digital space. In Fig. 3, WS Slide is limited to an image, whilst WS Infos contains labels and text boxes for editing both the current date and few comments. The mapping on outputers defines the "location" where the UI is displayed. In Fig. 3, the full desktop is projected on screen1 (CamNote plus other applications); the slides viewer is displayed on screen 2 whilst the notes editor appears on screen 3.

Next section introduces a set of functions and properties that make operational this notion of reference model.

FUNCTIONS AND PROPERTIES

Many functions and properties may be introduced based on the reference model. In this paper, we focus on the functions and properties that are powerful for reasoning about MDM UIs. Functions and properties are defined, then "formalized" according to a pseudo-Z notation. They may be useful for defining distribution or migration whilst some are of general interest. This criteria is used for structuring the section.

Functions of general interest

A useful notion for MDM UIs is the notion of *location*. Informally, the location of a digital entity expresses its *embodiment* in terms of physical and digital entities. More formally, the location of a digital entity *e* is the set of quadruples <*CPU*, *smo*: set of MappingUI-O, *smi*: set of MappingUI-I; *p*: path of DigitalEntity > where:

- *p* is a path in the graph going from *e* to *CPU*,
- *smo* is the set of mappings to outputers involved in *p*,
- *smi* is the set of mappings to inputers involved in *p*,
- *CPU* is the CPU managing *p*.

For instance, let us consider the "TextBox Date" in Fig. 3. Its location is based on a unique path *p* made of the following digital entities: TextBox Date-WS Date-WS Infos-A Window-Desktop-Root. Two mappings are involved (on screens 1 and 3) giving rise to the following location: {<CPU, {mapping screen 1, mapping screen 3}, {}, p>}. From now on, to make a clear distinction between a quadruple and the set of quadruples, a quadruple will be said *elementaryLocation*.

Location is a powerful notion for reasoning in HCI. Typically, if the set of mappings on outputers is empty, then the entity can not be observable [10]. "Let us formally define some functions of general interest. They apply on a graph.

- Replicas(*e*: DigitalEntity) computes the set of digital entities that replicate the entity e: **{j: DigitalEntity | Replicas (*e*, j)∨Replicas (j, *e*)** } where Replicas(i, j) is true if I is a replicas of j.,
- Locations(sui : set of UIs) computes the set of locations of sui.
- GetOutputers(*sui*: set of UI) computes the set of outputers where *sui* could be rendered. **∪{l: Locations(*sui*) • l.smo}**
- GetInputers(sui : set of UI) computes the set of inputers that provide input to sui. **∪{l: Locations(*sui*) • l.smi}**
- GetCPUs(sui : set of UI) computes the set of CPUs managing one ui among sui at least. **{l: Locations(*sui*) • l.cpu}**
- LocationsWhereUIsRendered(sui : set of UI) computes the set of locations where sui is rendered. **{<cpu, smo, smi, p>: Locations(*sui*) | smo'⊂ smo ∧ (∀o:smo'•IsRenderedOn(p,o)) • <cpu,smo',smi,p>}**
- OutputersWhereUIsRendered(sui : set of UI) computes the set of outputers where sui is rendered. **∪{l: LocationsWhereUIsRendered (sui) • l.smo}**
- LocationsWhereTasksRendered(st : set of tasks) computes the locations where st is rendered. It provides a higher level of information about the interactive

system.

sui = ∪{m: {mt:Map_Tasks-UI | m.st∩st≠∅}•m.sui} → LocationsWhereUIsRendered(sui)

Based on some of these functions, next sections introduces a new function, valuable for reasoning about distribution

Useful functions for reasoning about distribution

Distribution has several meanings among which one is about scattering. We define the *Are_X_Scattered* for measuring the extent to which extent a set of digital UIs is scattered according to an X criteria: CPU, screens, etc. The degree of scattering may be null, weak, medium or strong. Let us examine this range of values on an example (the screen scattering) before providing general formalizations.

- Strong scattering means that there is no screen shared by two digital UI,
- Medium scattering means that there is no digital UI using exactly the same screens than another one,
- Weak scattering means that, at least, two digital UIs differ in the screens they use.

For example, the set of entities "Image" and "Label comments" on Fig. 3 is medium scattered. Indeed:

- Image is mapped on two outputers {screen 1, screen 2},
- Comments is mapped on two outputers {screen 1, screen 3},
- Screen 1 is shared by the two entities, scattering is not strong.

