Prototyping Distributed Physical User Interfaces in Ambient Intelligence Setups

Gervasio Varela, Alejandro Paz-Lopez, Jose Antonio Becerra Permuy, and Richard J. Duro Fernandez

Integrated Group for Engineering Research, University of A Coruña, C/ Mendizabal S/N, 15403, Ferrol, A Coruña
{gervasio.varela,alpaz,joseantoniobecerrapermuy, richard}@udc.es

Abstract. Ambient Intelligence systems require the development of highly customized distributed UIs adapted to the user and environment characteristics. They make use of many different devices, from different manufacturers, technologies and modalities. Supporting this wide variety of devices and technologies increases the complexity of a system, affecting its costs and development time. The objective of Dandelion, the solution presented in this paper, is to alleviate this complexity and reduce development costs. Dandelion provides a development framework for distributed physical UIs. It is capable of decoupling the system logic from the characteristics and specifics of the interaction devices, and supports the easy prototyping of different physical realizations of a distributed UI.

Keywords: physical user interfaces, distributed user interfaces, ambient intelligence, ubiquitous computing, user interfaces, model-driven engineering.

1 Introduction

The implicit nature of Ambient Intelligence systems, and specially the intrinsic requirements of these systems, such as environment integration, proactivity or natural interaction [1][2][3], makes designing and implementing user interfaces for AmI systems a costly process. A lot of effort must be devoted to the design and testing of highly customized UIs that must be adapted to the user and environment characteristics in order to be perceived as natural.

Those customized UIs are usually conceived as distributed physical user interfaces that make use of many distributed hardware devices [1][4]. These devices come from different manufacturers, use heterogeneous technologies, and in some scenarios, even employ custom hardware that is especially built for the system. Furthermore, if the system is going to be deployed in different scenarios, with different users or environments, many different configurations of the UI may be required.

Designing and implementing the support to this multitude of technologies and devices introduces a lot of complexity in a system, increasing the development costs and the cost of adapting it to new environments. Reducing this complexity and cost is the

N. Streitz and P. Markopoulos (Eds.): DAPI 2014, LNCS 8530, pp. 76–85, 2014.

main objective of the solution presented in this paper, the Dandelion framework. It provides a development framework for distributed physical UIs, where the system logic and interaction logic can be physically and logically decoupled from the devices and technologies used to build the UI. Taking advantage of this decoupling, Dandelion can be easily cast as a prototyping tool that facilitates the testing of different implementations of physical user interfaces.

Dandelion follows a model-driven approach that allows developers to build UIs using abstract interaction components. At deploy time, the abstract UI is connected to a selection of physical distributed devices. This connection is achieved by the Generic Interaction Protocol (GIP) [4], a device abstraction technology that, implemented as a distributed communications protocol, encapsulates the specific behavior of physical devices behind a generic interface of user interaction actions. Model-driven approaches have been widely adopted by the Human-Computer-Interaction (HCI) community since Thevenin and Coutaz [5] proposed the use of models to support UI adaptation to context changes [6][7][8]. Furthermore, the usefulness of model-driven approaches has been recognized within the Ubiquitous Computing and Ambient Intelligence fields, which are starting to use models in order to improve the adaptation capabilities of their system to context changes [8][9].

The Dandelion framework design has been inspired in the CAMELEON-RT [6] and UsiXML [7] projects, and their four UI abstraction levels. Nevertheless, Dandelion relies only on two abstraction levels, the abstract UI level and the final UI level. Besides, Dandelion shares some similarities with the iStuff project [10], as both provide distributed access to smart physical objects and rely on proxy-like components to encapsulate the specific behavior of each device. However there are important differences. First, Dandelion allows the utilization of model-driven techniques to build distributed physical UIs. Second, Dandelion provides an abstraction technology for user interaction devices. Consequently, multiple heterogeneous devices can be accessed using the same homogenous API.

This paper is organized as follows. Section 2 provides a brief overview of the Dandelion framework design and implementation. Section 3 provides an overview of the prototyping capabilities of Dandelion, and Section 4 shows an illustrative example of how to use the Dandelion framework to prototype a distributed physical UI. Finally section 5 discusses some conclusions.

