# Model-based engineering of multi-platform, synchronous & collaborative UIs

## Extending UsiXML for polymorphic user interface specification

George Vellis[1], Dimitrios Kotsalis[1], Demosthenes Akoumianakis[1], Jean Vanderdonckt[2]
[1]Department of Applied Informatics & Multimedia, Technological Education Institution of Crete, Greece
[2]Université catholique de Louvain, Louvain School of Management, Belgium
{g.vellis, kotsalis, da}@epp.teicrete.gr, Jean.vanderdonckt@uclouvan.be

***Abstract*— The paper describes an engineering method for building user interfaces for ubiquitous environments. The method comprises of several extensions in the UsiXML family of modes as well as design and runtime support so as to enable multi-platform, synchronous and collaborative interactions. We demonstrate key concepts of the method and their application by elaborating a scenario of collaborative co-play of the 'tic-tac-toe' game. The specific use case features synchronous co-engagement in game play by remote users (players or observers) using desktop PCs or Android devices.**

***Keywords- Polymorphic UI instantiation; collaborative user interfaces; UsiXML; Model-based UI engineering***

## I. INTRODUCTION

In recent years, several interactive applications have emerged enabling users to engage in various sorts of collaborative and social endeavors using a variety of novel computational appliances such as smart phones and tablets. In such settings one important aspect to consider relates to the design and computational manifestation of user interfaces (UIs) capable to support increasingly complex and collaborative work. Early desktop-based UI engineering addressed this challenge through concepts such as groupware toolkits and multi-user UIs [1]. In all cases, the core approaches relies on toolkit programming methods, which however make assumptions about object classes, dialogue, and runtime environments that prohibit flexible UI instantiations for different purposes or across platforms and/or contexts of use. An alternative to toolkit-based programming is model-based development of UIs (MBUI). This is an approach featuring the use of models to specify different aspects of a UI. Arguably, MBUI engineering provides a better frame of reference for interactive applications intended for ubiquitous use. Advantages result from the commitment to abstract notations and mark-up languages to facilitate specification of abstract components and their subsequent mapping to platform-specific vocabularies. Such mappings entail transformation schemes that result in delegating the display to a platform-specific renderer [2].

The proliferation of a wide range of platforms and devices challenges MBUI engineering at several fronts. Firstly, due to variations in the target interaction vocabularies, mapping schemes are frequently ad hoc, limited to simple components and certain types of UIs (i.e., form-based). Secondly, the increasingly collaborative settings complicate MBUI engineering as they bring to the surface novel requirements such synchronization, awareness and true plasticity. Attempts to address these challenges (i.e., [3], [15], [16], [20]) are still immature or at best limited to a narrow range of collaborative engagements. For instance, there have been efforts concentrating primarily on devising notations and tools to model cooperative dialogue and workflows [20], but they seem to dismiss aspects of synchronous and cross-platform activities. It is therefore compelling to devise development methods, properly supported by dedicated tools, so as to accommodate intrinsic requirements of increasingly ubiquitous contexts of use.

This paper proposes a method and a set of tools that extent MBUI engineering by articulating the concept of polymorphic UI instantiation schemes. Polymorphic instantiation relies on implementation agnostic (i.e., abstract) specifications of UIs which at run-time – and once user and usage context parameters are discovered – may be translated to context-specific interaction vocabularies using dedicated tools. An imperative commitment of this claim is that UIs can be assembled at runtime, rather than programmed, so as to comprise of those interactive incarnations of 'abstract' widgets that best fit the current context of use. The present work elaborates the concept and describes how it is integrated into a popular MBUI engineering method, namely UsiXML, so as to facilitate complex requirements such as users' co-engagement in synchronous collaborative sessions and management of diverse collections of objects (both native and non-native), as well as novel affordances such as awareness and social translucence.

## II. RELATED WORK

The concept of polymorphic UI instantiation was initially proposed and implemented in the Platform Integration Module [5] and subsequently in the HOMER UIMS [4]. In both cases, it was conceived of as a language construct inscribed in toolkit-based implementations, while it was applied to accommodate specific accessibility challenges (i.e., access to visual and non-visual UIs). More recent research streams revisit aspects of polymorphic instantiation but from a totally different perspective.