More formally;

***Are_X_Scattered* (sui: set of UI;**
fct: (set of X) f (sui: set of UI))
ssx: set of <DigitalEntity, set of X> |
ssx={ui: sui • <ui, fct({ui})>} • α where:

- sui is the set of digital UIs which scattering is studied. For instance, { Image, Label comments },
- fct(sui) returns the set of X involved in the location of sui. For instance, {screen 1, screen 2} for the ui Image and X = "screen",
- ssx is the set of couples <ui, getX(ui)>, obtained by considering all the uis of sui. For instance, {<Image, {screen 1, screen 2}>, <Label comments, {screen 2, screen 3}>},
- α expresses the strength of the scattering. It may be:
  - *Strong:* α ≡ (¬∃s1,s2: SSX | s1.sx∩s2.sx≠∅)
  - *Medium:* α ≡ (¬∃s1,s2 : SSX | s1≠s2 • s1.sx=s2.sx)
  - *Weak:* α ≡ (∃s1,s2 : SSX | s1.sx≠s2.sx).

Next section introduces functions and properties operating on a set of reference models. These functions and properties are useful for reasoning about migration.

Useful functions for reasoning about migration

Many functions and properties could be defined. As an example, let us introduce the *LeaveAndGet* function that elicits the elementary locations a digital entity leaves and conversely gets when arriving in a new graph. *LeaveAndGet(sui: set of DigitalUI; G1, G2 : Graph)* returns a couple <lost, won> where *lost* and *won* respectively denote the elementary locations the entity has lost and won. More formally: L1: G1.location(sui), L2: G2.location(sui)

<{l1: L1 | l1=<cpu1, sm1, w1>
∧ (¬∃l2: L2 | l2=<cpu1,sm2,w1>∧sm1⊆sm2)• l1}
, {l2: L2 | l2=<cpu2, sm2, w2>
∧ (¬∃l1: L1 | l1=<cpu2,sm1,w2>∧sm2⊆sm1)• l2}>.

Next section applies most of functions and properties on case studies extracted from the literature.

ILLUSTRATIONS ON THE STATE OF THE ART

The state of the art is structured according to the what (UIs or tasks) and where (CPU, outputers, UIs), the distribution or migration is performed.

Distributing UIs on UIs

A typical example is multiple views. Multiple views refer to the visualization of a UI on different UIs. Fig. 4 is an example of an image factorized on two windows. This notion can be formalized in the following way:

Is_UI_distributed_on_UIs (ui : UI)
L : Locations(ui)
→ #L≠0

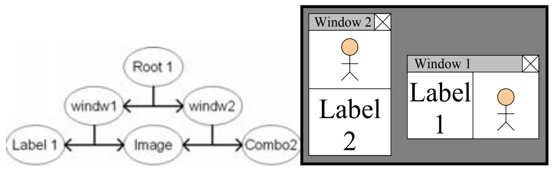


Figure 4 : Image is distributed on window 1 and window 2.

Migrating UIs on UIs

A typical example is the transfer of a part of the content of a window to another window. Fig 5 illustrates the transition of an image passing from window 1 to window 2. This notion can be formalized in the following way:

Migration (ui : UI; G1, G2 : graph)
<leave, get> : LeaveAndGet({ui}, G1, G2)
→ (#leave≠0)∧(#get≠0)

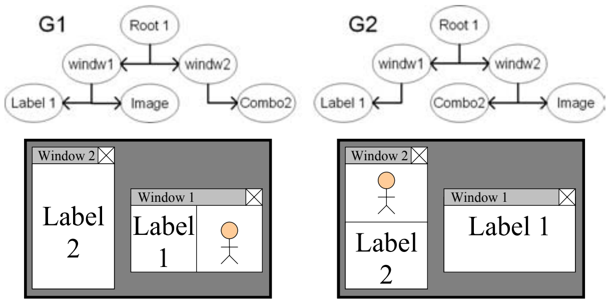


Figure 5 : Image migrate from window 1 to window 2.