2 Dandelion: A Development Framework for Distributed Physical User Interfaces

This paper presents Dandelion, a development framework for Distributed Physical User Interfaces (DPUIs) in Ambient Intelligence (AmI) environments. The main characteristic of this framework is its ability to increase the decoupling between a digital system and the particularities and specifics of the multiple technologies required by a distributed UI, especially when physical user interfaces are involved. Developers can take advantage of this characteristic to easily prototype and test different physical configurations of the UI. In this section, we are going to show the conceptual architecture and some implementation details of the framework.

The ability to decouple system logic, and more specifically, the system interaction logic, from the particular technologies and hardware devices used to interact with the user is an important characteristic for any system. However, it becomes especially relevant when the interaction system may change depending on the scenario. This is the case of Ambient Intelligence systems, which are expected to provide their functionality in very different scenarios, with their interaction systems adapted to the environment and user characteristics. In these kinds of systems, the devices and technologies used, and even the shape and behavior of the UI may change, not only for different physical environments, but also for different users in the same physical environment.

Dandelion has been conceived as a UI development framework for AmI systems, and as such, it has been designed to facilitate the development of distributed physical UIs capable of changing their shape depending on the use scenario. Figure 1 shows a block deployment diagram of a system using the Dandelion framework. As can be seen, the system logic and the interaction devices operate in a distributed manner, with Dandelion in the middle decoupling them through a UI management system (the Dandelion UI Controller) and a device abstraction layer (the Dandelion FIO layer and GIP protocol). This decoupling is provided at two different levels. First, they are logically decoupled, so that UIs can be built independently of the APIs or technologies used by specific devices. Second, they are physically decoupled, so that the system logic can be run without knowing where the devices implementing the UI are going to be physically deployed.

Dandelion uses a device abstraction layer in order to provide these two levels of decoupling. It is called the Generic Interaction Protocol (GIP), and it encapsulates the specific behavior of each device behind a generic interface of user interaction operations. GIP is a distributed communications protocol that defines a reduced set of interaction actions and creating a generic remote interface to any kind of interaction device. Any device or interaction resource that implements this interface can be remotely accessed using the same set of concepts and operations, thus decoupling the application from the underlying interaction technologies.

On top of this abstraction layer, Dandelion provides a user interface management system that allows developers to take advantage of the GIP in order to isolate their systems from the physical implementation of the UI. Dandelion uses a model-driven approach inspired on the CAMELERON-RT framework and the UsiXML models. The UIs are implemented at an abstract level using a UI Modeling Language, and then, at deploy-time or runtime, they are transformed into a Final UI.

A more detailed description of the Dandelion framework is provided in Figure 2. Dandelion uses the UsiXML Abstract UI model as the modeling language for the definition of UIs. This language allows developers to describe the UI of a system using a reduced set of generic user interaction operations, like input, output or trigger. These interaction operations, called Abstract Interaction Units (AIUs) are an abstract representation of the typical widgets found in graphical user interface toolkits. In Dandelion they represent any kind of interaction element capable of interacting with the user, such as physical devices like appliances or sensors, or even gestures or GUIs.