### *A. Early efforts in toolkit-based approaches*

At core, polymorphic instantiation entails a capability for adaptations and UI assembly rather than programming. Toolkit-based approaches do not support UI assembly. On the other hand, there have been proposals aiming to facilitate adaptive interactions. An early effort is Meta-

978-0-7695-4825-8/12 $26.00 © 2012 IEEE
DOI 10.1109/PCi.2012.27

widgets [7] focusing on architectural styles for encapsulating alternative object classes into widget abstractions. Meta-widgets were conceived of as components on top of implementation-specific toolkits and were applied for building adaptive multi-modal UIs. However, their implementation assumes non-extensible instantiation schemes, while provisions for multi-user aspects and novel affordances (i.e., information sharing, awareness, etc.) are completely dismissed. Subsequent efforts, such as [8] its ancestor [9] and [10], share similar grounds though adopting a slightly different perspective. The key difference from Meta-widgets is that these toolkits separate alternative input and output mechanisms from the actual (i.e., common) behavior supported by a widget. Nevertheless, as in the case of Meta-widgets, they fall short in support for synchronous collaborative aspects.

Another key intention of polymorphic instantiation is to support collaboration. Again, early efforts address this goal sub-optimally. Groupware toolkits [21] followed the path of providing high level-abstractions of low-level programmatic-intensive tasks to facilitate session management, communication, sharing, awareness and synchronization. MAUI [1] is an indicative example of this category exploiting 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, all efforts in this vein fall short in supporting heterogeneous contexts of use since they make assumptions about the underlying platform or toolkit. As a result they are biased either to a single-only or a set of homogenous platforms.

### B. *Promises of MBUI engineering*

With the advent of MBUI engineering methods, the constraints of toolkit-based approaches could be relaxed. Several new frameworks have been proposed claiming advantages over the previous development paradigm. In [12], a method is proposed allowing the specification of a UI at multiple levels of abstraction by means of specific model types. Adaptations are supported through appropriate transformations on source models so as to obtain the desired target model. Transformations can be applied at the same level (constituting translation process), in an abstract to concrete order (reification process) or the reverse i.e., from concrete to a more abstract definition (abstraction). A UIs capacity to properly adapt in its current context of use is determined by its ability to devise each time the appropriate transformation rules.

The COMMETS Framework [6] represents a further step in this direction. The mapping problem is addressed using semantic networks [18], thus enabling intuitive runtime adaptations while ensuring continuity of use (i.e., plasticity) at any level of abstraction (tasks, AUI, CUI). 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 which is demanding in terms of comprehension and successful management.

Some MBUI engineering methods have attempted to explore collaborative interaction. Indicative examples include FlowiXML [20], AMENITIES [15], CIAM [16] and TOUCHE [3]. These efforts concentrate primarily on devising notations and tools to model cooperative behavior and workflows. In effect, their primary contribution is that they make explicit different elements of collaboration (i.e., roles, responsibilities and tasks) using dedicated notations. However, only some of these efforts make inroads towards generating the UI of collaborative applications. An example in this direction is TOUCHE [3] providing multiuser functionality using ad-hoc mappings to a custom underlying groupware toolkit. As a result multi-platform support is limited by the availability and support of the underlying toolkit in every target context.

In spite of these shortcomings, MBUI engineering remains a promising strand as it offers methods that are extensible and can be augmented to cope with the issues pending. An example is UsiXML [12] which constitutes the reference implementation of the Cameleon Reference framework devised to provide support for plasticity [14]. UsiXML proposes the definition of UIs at four levels of abstraction each focusing on different aspects of the development process. Specifically, at the tasks and concepts layer (i.e., the most abstract level supported) a UI is specified in terms of incremental decomposition of tasks into sub-tasks anchored by operators defining sequence of execution. At the AUI level [11] (i.e., the next most concrete layer) a UI is enabled to be specified independently of any interaction modality as an interactive hierarchy comprised of abstract interaction objects and containers. Finally, at the CUI level of abstraction a UI is defined in terms of concrete interaction objects properly assembled independent of any implementation-specific technology (i.e., Platform Independent Model). Transitions between different levels of abstractions are supported by adopting a semiautomatic transformational approach (i.e., graph transformations). One serious limitation of UsiXML is the support it offers for utilizing different target presentation vocabularies. Specifically, at present the language is limited to interaction elements supported by most popular toolkits. This is obviously a shortcoming as it constrains the language's expressiveness and restricts the type of UIs possible to form-based.