Distributing UIs on outputers

A typical example is "multiple screens". Multiple screens refers to the addition of screens for extending the display surface. The UI is not changed, but it can be mapped in different ways. Fig. 6 models the availability of a second screen. The mapping on this new outputer can be performed in different ways, therefore providing different kinds (and feelings) of distribution. Distributions may range from a full replication (exactly, the same rendering on both screens) to a strong scattering (one part of the UI on one screen, the rest on the second one). If we mean by distributed that a UI is rendered on different outputers then this notion can be formalized in the following way:

Distributed_UI_O(sui : set of UIs)
$\rightarrow$ #OutputersWhereUIsRendered(sui) >1

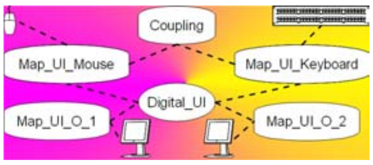


Figure 6 : An example of graph involving multiple screens.

Migrating UIs on outputers

This can be understood in at least two ways; The UI becomes ***renderable*** on another outputer or the UI becomes ***rendered*** on another outputer. Let's formalize the two cases.

UI become displayable on another UI.

Migration_UI_O_1(ui : UI ; G1, G2 : Graph)
SO1 = G1.GetOutputers({ui})
SO2 = G2.GetOutputers({ui})
$\rightarrow$ <SO1 \ SO2 ; SO2 \ SO1>

UI become displayed on another UI.

Migration_UI_O_2(ui : UI ; G1, G2 : Graph)
SO1 = G1.OutputersWhereUIsRendered({ui})
SO2 = G2.OutputersWhereUIsRendered({ui})
$\rightarrow$ <SO1 \ SO2 ; SO2 \ SO1>

Distributing/Migrating UIs on CPUs

Typical cases of distributed/migrated Uis on CPUs are I-AM and Beach

I-AM (Interaction Abstract Machine) [12] is a platform manager for MDM UIs. It supports the dynamic configuration of interaction resources to form a single logical interactive space. These resources can be managed by different elementary workstations running distinct operating systems (i.e., MacOS X, Windows NT and XP). Users can distribute and migrate user interfaces at the pixel level as if these UIs were handled by a single computer. This illusion of a unified space is provided at no extra cost for the developer who can re-use the conventional GUI programming paradigm.

The underlying principle of I-AM is to create the UI on an I-AM server, and replicate it on each I-AM client. Fig. 7 makes this principle explicit thanks to the reference model. The universe embeds a model of the topology of the surfaces.

In I-AM, an UI is said distributed if it is displayed on at least two surfaces managed by different CPUs. This can be expressed by the following *Distribution(ui: DigitalUI)* function. It returns the set of elementary locations of *ui* that involve different CPUs. The UI will be said distributed if this set is not empty.

L=LocationsWhereUIRendered (ui))
$\rightarrow \exists$ l1, l2 : L | l1.cpu$\neq$l2.cpu)

In I-AM, an UI is said to migrate when it leaves a set of screens S1 to target a set of screens S2, such as $S1 \cap S2=\varnothing$. This can be expressed by the following:

Migration(ui1,ui2: DigitalUI; G1, G2: Graph)
L1 = G1.LocationsWhereUIsRendered({ui1})
L2 = G2.LocationsWhereUIsRendered({ui2})
$\rightarrow (L1\neq\varnothing) \wedge (\forall l2: L2 \bullet \neg(\exists l1 : L1 | l1.cpu=l2.cpu))$

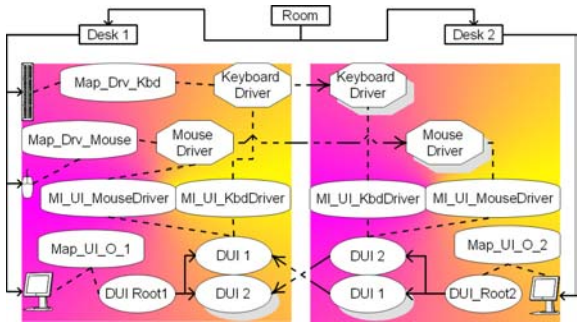


Figure 7 : An I-AM characterization based on reference model. The UI is created on a server (CPU2) and replicated on each client (only one in the figure 1 - CPU 1).