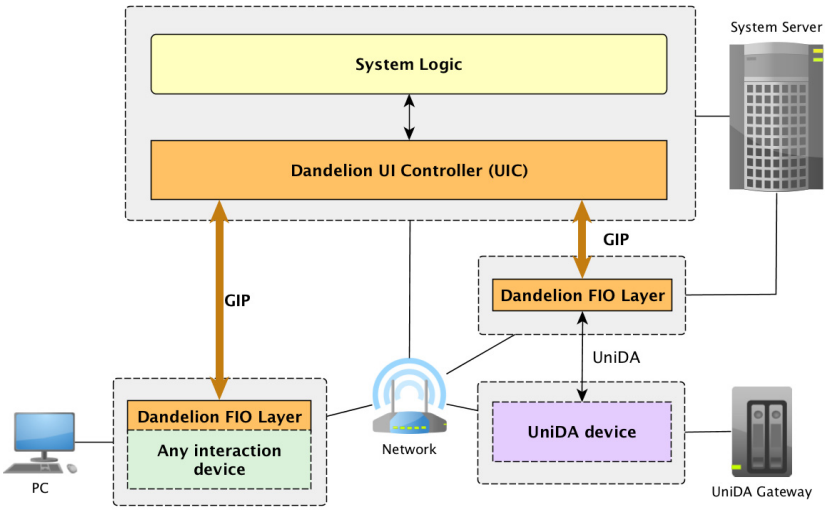

Fig. 1. Block and deployment diagram of a system using the Dandelion framework

Once a developer has implemented the system logic and described the UI using the Abstract UI model, she has to establish a relationship between the system logic and the elements forming the UI. For this purpose, Dandelion treats the AIUs as interaction black boxes capable of performing a specific kind of interaction with the user, for example obtaining data from the user or presenting data. The interface between the system logic and the UI is based on this concept, so the UI Controller uses the observer pattern to monitor some system logic objects specified by the developer. When it detects changes in those objects, the UIC notifies those changes to the AIUs that will translate them into interaction actions performed by the resources implementing the final UI, and vice versa. The data and action objects that must be monitored by the UIC are called the system I/O interface, and developers associate them to AIUs by using an API of the UIC.

In Dandelion, the interaction resources, the real devices used to interact with users, are represented by the Final Interaction Objects (FIOs) concept. They are software representations of the physical devices that encapsulate the specific logic and characteristics of each device behind the General Interaction Protocol interface. Thus, the abstraction of devices is conceptually provided by the GIP and physically realized by the Final Interaction Objects (FIOs).

The GIP is used by the UIC to establish a decoupled distributed connection between the AIUs of the user interface and the FIOs, the end devices that realize the abstract interactions specified by the AIUs. It is an event-based protocol following a publish/subscribe model and it is designed to match the set of interaction operations supported by the UsiXML Abstract UI model, so that it can easily translate between the AIUs of the UI definition and the actions performed by the FIOs. As shown in Figure 2, the GIP defines five different types of events: input, output, selection, action and focus. The UsiXML AUI model directly inspires the first four.

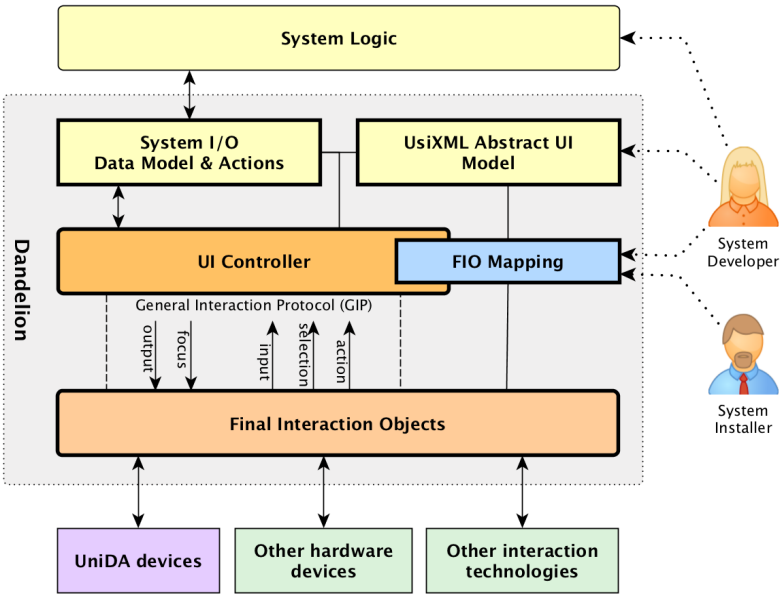

Fig. 2. Architecture diagram of the Dandelion framework

Final Interaction Objects implement the GIP as their interface to external systems. On the one hand, they receive GIP events from the UIC and translate them into specific actions, like showing a text, lighting up a LED, activating a servo, etc. On the other hand, they publish GIP events when a user performs an action in a device. These last events are received by the UIC, which, using a mapping between FIOs and AIUs, transfers them to the AIUs associated to a specific FIO. Finally, the AIUs transform those events into modifications to the data of a system logic object, or the activation of a system logic action.

