Implementation-agnostic instantiation schemes for ubiquitous, synchronous multi-user interfaces

Dimitrios Kotsalis
Dept. of Informatics Engineering, Technological Education Institute of Crete, Heraklion, Crete, Greece
+302810379892
dim.kotsalis@gmail.com

George Vellis
Dept. of Informatics Engineering, Technological Education Institute of Crete, Heraklion, Crete, Greece
+302810379892
g.vellis@hotmail.com

Demos Akoumianakis
Dept. of Informatics Engineering, Technological Education Institute of Crete, Heraklion, Crete, Greece
+302810379190
da@ie.teicrete.gr

Jean Vanderdonckt
Louvain Interaction Lab
Université Catholique de Louvain, Belgium
+32(0)10478525
jean.vanderdonckt@uclouvain.be

ABSTRACT
The paper describes an engineering approach for building user interfaces for synchronous peer co-engagement in virtual work by operating *with*, *on*, *through* and *within* different digital representations. The proposed approach introduces several extensions to the UsiXML family of models as well as design and runtime support for multi-platform, synchronous and collaborative interactions. We demonstrate the key concepts by elaborating a scenario of collaborative co-play of a soccer game. The specific example features synchronous co-play by remote users in different roles (players or observers), a range of devices (PCs or Android) and interaction facilities (visual and non-visual).

Categories and Subject Descriptors
H.5.2 [Information Interfaces and Presentation]: User Interfaces

General Terms
Design, Human Factors, Languages

Author Keywords
Multi-user interfaces; User interface description languages

1. INTRODUCTION
User interface (UI) software and technologies have had a major contribution towards novel and ubiquitous computing environments. As a result, a range of UI engineering practices have been established and continue to attract research attention. This paper is concerned with three major challenges. Firstly, the increasingly collaborative settings complicate UI engineering as they make compelling the need to support novel design affordances such (various forms of) awareness, translucence and plasticity. Secondly, established engineering practices appear to be at odds with creative designs that implicate non-native and custom widgets, as in the case of serious games. Thirdly, due to variations in the target interaction vocabularies, automatic UI generation is still limited to certain types of UIs.

Attempts to address these challenges traditionally impose constraints on the type of synchronous and cross-platform activities or the range of available interaction resources. There is also a genuine lack of efforts targeted to synergistic affordances of different media types (e.g., sounds and 3D audio for augmenting visual interaction) and their exploitation in complex situations of virtual work. The term 'virtual work' may be conceived through different lenses. For the purposes of this research, we do not confine virtual work as being work with a digital representation of the physical rather than with the physical itself. Instead, we commit to the line of research that emphasizes additional aspects such as what is actually virtualized (i.e., people, artifacts and/or processes), what affordances are to be inscribed in digital representations (i.e., awareness, translucence, plasticity) and how these may be manifested through the UI. Consequently, we are concerned with UIs which imbricate representations – that stand or completely substitute for physical referents – to create novel user experiences, such as in the case of simulations and games. In these examples, people interact with each other, but from very different settings, using very different UIs, and fulfilling different roles.

To this effect, we review prominent UI engineering practices and propose a method and a set of tools that extend Model-based User Interface (MBUI) engineering by articulating the concept of implementation-agnostic specification of UIs. Although the underlying goal is widely acknowledged [1] [12], the results today suffer from the shortcomings briefly outlined earlier. The rest of the paper is structured as follows. The next section reviews relevant streams of research in UI engineering and motivates the present research. Then, we present a motivating scenario and outline the premises of implementation-agnostic UI instantiation. Following this, the engineering approach and the run-time implementation of the collaborative soccer game are elaborated. The last section consolidates results and points to future research.