Another example of UI distributed or migrated on CPUs is given by Beach. Beach [16] has been developed in order to support roomware application. In addition to a local object space on each platform containing all non-distributed object, Beach is based on COAST for the management of shared objects. Fig. 8 characterizes Beach on the Wall&Chair example. The Wall&Chair roomware is composed of a wall-screen (DW for DynaWall) and a tablet-PC on a chair (CC). Both render the same application but the tablet-PC also renders another one, making it possible for the user to have a private space.

The distribution occurring in beach consists in the replication of shared elements on a server. The distribution is effective when a UI is replicated on different CPUs, witch can be formalized by the following function Distribution_Beach_UI_CPU (sui : set of UIs) :

#getCPUs(sui) > 1

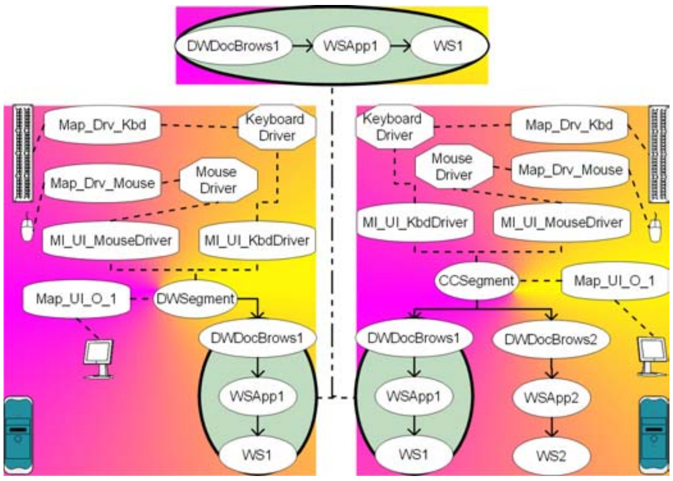


Figure 8 : Beach illustrated on the Wall&Chair case study.

Distributing Task on outputs/CPU

Knowing which tasks of an interactive system are supported by which CPUs and/or outputers provides a high level of knowledge of the deployment of the INTERACTIVE SYSTEM [20]. An example is given in Fig 9: the task model of a basic text editor is deployed on two CPUs. One of them achieve the tool selection whilst the other one achieves tool application and text edition. The task "Editing textual document" is then distributed on this two CPUs. Knowing weather a set of tasks is distributed on outputers can be formalized in the following way:

AreTasksDistributedOnOutputers (st : set of Tasks)
smUI = {m : MapTasksUI | (∃t : st | t∈m.Tasks) • m}
sUI = ∪{m : smUI • m.UIs}
so = ∪{ui : sUI • OutputersWhereUIsRendered({ui})}
→ #so > 1

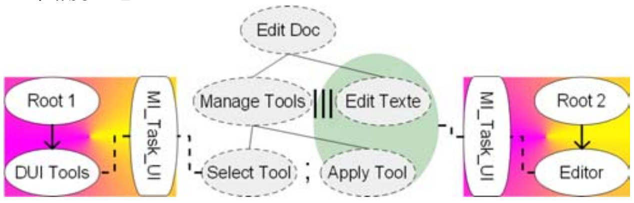


Figure 9 : Tasks distributed on different computers.

Migrating tasks on another CPU/UI

A typical example of this kind of migration is Aura. In Aura [15], the adaptation to the context of use is performed by substituting one application (e.g., a text editor) by another one. Fig. 10 provides an example based on a text editor. The user edits a text using word on his work's computer (G1). He then decides to go back home. When arriving in front of his home's computer, the system launches emacs, which is the equivalent software he has to perform text edition (G2). We can notice that in G1 the task distribution on the UIs is more precise than in G2. However the whole task tree is mapped to UIs in both G1 and G2. Such a migration of tasks from/to UI can be formalized in the following way:

MigrationAura(st: set of tasks; G1, G2: Graph)
sm1 = {m: G1.sm | st∩m.st≠∅ • m}
sm2 = {m: G2.sm | st∩m.st≠∅ • m}
→ {m1:sm1, m2:sm2 | m1.st=m2.st ∧ m1.sui≠m2.sui • <m1.st, m1.sui, m2.sui>}

This function returns a set of triples <st, sui1, sui2> where st is the set of tasks being migrated form sui1 in G1 to sui2 in G2.