When using Dandelion to build a distributed physical UI, developers are only required to implement the system logic, describe the UI requirements of the system at an abstract level with the UsiXML Abstract UI model, establish an I/O interface between the system logic and the UI, and finally, at deploy time, select the physical devices that implement the different interaction operations required by the UI. These devices can be physically distributed and use heterogeneous technologies. Dandelion integrates support for the UniDA framework [11][12], an open source framework that provides remote access to heterogeneous devices like home automation systems, so that devices supporting UniDA can be directly used. In the case of custom physical UIs, developers will be required to implement a series of FIOs to adapt their devices to the GIP interface.

Dandelion is implemented as a JAVA library that can be used by any JAVA application to build distributed UIs. It uses STOMP [13] as the supporting technology to implement the GIP protocol. STOMP allows the implementation of FIOs with any programming technology. Furthermore, Dandelion provides a FIO development framework for JAVA to facilitate the implementation of new FIOs in order to support

custom interaction resources (new hardware devices or GUIs), and has integrated support for devices compatible with the UniDA device access framework.

3 Dandelion as a Prototyping Framework for Physical Distributed User Interfaces

Dandelion has been designed and is being implemented as an UI development framework for AmI systems. Therefore, the main idea behind it is to support the changes in the shape of a UI at runtime, so that ubiquitous systems can adapt their UI to changes in the users and changes in the environment. Because of this, it has been designed to integrate, in the future, support for the autonomous selection of the FIOs at runtime as illustrated in [3]. Nevertheless, in its current implementation the selection of FIOs is the responsibility of the developer or the installer of the system.

In this state, Dandelion can be a very useful tool for the prototyping of distributed physical user interfaces. As shown in Figure 3, by taking advantage of the physical and logical decoupling capabilities of Dandelion, developers of the system logic can focus on the system intelligence, while the developers of the UI can focus on the development of custom devices and FIOs. They can thus easily try very different physical configurations of the UI for different combinations of user and environment characteristics.

The use of the GIP and the FIO concept combined with the UsiXML language makes it easy to test a system using devices from different manufactures, with different physical distribution schemes and even try different and multiple simultaneous interaction modalities for the same interaction action. Developers are only required to change the mapping between the AIUs of the user interfaces and the FIOs. A change that can be easily implemented at deploy time, as the mapping is specified in an XML configuration file.

With the idea of using Dandelion for distributed UI prototyping, basic user monitoring capabilities are being integrated into the framework. For each system run, Dandelion generates a log file with the actions of the user and the UI, specifying which interaction has been activated, in which FIO and the time of the activation. Developers can use this information to compare different UI implementations and how the users respond to them.

In the next section, we are going to present a simple illustrative example of how Dandelion can be used to prototype and implement a distributed physical UI.

4 Using Dandelion for DPUI Prototyping: An Example Use Case

In our laboratory we have been working on the implementation of a life assistant application for elderly people. This assistant is an AmI system that can be deployed in the different environments where the user lives, like her own home and the home of her relatives. In order to engage elderly people to use the system, it is very important that they find the system natural and easy to use. Because of this, the interaction must be adapted to the user needs and characteristics, without forgetting the environment

characteristics. Therefore, the system will not use the same devices and interaction technologies in the case of a blind user or a deaf user, and it will not use the same devices in the user home as in a relative's home, as probably the former will have dedicated devices that are not available in the latter.

To simplify the presentation we have selected the UI of a small subsystem of this life assistant application. It is the notification system of the assistant, which is in charge of notifying events, alarms or messages to the users, for example if they have a meeting with the doctor, they are going to receive a visit at home or they have to take some medications.