2. BACKGROUND AND RELATED WORK
Managing diverse widgets has been addressed through various streams of research. One method is polymorphism which refers to run-time handling of alternative interaction components. The method was initially proposed and implemented in the Platform Integration Module [2], and subsequently, in the HOMER UIMS [3]. In both these cases, polymorphism was conceived as a language construct inscribed in toolkit-based implementations aiming to address specific accessibility challenges (i.e., access to visual and non-visual UIs). Other researchers have embarked in similar efforts, although using different terminology. For instance, Meta-widgets [4] were introduced as components on top of implementation-specific toolkits and were applied for building adaptive multi-modal UIs. The key concept relied on architectural styles for encapsulating alternative object classes into widget abstractions. However, their implementation assumed non-extensible instantiation schemes, while provisions for multi-user aspects and novel affordances (i.e., translucence, awareness,

PCI '14 , Oct 02-04 2014, Athens, Greece
ACM 978-1-4503-2897-5/14/10.
http://dx.doi.org/10.1145/2645791.2645825

plasticity, etc.) were completely dismissed. In [5] and its ancestor [6] widget management was approached by separating alternative input and output mechanisms from the 'common' behavior supported by a widget. Nevertheless, as in the case of Meta-widgets, synchronous and collaborative aspects are bypassed.

Collaboration-specific issues are focal concerns in Groupware toolkits. MAUI [7] is an early effort that exploits toolkit-level sharing to provide an extended set of groupware widgets (i.e., multi-user scrollbar, menus, etc.) with native support for group awareness inscribed in the widgets' dialogue. Again, the approach lacks support for heterogeneous contexts of use since it makes assumptions about the underlying platform or toolkit. An exception is [8] where a factory-based programming model for 'multi-domain' collaboration toolkits is presented. The authors describe a system for the stand-alone, Eclipse, and web domains; and the AWT, Swing, SWT, and GWT single-user toolkits associated with these domains. Worth mentioning is also Fiia, a novel architectural style for adaptive groupware [12] which however, focuses on building distributed collaborative applications using ad-hoc mappings to a groupware toolkit.

MBUI engineering advocates UI specification using models and model transformations. The COMMETS Framework [9] consolidates recent practice and experience. Nevertheless, the approach dismisses user roles, session management, replication and awareness. Moreover, it offers no tool support, thus by passing aspects related to low-level issues such as managing diverse collections of objects, distributed class loading, dependency libraries, specific toolkit-level instantiation instructions, pre-instantiation configuration of widgets, etc. On the other hand, it does introduce a new development workflow. A particular stream of research within the broad venue of MBUI engineering has focused explicitly on collaborative interaction. Examples include FlowiXML [10], CIAM [11] and TOUCHE [16]. These efforts concentrate primarily on devising notations and tools to model cooperative behavior and workflows. Their primary contribution is that they make explicit different elements of collaboration (i.e., roles, responsibilities and tasks) using dedicated notations.

From the discussion thus far, it becomes evident that toolkit-based techniques and MBUI engineering methods focus on different issues. Toolkits are programming-intensive but can offer rich interaction facilities especially through non-native and semantic widgets [13]. MBUI engineering relies on models for specifying interaction and model transformations for mapping abstractions to concrete UIs. The present research attempts to establish the 'connective tissues' that would make possible the appropriation of the benefits each may offer in the context of synchronous, collaborative and distributed UIs.

3. METHOD

3.1. Motivating scenario

To illustrate the challenges, an example of synchronous, collaborative and distributed co-play of the soccer game is used as reference scenario. The setting entails multi-player co-engagement using different terminals and interactive representations. To anchor 'multi-player' co-engagements we assume not only users in different roles (i.e., players versus observers with limited access rights) but also users with different capabilities (i.e., sighted versus non-sighted users). Platforms may also vary to include Android devices and conventional desktops equipped with appropriate resources for non-visual interaction. Consequently, soccer game co-play is an example of 'virtual' work where users operate predominately *through* and/or *within* representations. Operating *through* representations entails interactive embodiments that stand for or completely substitute physical referents (people or objects). Operating *within* representations assumes a sort of 'merging' different representations to create virtualities with no single or solid physical referent.