## III. APPROACH

The present work seeks to alleviate several of the limitations elaborated earlier, thus establishing new grounds for MBUI engineering. At base, our effort is anchored by (a) extending UsiXML ([12], [13]) so as to provide full support for polymorphic UI instantiation and (b) providing sufficient design and run-time support. The rest of the paper presents recent extensions following the proposal detailed in [17].

### A. *Language Extensions*

UsiXML extensions cover new workflows and language-level constructs as well as enhancements in the UsiXML family of models.

*Polymorphic Widget Specification Work flow.* In order to provide support for the diverse collections of objects, either native or custom, a Widget Specification work flow has been introduced to allow XML schema compliant widget specs. The workflow relies on a dedicated specification language (WSL) that allows our tools to integrate and utilize third party widget libraries (Figure 1). In general, for any 'abstract' widget to be deployed in our platform it must first be associated with a corresponding instance of the WSL. Each WSL instance comprises of a unique id and name, platform availability, a list of all abstract properties (i.e., common across all alternative instantiations), as well as the alternative polymorphic instantiations to be exploited at runtime. Additionally, each instantiation has to expose its API (i.e., constructors, accessor and mutator methods, etc.) and define a list of all polymorphic properties it supports.

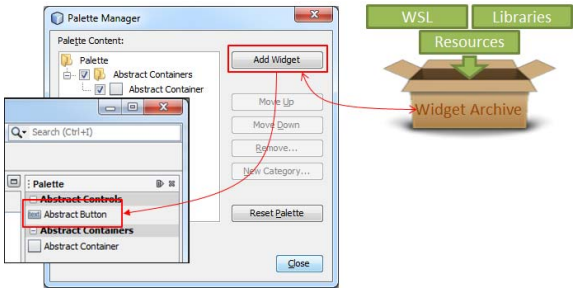


Figure 1: Overview of the widget specification workflow

An example of a WSL abstract widget is depicted in Figure 2, summarizing an abstract button and its designated instances in terms of properties. As shown the specification comprises of the widget's unique id and name, followed by an enumeration of all alternative interactive instantiations supported. In this manner, it is possible to define polymorphic instances of an abstract button as in Figure 3.

```
<widget id="abstract_button" name="abstractButton" type="control">
    ...
    <instances>
        <instance id="inst_1" name="rectangleButton" isSocialAware="false"... >
            ...
        </instance>
        <instance id="inst_2" name="roundButton" isSocialAware="true"... >
            ...
        </instance>
    </instances>
</widget>
<widgetResourceModel>
    <widgetResource id="wdg_rsc_4" cioId="abstract_button_1" widgetId="abstract_button">
        ...
        <instance id="inst_1" ... >...</instance>
    </widgetResource>
    <widgetResource id="wdg_rsc_5" cioId="abstract_button_2" widgetId="abstract_button">
        ...
        <instance id="inst_1" ... >...</instance>
        <instance id="inst_2" socialAwarenessEnabled="false"... >...</instance>
    </widgetResource>
    <widgetResource id="wdg_rsc_6" cioId="abstract_button_3" widgetId="abstract_button">
        ...
        <instance id="inst_2" socialAwarenessEnabled="true"... >...</instance>
    </widgetResource>
    ...
</widgetResourceModel>
```



Figure 2: 'abstractButton' WSL


Figure 3: Polymorphic instantiations

It is worth noticing that the left hand side (rectangular) represents a conventional single-user instance, while the right hand side is non-native, intended for distributed synchronous collaborative rendering with social scent. Figure 2 details that social scent is a property of the abstract widget, allowed only in the second (round) instance.

*Behavior Model*. The two version of the abstract button indicate that widget instances need not differ only at physical level but also in terms of dialogue. This necessitates provisions for accommodating different interactive behaviors. Thus, a behavior model is used to capture application-specific states and the state transition logic registered to designated polymorphic instances. This is done using Finite State Machines (FSM) for interfacing and monitoring without the need of using low-level event-listener classes [19]. Moreover, FSMs make it possible for designers to define custom states and thereby associate alternative behaviors to different object instances as needed. Another property of FSMs is that they allow for a high level synchronization of widgets supporting alternative input mechanisms and transitions. For example, it is possible to synchronize two buttons by manipulating designated states. Thus, a conventional desktop button sensitive to 'mouseOver' input behavior can be synchronized with a touch sensitive android with a corresponding 'onTap' input mechanism. Such synchronization can be easily implemented using FSMs on the grounds of the common states supported (i.e., pressed, released) completely ignoring the local widget transitions. This allows not only for relaxed-coupling between distributed users, but also for more advanced behavior modeling. Finally, FSMs can be defined in an implementation-independent way. 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 spec.