The UI of the system is a fairly simple one. It only needs to output a message to the user and receive a confirmation. In a classical PC system, it could be a simple notification dialog with a label and a button to accept the message. But if we think of this UI in the terms of Dandelion, that is, in terms of abstract interaction operations, it needs an output interaction to show the message to the user, and an action interaction coming from the user to represent the action of confirming the message. The following code shows how this simple interface could be described using UsiXML. It only requires an output AIU and a trigger AIU that are related inside a container using a compound AIU.

```
<AbstractCompoundIU id="1" shortLabel="Notif. Dialog">
 <AbstractDataIU id="2" shortLabel="Notif. Label">
  <AbstactOutputIU\>
 <\AbstractDataIU>
 <AbstractTriggerIU id="3" shortLabel="Discard action">
  <AbstractOperationIU\>
 </AbstractTriggerIU>
</AbstractCompoundIU>
```

[UsiXML code describing the user interface of the example confirmation dialog.]

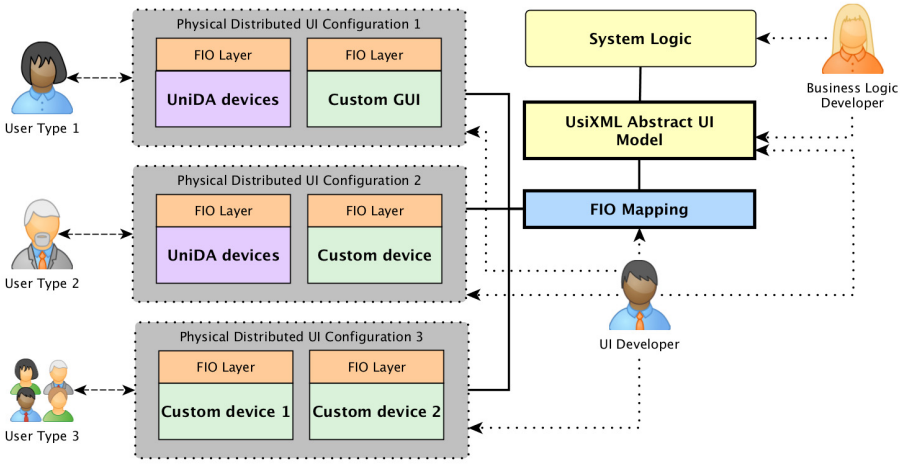

Fig. 3. Different physical configuration of the UI can be easily developed and tested. It is only necessary to implement new FIOs and change the FIO mapping of the application.

The next step for the developer is to connect the application logic with the UI. As described in section 2, this is achieved programmatically at runtime. The developer associates application data or action objects to the different AIUs described in the UsiXML file. In this example, a string object, that will contain the message that must be shown, is associated to the AIU with id '2', and a callback action, that must be executed when the user confirms the reception of the message, to the AIU with id '3'.

Until now, the developer has implemented the system logic and the system UI, and all has been done at an abstract level. The system now needs some FIOs that implement the real interaction with the user, and the developers can now take advantage of the decoupling between the UI definition and its realization in order to test different physical configurations of the UI.

For the illustration of this example, two different configurations have been implemented and tested: First a configuration of the UI for blind users, and then a configuration for deaf users. Figure 4 shows a picture of the physical setup and the relation between the physical interaction devices and the application through the FIO mapping process. As can be seen, five FIOs have been implemented to support these scenarios:

- *outputs*: colored lights using UniDA, an overlay GUI shown in a TV display, and voice synthesis using Festival [14]
- *inputs*: a light switch connected to a KNX network [15] accessed using UniDA and a hand gesture detected by a Kinnect like device