3.2. Implementation-agnostic instantiation

Our approach to address the challenges outlined above is grounded on implementation-agnostic instantiation schemes that serve the purpose of managing alternative interactive embodiments suited for designated users and contexts of use. Implementation-agnostic instantiation schemes rest on (a) certain design affordances that are explicitly modeled; (b) a specification that details how these affordances are supported in a certain vocabulary and (c) instantiation schemes that generate instances of concrete widgets given the behavior designated in their specification. According to this concept the UI is assembled from diverse interaction components (which need not be available through a single toolkit library or interaction platform). For this to be viable, abstractions are compelling to 'hide' intrinsic low-level technicalities. Widget Specification Language (WSL) forms such an abstraction scheme tasked to designate not only the widgets' look & feel but also the range of alternative possible manifestations and the mechanics of 'linking to' rather than directly 'calling' toolkit libraries.

Referring to our reference scenario, it is worth recalling that the multi-party soccer game implicates distributed co-play between users with different rights (players versus audience), capabilities (sighted versus non-sighted), devices (i.e., desktop PCs and mobile) and platforms (i.e., Swing, non-visual toolkit and Android). In an effort to keep our discussion in focus, we bypass implementation details and assume that appropriate facilities are available for building embodiments of soccer game as indicated in Figure 1.

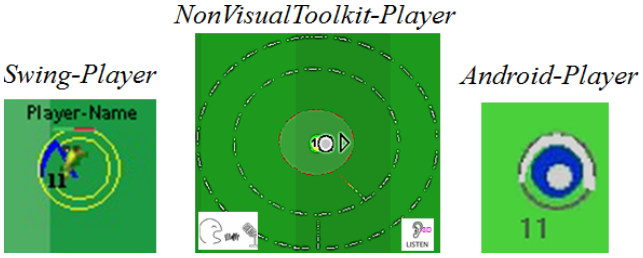

Figure 1: Platform-specific instantiations of players

It is worth noticing that these instances vary with respect to physical / lexical attributes as well as syntax / dialogue and semantics. For instance, in the case of desktop sighted user settings, 'self-awareness' is manifested through a visual proxy conveying the player's stamina. In other settings, the same affordance may necessitate quite different design decisions (i.e., use of audio cues for the non-visual setting) or be dismissed due to space constraints (i.e., mobile/Android setting). It is also possible to envision more complex affordances. One such case is 'situation awareness'. For sighted users situation awareness is facilitated by immediate and direct space perception. As non-sighted users lack this capability, situation awareness becomes critical for co-engaging in a highly volatile environment, such as the game. One solution for conveying spatial information is to instantiate the non-visual player augmented with three concentric circles that help blind users discover points of interest (i.e., enemy or friendly players, the ball, certain lines, etc.) in their surrounding virtual

space. However, this entails platform-specific implementation that may not be needed or comparable to the implementation of players being instantiated on conventional desktop or Android devices. Making these details transparent is the core concern of implementation-agnostic instantiation schemes. One way to unify alternative device-specific incarnations is to devise abstractions that qualify how each alternative is interactively manifested (in a certain setting) and how it invokes whatever design affordances have been assigned. This implicates a design focus on how certain objects are engineered to facilitate designated design affordances such as self and situation awareness, translucence, plasticity, etc. Then, these affordances and the properties though which they are exposed to designers drive subsequent engineering activities.

4. ENGINEERING APPROACH

Implementation-agnostic instantiation in the context of our reference scenario rests on two commitments. Firstly, players should be defined as abstract objects possessing minimal properties (i.e., label) and a set of qualifiers that designate affordances (i.e., self-awareness, situation awareness, translucence, synchronization, etc.) inscribed in the three different platforms. Secondly, specification schemes can be used to 'link' abstract players with their platform-specific embodiments without making direct calls to the platform libraries. The engineering approach to address these targets entails (a) extensions of UsiXML[14] to accommodate a Widget Specification Workflow and (b) design and run-time support for handling interaction resources for radically different contexts of use.