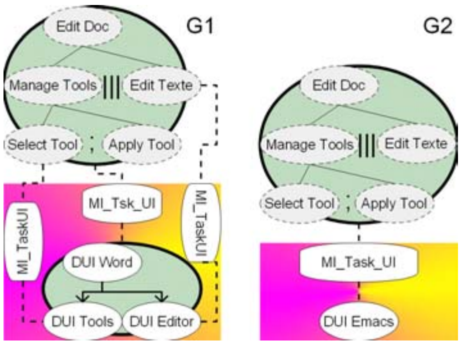


Figure 10 : The change of graph in Aura: Word is substituted by Emacs.

Distributing Inputs

Distributing inputs could mean several things. In I-AM, "distributing inputs" means replicating every input on every CPU. Thus it makes it possible to control a window of a CPU with the mouse linked to another CPU.(Fig. 11)

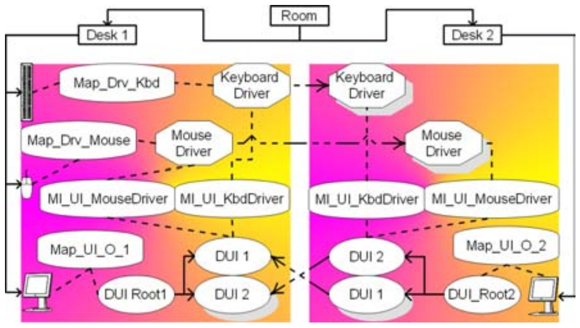


Figure 11 : Exemple of distribution of inputs in IAM. Each inputer is replicated on every CPU.

In Synergy, "distributing inputs" means that every input device of the same type (e.g. mouse, keyboards…), controls the same "logical input". For example, this system makes it possible to control the mouse cursor of a PC with a mouse device connected to a MAC. PC cursor and MAC cursor are superposed, giving thus the illusion that it is the same. (Fig 12.)

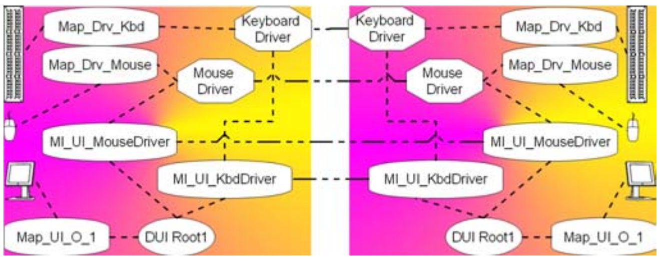


Figure 12 : Example of distribution in Synergy; Keyboard and mouse devices are respectively connected to the same conceptual keyboard and mouse driver.

Migrating Inputs

Migrating inputs can mean making possible for an input device to act on a UI managed by another CPU than the one managing the device. Fig. 13 shows an example. In G1, the keyboard is managed by CPU1, and acts on Root 1 only. In G2, by transferring the keyboard driver information to Root1 and Root2, the keyboard migrates to CPU2.

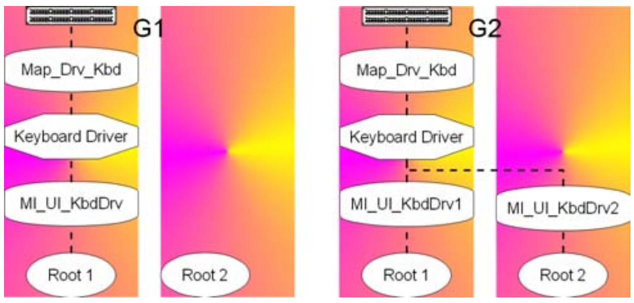


Figure 13 : Exemple of a migration of a keyboard to another computer

5 CONCLUSION AND PERSPECTIVES

This paper introduces a reference model as a powerful tool for reasoning about different kinds of distributed UIs. Based on a set of examples, it shows how this reference framework could help in formalizing different kinds of distribution and migration. The claim of the paper was not to introduce new definitions, but to show how this notion helps in characterizing existing systems, and envisioning the problem space of MDM. One perspective of the work is of course the standardization of the definitions.

ACKNOWLEDGMENTS

We thank the Network of Excellence SIMILAR for its support.