*Abstraction Model*. In addition to coping with alternative behaviors, it is also important to devise models to unify alternative interactive instances across different settings (i.e., distributed or collocated) on the grounds of shared models (in the sense of MVC). Thus, an abstraction model comprises of classes defining those common properties (relieved from physical characteristics). Consequently, an abstraction model facilitates model-level sharing (in contrast to toolkit-level sharing) in distributed settings. The reason for not directly synchronizing widget models (again in the MVC sense) is because it is quite simpler to centralize concerns via abstraction classes, than it would be by trying to define intertwined relationships across several widgets. Moreover, it would be rather impossible to inject collaboration aware code to collaboration unaware widgets.

*Consistency Model.* Having detailed the role of behavior and abstraction models, we now turn to consistency issues. A consistency model implements the role of a broker between widgets to be synchronized through abstraction

classes. Such a role comprises declaration of 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.

### B. *Design tools and suite*

Having described briefly key language extensions and underlying models, the focus is now on explaining how these concepts are implicated during the design and run-time phases. Starting with the design phase, a prototype system has been developed on top of the NetBeans platform to allow development of either single user or distributed collaborative projects. The main differences between these two project types are in the way they compile, distribute and execute the produced UI specifications, as well as in the number of available plugins engaged by default. For instance in case of collaborative application, a pre-requisite is the registration of a compatible Server Side Environment dedicated to managing special purpose collaborative aspects. Moreover, in distributed collaborative applications where UI models need to be accessible by several users over the network the pre-requisite is a centralized repository for depositing shared resources (i.e., common models, widget archives, etc.). Furthermore, additional provisions are required for distributing a reference to all users that may be engaged in a particular session (i.e. 'distributed shortcuts'). Nevertheless, the design process is unified and common in terms of steps and custom plugins (i.e., editors for manipulating CTT, CUI, other models, etc.). We will discuss the design tool when elaborating the use case.

### C. *Run-time environment*

In order to support the novel features introduced in the previous sections, advanced software components have been crafted both at the client and server sides (Figure 4).

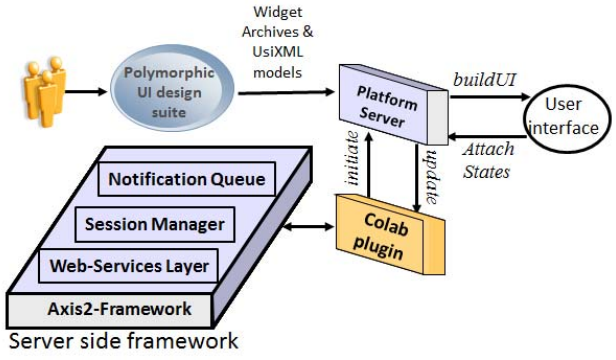


Figure 4: Run time environment

*Client-side components*. At the client side of particular interest is a runtime infrastructure developed, namely the 'Platform Server' (PS) [17]. The platform server is multifunctional software component which guarantees smooth and consistent boundary spanning capability by handling 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 intrinsic 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. Specifically, the PS undertakes the handling of replication by managing and maintaining a client-side replication list containing the replicaIds associated to corresponding object-references. In case of detected variations (via 'context-sniffer' daemon thread) regarding the context of use, PS is responsible for engaging a re-adaptation process instructed by the Server-Side Framework (SSF) framework. Furthermore, another important function assigned to the PS relates to the process of handling WSL compliant non-native widgets. In part this amounts to 'custom events management' and 'widget data model' handling briefly discussed in previous section. A separate feature of the PS relates to collaborative session management, in case of synchronous or asynchronous co-engagements. Specifically, each PS handles both grabbing and distribution of shared actions via triggering and inter-client (and thus inter-PS) message exchanges in the course of a session. To support this functionality PS interoperates the SSF framework supporting session management.