4.1. Widget Specification Workflow

The Widget Specification Workflow designates a protocol for appropriating diverse collections of objects, either native or custom. It relies on a Widget Specification Language (WSL) [15] that allows our models and tools to integrate and utilize third party widget libraries. The WSL extends traditional CUI models in so far as it adopts a meta-model that accommodates semantics of a collection of widget archives. This is in contrast with the older UsiXML's CUI specification where syntactic and semantic rules are hardcoded as part of language's formal definition (i.e., explicitly defining elements such as button, label, etc.). Another constraint imposed by the UsiXML CUI specification is that each CUI-element should possess exactly the same range of physical attributes; those common across most popular platforms. For instance, a button CUI element should always define a value for the width and height attributes. While this generalization is convenient for typical rectangular buttons, it constrains alternative possible instantiations of buttons such as round or circular button where instead of width and height we need a radius attribute. The problem is further revealed when considering our reference scenario where non-visual access is also required.

Figure 2 (top) presents the current version of the WSL. It is this specification that comprises the range of enabled polymorphic instantiations depending on the physical (context-specific) libraries available. Figure 2 (bottom) depicts an instance of the design tool which is used to populate the interaction elements of a UI based on their designated WSL properties. As shown, the WSL makes provisions for encoding properties for each different widget as well as their inscribed affordances. In this manner abstract widgets may be assigned alternative interactive behaviors, not only at the physical/lexical but also the syntactic and semantic levels. This version of the WSL has been used for specifying the instantiations of the 'Player Artifact' widget for Swing, the non-visual toolkit and Android (see Figure 1). Noticeably, they all share the same model specification but for each case the specification designates different affordances. The Swing-player is instantiated with a visual proxy for stamina. The non-visual toolkit player is instantiated with radars intended to facilitate social awareness and space perception. In the WSL (Figure 2, top) these affordances are declared as polymorphic properties of the abstract object 'Player Artifact'. Then, implementation-agnostic instantiation of the object is achieved by (a) allowing alternative CUI elements to inherit design affordances designated in the corresponding instance in the WSL and (b) linking to the library-specific resources that support each property. Thus, properties such as self-awareness, social-awareness, translucence and indeed any other abstract quality may be simply utilized through a standard instantiation scheme irrespective of how they may be implemented in a target library. This 'separation logic' can support interactive behaviors not viable with existing MBUI.

4.2. Models and primitives

4.2.1. Behavior model

Apart from specifying abstract widgets and their properties, it is also important to devise behavioral models to capture dynamic aspects of interaction. UsiXML does not make such provisions, except for simple dialog transitioning on button press and direct calls to designated domain model functionality. To alleviate this shortcoming, the behavior model makes explicit the desirable states of each different abstract widget. The 'states' of abstract widgets in the behavior model should not be confused with the states of concrete interaction elements as implemented in the various libraries or toolkits. In fact, our 'states' are model elements with properties that dictate the transitions afforded as well as the way in which these transitions are triggered by library- or toolkit-specific intrinsic code. Thus for instance, an abstract button may be declared to have as many states as required (including standard states such as 'pressed' or 'released' but also case-specific such as 'initiation', 'interim' and 'completion'). Transitions from one state to another, may then implicate radically different procedures across libraries and or toolkits. Finite State Machines (FSM) as reported in [17] allow designers to define custom states and to associate alternative behaviors to different object instances as needed bypassing low-level event listener classes. This provision allows for a high level synchronization of widgets that support alternative input mechanisms and transitions. For example, in our scenarios it is made possible for a Swing-player sensitive to 'mouseDown' input behavior to be synchronized with a touch sensitive Android-player with a corresponding 'onTap' input mechanism. Such synchronization can be easily implemented using FSMs on the grounds of the common states supported (i.e., focused, unfocused), thus completely ignoring the local widget transitions which are typically sensitive to alternative input mechanisms. This allows not only for relaxed-coupling between distributed users, but also for more advanced behavior modeling. In light of the above, our behavior model may comprise several FSMs per polymorphic instance while transitions supported by each are codified in the instance's section in the WSL specification (see Figure 2, top).