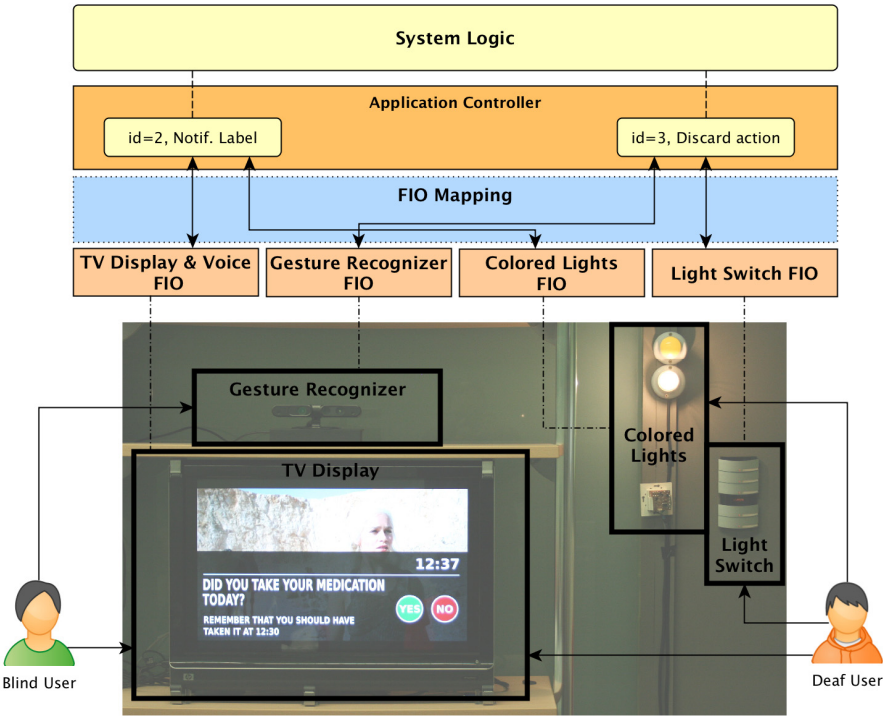

Fig. 4. Notification UI of the life assistant application adapted for deaf and blind users

In the home of a deaf user, the output AIU could be mapped simultaneously to a colored light and to a TV display. As shown in Figure 5, when the application changes the value of the message object, the UIC will send an output GIP event to the colored light and TV FIOS. When the event is received by the FIOs, the colored light will be turned on, so that the user knows that there is a notification, and can go to the living room to read it on the TV. To confirm the message she could activate a light switch deployed in the wall, which will publish an action GIP event that will be received by the UIC. While in the home of a blind user, the output AIU will be associated to the voice synthesizing FIO, and the trigger AIU to the Kinnect hand gesture recognizer.

```
10:43:34.413 | changeIO(message) | AIU(2)('Notif. Label')
10:43:34.867 | output(message) | FIO(1)('colored light')
10:43:34.985 | output(message) | FIO(2)('display')
10:46:17.146 | action() | FIO(3)('switch')
10:46.17.227 | trigger('msgConfirmationCallback')
```

[Example of UI monitoring log generated for the example described.]

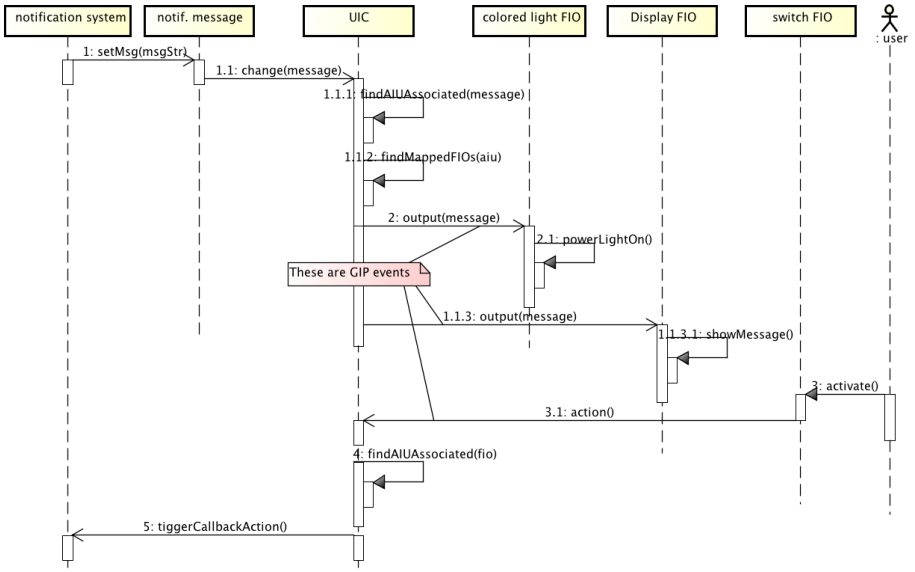

Fig. 5. Sequence diagram illustrating the process of showing a notification message to the user in the physical configuration of the UI for deaf users