*Server-side components.* The SSF implements several generic components such as session management, notification and web services. Additionally it maintains a repository of runtime UsiXML models associated to a particular session (either synchronous or asynchronous). The SSF also handles low-level session management built on top of apache axis2 framework, by performing several functions such as creation, registration, etc. It also maintains a list with all running sessions. As for UIs utilizing non-native widgets, the SSF maintains a shared repository with platform-specific widget libraries and facilitates distributed class loading.

## IV. USE CASE SCENARIO

This section briefly elaborates on a relatively simple but demanding use case entailing synchronous collaborative co-play of the 'tic tac toe' game. Game co-play is conceived of as multiuser co-engagement (using different terminals) in a synchronous session. For purposes of illustration two roles are assumed - one for players and one for observers. Observers are assigned read-only access to application's shared state. Players can update the shared state/model as they co-engage synchronously in play. In terms of platforms, the game is conducted using a conventional desktop (with JSE) and Android devices. Figure 5 depicts an instance of the polymorphic UI designer suite.

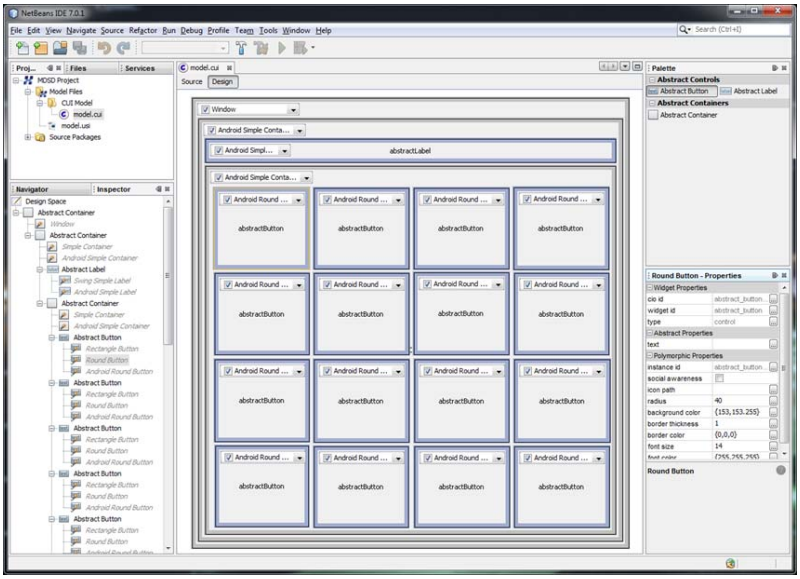

Figure 5: Instance of the polymorphic UI designer

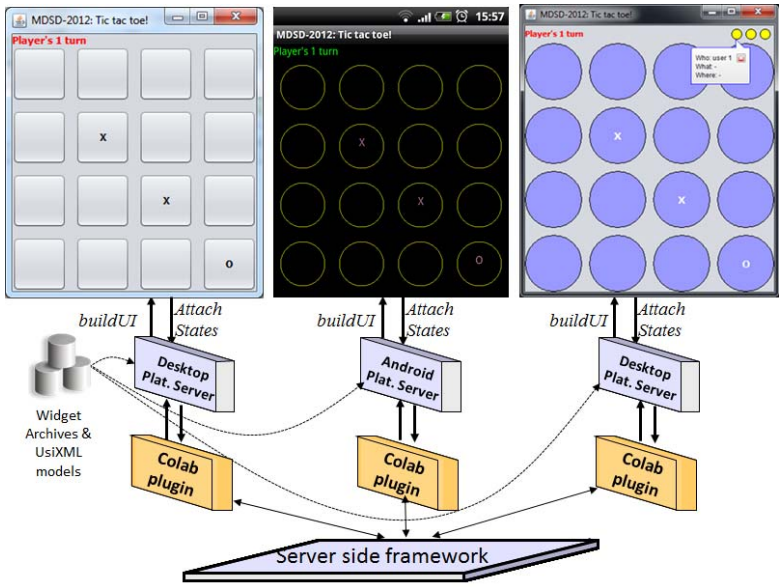


Figure 6: The run-time deployment of the use case

This is an IDE that allows (a) integration of (native or custom) widgets compiled in accordance to WSL (b) drag & drop operations to craft abstract UIs (c) specification of semantic and physical properties of interaction (d) automatic inspection of the resulting XML code by switching from design to source code overview. A noticeable feature of the IDE in design mode (as in Figure 5 depicting the Android instance) is the palette with the abstract controls. This palette assembles and presents all interaction elements and resources (native and custom objects of Figure 3) that have been integrated (through the process depicted in Figure 1). During the design phase, designers manipulate abstract controls and set their properties accordingly for each role- and platform-specific UI instance. Figure 5 illustrates the abstract class hierarchy for the player using Android. As shown, an abstract window is designated as the host of an abstract Android container (with a label) and 16 abstract buttons each set to the custom type 'roundButton'. Abstract and polymorphic properties of the 'roundButton' widget can be set using the design palette (lower part dialog).

Figure 6 illustrates the run-time details of our use case scenario. The left-hand and middle UIs correspond to the two distributed players, while the third UI is the observer's instance. All of them are derived from the polymorphic UI specification process briefly described earlier. As shown, each is dynamically assembled to link to the appropriate platform server so as to cope with specific constraints (i.e., interaction components, input models, event handling, etc.) or end user-related parameters (i.e., roles). It is worth noticing that the UIs exhibit inconsistencies both at lexical level (i.e., type of buttons) and syntactic level (i.e., provisions for social scent in the observer's UI).