4.2.2. Shared abstraction model

In addition to synchronizing alternative behaviors, it is also important to unify alternative interactive instances across different settings (i.e., distributed or collocated) using shared model abstractions. This is because synchronization based on 'abstract' states (i.e., common across polymorphic instantiations of an 'abstract widget') may not be sufficient.

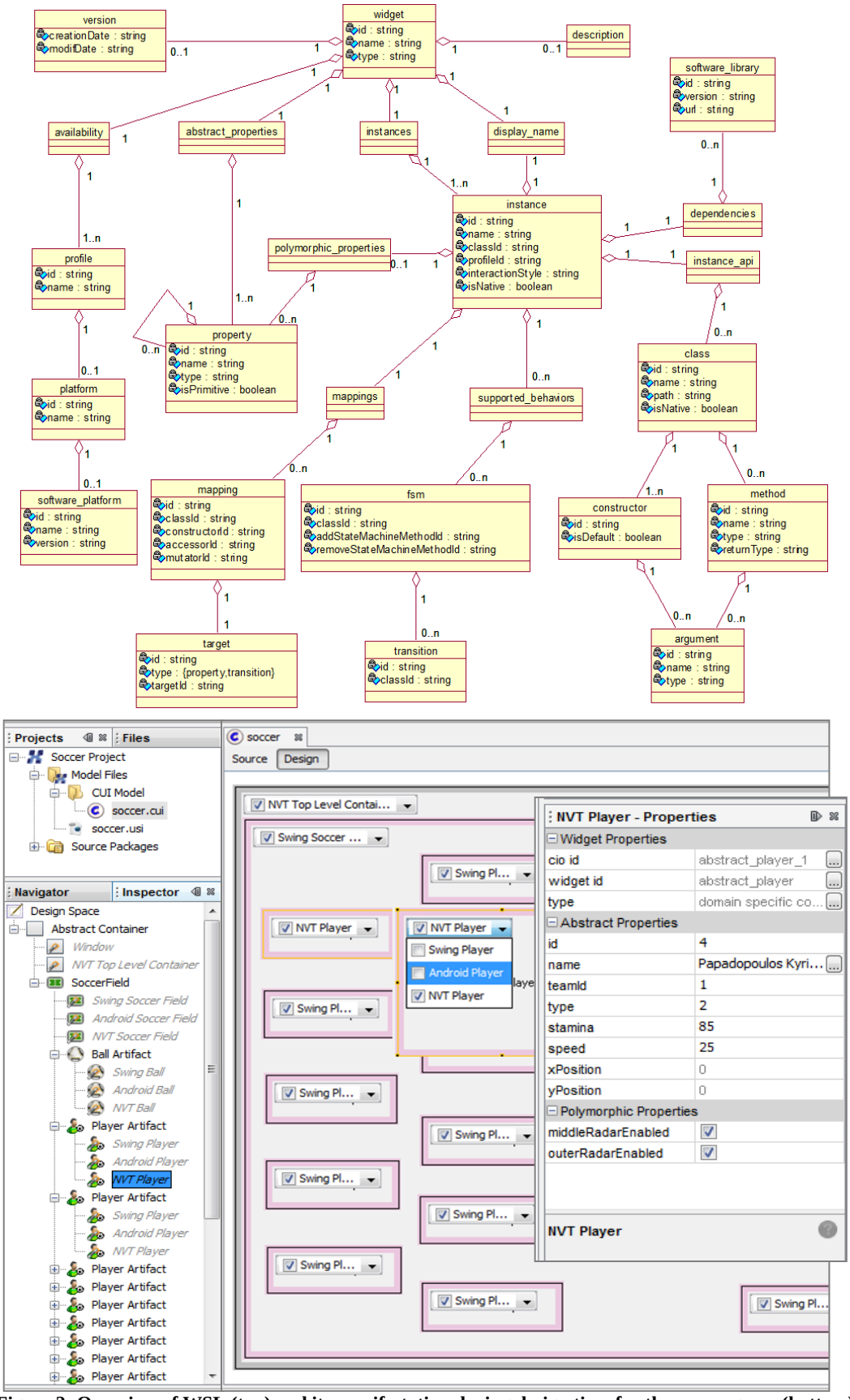

Figure 2: Overview of WSL (top) and its manifestation during design time for the soccer game (bottom)

Consider for instance, the instantiations of 'abstractPlayer' presented in Figure 1 where the common states to be synchronized are 'focused' and 'unfocused'. Each time a state transition is triggered a 'state changed' event is propagated to co-active instantiations. While this seems to be working, there are many cases were it does not suffice. For example, it is not sufficient when an 'abstractPlayer' is to be synchronized with an 'abstractTextEntry' so that the latter provides a narrative view i.e., sub-titles or 'LiveHistory' of the game. Apparently, an 'abstractPlayer' and an 'abstractTextEntry' have no common states to synchronize. However, even if we strictly consider synchronization across instances of the same type of 'abstract widget' (where common states are most likely to exist), there would still be failures. Specifically, the alternative instantiations of the 'abstractPlayer' presented in Figure 1 use different semantics to dictate the same state. For instance, the Android player may use a range of values to indicate transitions from 'focused' to 'not-focused' and this may be different from the value range used by the Swing player to coin transitions from 'selected' to 'nonSelected' states. Therefore, models cannot be directly synchronized, unless dedicated translators are devised; something which is inappropriate and computationally demanding. The above provide the rationale for an abstraction pattern defining a shared model (i.e., common semantics) based on which alternative interactive instantiations of the same or different 'abstract widgets' may synchronize. This type of sharing decouples presentation specific details from semantics of collaboration (i.e., shared data, distribution mode and scheme, access policy, etc.).

To illustrate the concept, synchronization of the three players in Figure 1 could be achieved by defining a shared property (i.e., pressed, unpressed, mouse over) to trigger synchronization. This is challenging and novel as it allows indirect coupling of behaviors of potentially different objects classes. An example is when a button is synchronized with a text label so that the text level depicts a specific value upon button state change. The shared abstraction model is interactively manifested at design time comprising suitable classes that define the semantics of collaboration in terms of properties (i.e., 'shared properties'). This model is then replicated across physical settings - a process transparently handled by the underlying runtime environment. In this manner, it becomes simpler to centralize concerns via abstraction classes, than it would be by trying to define intertwined relationships across several widgets.

4.2.3. Consistency Model

Having detailed the role of behavior and abstraction models, we now turn to feedthrough and consistency issues that arise when collaborators do not share the same user interface. Our consistency model implements a feedthrough mechanism that declares the links (or bindings) to be established across 'abstract' properties of abstraction classes and instance-specific properties. Through such bindings, it is possible to broadcast potential changes in the value of an abstract property to all widgets linked to that property. This process is transparent to developers as it is automatically handled by the runtime environment in a manner that guarantees that states of peer widgets comply with a 'consistent state' designated in the corresponding block tag of the abstraction model.