5 Conclusions

This paper has presented the Dandelion framework and its prototyping capabilities. As has been shown, Dandelion is able to effectively decouple the system intelligence and system interaction logic from the complexities, modalities and specific control logic of

the devices (physical or not) used to implement the user interaction. It does so by using a distributed device abstraction layer that encapsulates the behavior of devices behind a remote interface of generic interaction actions, and a distributed UI management system that allows the easy modification of the final UI at deployment time.

The ease in changing the physical implementation of the UI without affecting the system and interaction logic makes Dandelion an interesting tool for prototyping user interaction in distributed systems, especially when physical user interfaces are involved, like in Ambient Intelligence systems or Ubiquitous Computing systems.

References

1. Augusto, J.C., McCullagh, P.: Ambient Intelligence: Concepts and Applications. Int'l J. Computer Science and Information Systems 4(1), 1–28 (2009)
2. Dadlani, P., Peregrin Emparanza, J., Markopoulos, P.: Distributed User Interfaces in Ambient Intelligent Environments: A Tale of Three Studies. In: Proc. 1st DUI, pp. 101–104. University of Castilla-La Mancha (2011)
3. Varela, G.: Autonomous adaptation of user interfaces to support mobility in ambient intelligence systems. In: Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS 2013, pp. 179–182. ACM, New York (2013)
4. Varela, G., Paz-Lopez, A., Becerra Permuy, J.A., Duro, R.J.: The Generic Interaction Protocol: Increasing portability of distributed physical user interfaces. In: Revista Română de Interacţiune Om-Calculator, vol. 6(3), pp. 249–268. ACM SIGCHI Romania (2013)
5. Thevenin, D., Coutaz, J.: Plasticity of user interfaces: Framework and research agenda. In: Proc. INTERACT 1999, pp. 110–117. IOS Press (1999)
6. Balme, L., Demeure, A., Barralon, N., Calvary, G.: Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. In: Markopoulos, P., Eggen, B., Aarts, E., Crowley, J.L. (eds.) EUSAI 2004. LNCS, vol. 3295, pp. 291–302. Springer, Heidelberg (2004)
7. UsiXML, `http://www.usixml.org`, `http://www.usixml.eu`
8. Blumendorf, M., Lehmann, G., Albayrak, S.: Bridging models and systems at runtime to build adaptive user interfaces. In: Proc. 2nd EICS 2010, pp. 9–18. ACM (2010)
9. Abascal, J., Fernández de Castro, I., Lafuente, A.L., Cia, J.M.: Adaptive interfaces for supportive ambient intelligence environments. In: Miesenberger, K., Klaus, J., Zagler, W.L., Karshmer, A.I. (eds.) ICCHP 2008. LNCS, vol. 5105, pp. 30–37. Springer, Heidelberg (2008)
10. Ballagas, R., Ringel, M., Stone, M., Borchers, J.: iStuff: A physical user interface toolkit for ubiquitous computing environments. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 537–544. ACM, New York (2003)
11. Varela, G., Paz-Lopez, A., Becerra, J.A., Vazquez-Rodriguez, S., Duro, R.J.: UniDA: Uniform Device Access Framework for Human Interaction Environments. Sensors 11(10), 9361–9392 (2011)
12. UniDA: Uniform Device Access framework, `http://www.github.com/GII/UNIDA`
13. STOMP, Simple Text Oriented Messaging Protocol, `http://stomp.github.io/`
14. The Festival Speech Synthesis System, `http://www.cstr.ed.ac.uk/projects/festival/`
15. KNX Technology, KNX Association, `http://www.knx.org/knx-en/knx/technology/introduction/index.php`