REFERENCES

1. Balme, L, Demeure, A., Barralon, N., Coutaz, J., Calvary, G. CAMELEON-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces, Lecture Notes in Computer Science, Volume 3295 / 2004, Ambient Intelligence: Second European Symposium, EUSAI 2004, Markopoulos P., Eggen B., Aarts E. *et al.* (Eds), Springer-Verlag Heidelberg (Publisher), ISBN: 3-540-23721-6, Eindhoven, The Netherlands, November 8-11, (2004) 291-302
2. Bandelloni, R., Paternò, F.: Flexible Interface Migration. In: Proceedings of ACM Con. On IUI'04 (Funchal). ACM Press, New York (2004) 148–155
3. Bharat, K.A., Cardelli, L.: Migratory Applications Distributed User Interfaces. In: Proc. of ACM Conf. on User Interface Software Technology UIST'95. ACM Press (1995) 133–142
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers 15, 3 (June 2003) 289–308
5. Coutaz, J., Balme, L., Lachenal, Ch., Barralon, N. Software Infrastructure for Distributed Migratable User Interfaces. In: Proc. of UbiHCISys Workshop on UbiComp (2003)
6. Grolaux, D., Van Roy, P., Vanderdonckt, J.: FlexClock, a Plastic Clock Written in Oz with the QTk toolkit. In: Proc. of 1st Int. Workshop on Task Models and Diagrams for user interface design TAMODIA'2002 (Bucharest, 18-19 July 2002). Academy of Economic Studies of Bucharest, INFOREC Printing House, Bucharest (2002) 135–142

7. Grolaux, D., Van Roy, P., Vanderdonckt, J.: Migratable User Interfaces: Beyond Migratory User Interfaces. In: Proc. of 1st IEEE-ACM Annual International Conference on Mobile and Ubiquitous Systems: Net-working and Services MOBIQUITOUS'04 (Boston, August 22-25, 2004). IEEE Computer Society Press, Los Alamitos (2004) 422–430
8. Guimbretière, F., Stone, M., Winograd, T.: Fluid Interaction with High-resolution Wall-size Displays. In: Proc. of ACM Conf. on User Interface Software Technology UIST'2001
9. Han R., Perret V., Naghshineh M., WebSplitter: A Unified XML Framework for Multi-Device Collaborative Web Browsing, Appeared in ACM Conference on Computer Supported Cooperative Work (CSCW) (2000)
10. IFIP BOOK: Design Principles for Interactive Software, Gram C. and Cockton G. (eds), Chapman & Hall, (1996)
11. Lachenal, C., Rey, G., Barralon, N. MUSICAE, an infrastructure for MUlti-Surface Interaction in Context Aware Environment. In Proc. *HCI International*, Crète, June (2003) 125-126
12. Lachenal, C.: Models and tools for multi-instrument and multi-surface interaction, PhD thesis of the University Joseph-Fourier, Grenoble I, Informatics (2004)
13. McKinley, P.K., Masoud Sadjadi, S., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer (July 2004) 56–64
14. Rekimoto, J.: Pick and Drop: A Direct Manipulation Technique for Multiple Computer Environments. In Proc. of UIST97, ACM Press, (1997) 31-39
15. Sousa, J., Garlan, D.: Aura : an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In: Proc. of IEEE-IFIP Conf. on Software Architecture (2002)
16. Streitz, N. *et al.*: i-LAND: An interactive Landscape for Creativity and Innovation. In: Proc. of ACM Conf. on Human Factors in Computing Systems CHI'99 (1999) 120–127
17. Tandler, P.: Software Infrastructure for Ubiquitous Computing Environments: Supporting Synchronous Collaboration with Heterogeneous Devices. In: Proc. of Conf. on Ubiquitous Computing'01. LNCS vol. 2201. Springer-Verlag, Berlin (2001) 96–115
18. Tandler, P. Prante, Th., Müller-Tomfelde, Th., Streitz, N., Steinmetz, R.: ConnecTables: Dynamic coupling of displays for the flexible creation of shared workspaces. In: Proc. of 14th ACM Symp. on User Interface Software and Technology UIST'01. ACM Press, 11–20
19. Vandervelpen, Ch., Coninx, K., Towards Model-Based Design Support for Distributed User Interfaces. In: Proc. of NordiCHI'2004. ACM Press, New York (2004)
20. Vandervelpen, Ch., Vanderhulst, K. Coninx, K., Light-weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments, accepted for the 5th International Conference on Web Engineering (ICWE'2005), 25-29 July 05 Sydney, Australia.