5. IMPLEMENTATION

5.1. Client- and server-side components

The client side comprises a runtime infrastructure referred to as 'Platform Server' (PS) [15]. A PS is a multifunctional software component which guarantees smooth and consistent handling of all mappings of abstract to local (i.e., device-specific) and vice versa. To achieve such mappings the PS constitutes a virtual software layer between UsiXML models and the libraries of a specific platform. Its role amounts to undertaking distributed class loading (in case of managing non-native interactive elements), event management (as part of facilitating collocated and/or distributed synchronization), as well as runtime compilation and interpretation of UsiXML models. Further technical details of the PS implementation require insight which is beyond the scope of the present work. The server-side implements generic components for session management, notification and web services. It also maintains a list with all running sessions and a repository of UsiXML models associated to a particular session (either synchronous or asynchronous). Finally, it makes use of a shared repository with platform-specific widget libraries (native and non-native) to facilitate distributed class loading.

5.2. Run-time scenario

The actual UIs generated for our scenario are depicted in Figure 3. Each UI version is assembled to link to the appropriate platform server to account for available interaction components, input models, event handling, etc., and the users' parameters (i.e., roles). For non-sighted users a customized Non-visual Toolkit was built and integrated using our WSL. Part of the non-visual toolkit's functionality is provided by a 3D audio plugin bundled within its distribution. As a result, users can experience the 'illusion' of hearing sound effects like these originating in a specified distance (i.e., very close, far, etc.) and direction (i.e., back-left, front right, etc.) in virtual space. This mechanism has been extensively used so as to support a range of features, such as group and location awareness. The non-sighted user's player is instantiated with three concentric circles formed around the player having the focus (with player id 22). The inner radar informs users about events of immediate intention, triggering a continuous 3D sound signal for as long it collides with any point of interest. The middle and outer disc-shaped radars notify the player about events of medium and loose importance, respectively. To this effect, each generates a specific sound signal only for as long as the moving radar line intersects with the point of interest. As game co-play progresses, there are various exchanges between the components of the run-time environment. Each device-specific operation triggers its local effects and propagates these effects to the corresponding platform server for further processing. Then, the platform server informs the collaboration plug-in to update the shared model. The collaboration plug-in notifies all registered platform servers, which in turn, initiate the appropriate actions. For instance, when the non-sighted user's player issues a move down-right voice command, two actions are triggered – one is to track the change in player's position and the other is to notify that the player's current state has changed to 'isMoving'. For such 'local' actions to be effected across devices, they need to be propagated to the shared model in the collaboration plug-in. In our example, the shared data model implements one abstraction class for each player, tracking changes in the player's horizontal and vertical coordinates and the player's selection (i.e., true/false for selected/non-selected respectively). As the clients' UIs need not be identical, the consistency model undertakes to define links between abstract properties of the shared data model and the properties of the distributed instances (i.e., players' position and states, highlight for selected users). At runtime, the platform server undertakes the required transformation to map the property to a corresponding API so that the designated change is locally effected.

Figure 3: The run-time settings for non-sighted/desktop (left), sighted/mobile (middle) and audience/ desktop (right)

6. DISCUSSION AND CONCLUSION

This research extends MBUI engineering to support non-trivial interaction scenarios where virtual work is framed as operations *with*, *on* and *through* and *within* digital representations. Specifically, remote users are enabled to co-lay when replaced by their digital identities, accounts or profiles. This is a typical case of certain physical referents (i.e., our users) operating *with* their digital 'voices' or counterparts. During game co-play users operate *on* digital representations that stand for objects (i.e., players) situated in a certain context (i.e., a soccer field) and certain state (i.e., possessing the ball). Manipulation of the properties of the objects in the soccer field entails a kind of remote control where visual and auditory cues lead users to modify aspects of the situation (e.g., relocating players on the field by updates of the coordinates). In effect, these actions entail operations *through* representations to change the state of affairs in an emergent reality. Noticeably, these effects are mapped to or invoked by potentially different UI instances and interaction techniques. As game co-play involves turn-raking, shooting the ball, moving around in response to the enemy's position on the field, etc., the resulting experience is a simulation where users operate *within* digital representations to create, if only temporarily, something that lacks a single physical referent. In such cases, UIs do not only mediate the users' relationship with objects or people, but they also emulate physical processes and, for this reason, they create new virtualities. This is achieved by a unified specification of fully synchronous UIs exhibiting variations at physical (i.e., platforms and I/O devices), syntactic (i.e., interaction objects and custom widgets) and semantic (i.e., metaphoric representations) levels.