Nevertheless, they remain synchronized at all times. As game co-play progresses, there are various exchanges between the components of the run-time environment. Specifically, each operation initiated in a specific device context triggers its local effects and is also propagated to the corresponding PS for further processing. The PS informs the collaboration plug-in so as to update the shared model. In doing so, the collaboration plug-in notifies all registered PSs, which in turn initiate the appropriate actions. For instance, when a player presses a designated button located in a certain position of the grid and marked with a label, two actions are implicated - one is to track the change in the button's label and the other is to notify the change in the button's state. For such 'local' actions to be effected across devices, they need to be propagated to the shared model in the collaboration plug-in. For our example, the shared data model needs to implement two abstraction classes – one for tracking changes in the value of buttons' label (i.e., sixteen in total in the grid) and one for tracking changes in the state of each button (i.e., true/false for pressed/released respectively). For propagating updates of the shared model across target vocabularies, a separate FSM is attached to each polymorphic widget in the players' sides. This is not needed for observers as no updates to shared data are allowed. As the players' 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 polymorphic instances affected (i.e., button labels and states, radius for round buttons). This is achieved by linking an abstract property with the corresponding property of a particular instance of the UI as defined in the widget resource model. Accordingly, at runtime, the PS undertakes the required translation to map the property to a corresponding API call so that the designated change is effected locally.

## V. Concluding Remarks

At present, we have a fully implemented version of the components (models, plug-ins and run-time environment) introduced earlier as well as working prototypes of several UIs. Ongoing work concentrates on several fronts. One is coping with more complex widgets, either domain-specific or available in advanced toolkits (i.e., visualization). Another is extending the framework to further enhance its capabilities with regards to certain affordances such as social translucence, run-time adaptivity and UI plasticity in distributed and ubiquitous settings.

## Acknowledgments

The present work is part of the first two authors' doctoral research at Catholic University of Leuven (Belgium), conducted at iSTLab (www.istl.teicrete.gr).

## References

[1] J. Hill and C. Gutwin, "The MAUI toolkit: Groupware widgets for group awareness", Computer Supported Cooperative Work, vol. 13, Dec. 2004, pp. 539-571, doi:10.1007/s10606-004-5063-7.

[2] L. Choonhwa, S. Helal, and L. Wonjun, "Universal interactions with smart spaces", IEEE Pervasive Computing, vol. 5, Jan. 2006, pp.16-21, doi:10.1109/MPRV.2006.19.

[3] V. Penichet, M. Lozano, J. Gallud, and R. Tesoriero, "User interface analysis for groupware applications in the TOUCHE process model", Adv. Eng. Softw, vol.40, Dec. 2009, pp. 955-964, doi:10.1016/j.advengsoft.2009.01.026.

[4] A. Savidis and C. Stephanidis, "Developing dual user interfaces for integrating blind and sighted users: the HOMER UIMS", Proc. ACM Conf. Human Factors in Computing Systems (CHI 95), ACM Press, May 1995, pp. 106-113, doi:10.1145/223904.223918.