7. ACKNOWLEDGMENTS

This research has been co-financed by the ESF and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: ARCHIMEDES III "Investing in knowledge society through the European Social Fund".

8. REFERENCES

[1] Frey, A-G., Céret, E., Dupuy-Chessa, S., Calvary, G., and Gabillon, Y. 2012. UsiComp: an extensible model-driven composer. In *Proc. of EICS '12*. ACM, NY, 263-268.

[2] Savidis, A., Stephanidis, C., and Akoumianakis, D. 1997. Unifying toolkit programming layers: a multi-purpose toolkit integration module, In *Proc. of DSV-IS'97*. 177–192.

[3] Savidis, A., and Stephanidis, C. 1995. Developing dual user interfaces for integrating blind and sighted users: the HOMER UIMS. In *Proc. of CHI'95*. ACM, NY, 106-113.

[4] Blattner, M., Glinert, E., Jorge, J. and Ormsby, G. 1992, Metawidgets: Towards a Theory of Multimodal Interface Design. In *Proc. of the 16th COMPSAC '92*, 115-120.

[5] Crease, M., Gray, P., and Brewster, S. 2000. A toolkit of mechanism and context independent widgets. In *Proc. of DSV-IS'00*, Springer, 121-133.

[6] Crease, M., Brewster, S., and Gray, P. 2000. Caring, Sharing Widgets: A Toolkit of Sensitive Widgets. In *Proc. of BCS HCI'00*, Springer, 257-270.

[7] Hill, J., and Gutwin, C. 2004. The MAUI Toolkit: Groupware Widgets for Group Awareness. *CSCW* 13 (5-6), 539-571.

[8] Bartel, J., Dewan, P. 2012. Towards multi-domain collaborative toolkits. In *CSCW '12*. ACM, NY, 1297-1306.

[9] Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. 2003. Unifying Reference Framework for multi-target UIs. *IwC* 15(3), 289-308.

[10] Guerrero, J., Lemaigre, Ch., Gonzalez Calleros, J.M., and Vanderdonckt, J. 2008. Towards a Model-Based User Interface Development for Workflow Information Systems, *Journal of Universal Computer Science*, 14(19), 3236-3249.

[11] Molina, A.I., Redondo, M.A., Ortega, M., Hoppe, H.U. 2007. CIAM: A methodology for the development of groupware user interfaces. *J. of Universal Computer Science*, 14(9).

[12] Wolfe, C., Graham, N., Phillips, G., and Roy, B. 2009. Fiia: user-centered development of adaptive groupware systems. In *Proc. of EICS '09*. ACM, NY, 275-284.

[13] Stegemann, T., Ziegler, J., Hussein, T., and Gaulke, W. 2012. Interactive construction of semantic widgets for visualizing semantic web data. In *EICS '12*. ACM, NY, 157-162.

[14] Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. 2004. USIXML: a language supporting multi-path development of user interfaces. In *Proc. of EHCI-DSVIS'04*, Springer, 200-220.

[15] Vellis, G., Kotsalis, D., Akoumianakis, D., and Vanderdonckt, J. 2012. Model-Based Engineering of Multi-platform, Synchronous and Collaborative UIs - Extending UsiXML for Polymorphic User Interface Specification. In *Proc. of PCI '12*. IEEE, Washington, DC, 339-344.

[16] Penichet, V., Lozano, M., Gallud, J., Tesoriero, R. 2009. User interface analysis for groupware applications in the TOUCHE process model. *Adv. in Software Eng.* 40(12), 1212-1222.

[17] Appert, C., and Beaudouin-Lafon, M. 2008. SwingStates: adding state machines to Java and the Swing toolkit. *Software Practice & Experience* 38(11), 1149-1182.