[5] A. Savidis, C. Stephanidis, and D. Akoumianakis, "Unifying toolkit programming layers: a multi-purpose toolkit integration module", Proc. Eurographics Workshop on Design, Specification & Verification of Interactive Systems (DSV-IS 97), 1997, pp. 177-192.

[6] A. Demeure, G. Calvary, J. Coutaz, and J. Vanderdonckt, "The COMETs inspector: towards run time plasticity control based on a semantic network", Proc. Conf. Task models and diagrams for users interface design (TAMODIA 06), Springer-Verlag, pp. 324-338.

[7] M. Blattner, E. Glinert, J. Jorge, and G. Ormsby, "Metawidgets: towards a theory of multimodal interface design", Proc. Int. Conf. Comp. Soft. and Applic. (COMPSAC 92), Sep. 1992, pp. 115-120.

[8] M. Crease, P. Gray, and S.A. Brewster, "A toolkit of mechanism and context independent widgets", Proc. Int. Workshop on Design, Specification, and Verification of Interactive Systems (DSVIS 00), Springer-Verlag, 2000, pp. 121-133.

[9] B. Jabarin and T.C.N. Graham, "Architectures for widget-level plasticity", Proc. Eurographics Workshop on Design, Specification & Verification of Interactive Systems (DSV-IS 03), 2003, pp. 124-138.

[10] M. Crease, S. Brewster, and P. Gray, "Caring, sharing widgets: a toolkit of sensitive widgets", Proc. HCI'2000, Springer, pp. 257-270.

[11] A. Stanciulescu, "A Methodology for Developing Multimodal User Interfaces of Information Systems", Ph.D. thesis, Université catholique de Louvain, Louvain-la-Neuve, 2008.

[12] Q. Limbourg, "Multi-Path Development of User Interfaces", Ph.D thesis, Université catholique de Louvain, Louvain-la-Neuve, 2004.

[13] Q. Limbourg and J. Vanderdonckt, "USIXML: A User Interface Description Language Supporting Multiple Levels of Independence", Proc. ICWE Workshops, 2004, pp.325-338.

[14] G. Calvary, et al. "A Unifying Reference Framework for multi-target user interfaces". Interacting with Computers, vol. 15, June 2003, pp. 289-308, doi:10.1016/S0953-5438(03)00010-9.

[15] J.L. Garrido, M. Gea and M.L. Rodríguez, "Requirements enginnering in cooperative systems", in Requirements Enginnering for Sociotechnical Systems, Idea Group, USA, 2005, pp. 226–244.

[16] A.I. Molina, M.A. Redondo, M. Ortega and U. Hoppe, "CIAM: A methodology for the development of groupware user interfaces", J. UCS, vol.14, 2008, pp.1435-1446, doi:10.3217/jucs-014-09-1435

[17] G. Vellis, D. Kotsalis, D. Akoumianakis and J. Vanderdonckt, "Towards a new generation of MBUI engineering methods: Supporting polymorphic instantiation in synchronous collaborative and ubiquitous environments", in Coyette, A., Faure, D., Gonzalez, J., Vanderdonckt, J. (Eds.), Proc. Int. Workshop on User Interface Description Languages (UIDL 11), 2011 (ISBN 978-2-9536757-1-9).

[18] Q. Limbourg and J. Vanderdonckt, "Addressing the mapping problem in user interface design with UsiXML", Proc. Int. Workshop on Task Models and Diagrams for User Interface Design (TAMODIA '04), ACM Press, Nov. 2004, pp. 155-163. doi:10.1145/1045446.1045474

[19] C. Appert and M. Beaudouin-Lafon, "SwingStates: Adding state machines to Java and the Swing toolkit", Softw. Pract. Exper., vol. 38, Sep. 2008, pp. 1149-1182, doi:10.1002/spe.v38:11.

[20] J. Guerrero, C. Lemaigre, J.M. Gonzalez Calleros and J. Vanderdonckt, "Model-Driven Approach to Design UIs for Workflow Information Systems", J. UCS, vol. 14, 2008, pp. 3160-3173.

[21] B. de Alwis, C. Gutwin and S. Greenberg, "GT/SD: Performance and Simplicity in a Groupware Toolkit", Proc. ACM Symp. Engineering Interactive Computing Systems (EICS 09), ACM Press, Jul. 2009, pp. 265-274, doi:10.1145/1570433.1570483.