The Computer Journal Advance Access published March 13, 2009

doi:10.1093/comjnl/bxp014

Ambient Intelligence for Supporting Task Continuity across Multiple Devices and Implementation Languages

FABIO PATERNÒ*, CARMEN SANTORO AND ANTONIO SCORCIA

ISTI-CNR, Via Moruzzi 1, 56124 Pisa, Italy
**Corresponding author: fabio.paterno@isti.cnr.it*

Nowadays users are surrounded by environments characterized by an abundance of devices and appliances that they use for carrying out their everyday activities. In order to improve user experience, the surrounding environments should be intelligent enough to allow users to freely move around and be able to perform their tasks in a continuous manner, without forcing them to start their interactive session from scratch at each device change. In this paper, we present an agent-based solution able to support migration of interactive applications among various devices, including digital TVs and mobile devices, and therefore useful for users freely moving about in the home and outdoor. The aim is to provide users with a seamless and supportive environment for ubiquitous access in multi-device contexts of use. The proposed open architecture for migratory user interfaces (UIs) is composed of several agents. It exploits their functionalities and is able to automatically build logical descriptions of existing interactive Web applications and then dynamically generate UIs that are adapted to various types of target devices and implementation languages, including non-Web languages, with the state updated to the point at which it was left off in the previous device.

Keywords: multi-device environments; user interface adaptation; migratory interactive services

Received 21 January 2008; revised 6 February 2009
Handling editors: Fariba Sadri and Kostas Stathis

1. INTRODUCTION

The vision of Ambient Intelligence (AmI) (see, for example, [1]) is that the users operate in intelligent environments, which are aware of users' needs and able to assist, even proactively, the users in performing their activities and reaching their goals. To this end, one important aspect is the possibility for a user surrounded by multiple devices to freely move about and continue the interaction with the available applications through a variety of interactive devices (i.e. cell phones, PDAs, desktop computers, digital television sets, intelligent watches and so on). Indeed, in such environments one big potential source of frustration is that people have to start their session over again from the beginning at each interaction device change. Continuous task performance implies that interactive applications be able to follow users and adapt to the changing context of use while preserving their state. Thus, migratory user interfaces (UIs) require integrated solutions able to address state persistence and UI adaptation when the user changes the device. There are many applications that can benefit from migratory interfaces: in general, applications that support tasks requiring some time to complete and for which a device change will therefore probably be useful (such as games, making reservations) or applications that have some rigid deadline and thus need to be completed wherever the user is (e.g. online auctions).

With the proliferation of different devices in recent years, users will expect nearby devices to be used more and more opportunistically to access data and applications without being tied or restricted to a specific device, but as belonging to a single, shared information workspace. Migratory interfaces enable and empower such a concept of seamless task performance through different devices. Users should be allowed to carry out their tasks regardless of the device they are using, without the burden of having to redo already partially performed tasks to catch up to the point where they left off when they changed device. Therefore, they should be able to perceive the task carried out through different devices as a continuous, uninterrupted activity. This concept is highly innovative and appears capable of provoking some remarkable changes in the way people live, with

many benefits in the medium and long term, and the potential to significantly impact business processes, as well as private life.

While some work on supporting migratory interfaces has already been introduced in [2], here we present a novel solution that extends the previous work in many respects: an agent-based, service-oriented architecture that facilitates the maintenance and extension of the environment, support for a wider set of target platforms that can use a diverse set of implementation languages, and more flexible algorithms to implement more efficient state preservation and effective interface adaptation. In particular, we have designed a new modular environment through the exploitation of a service-based architecture, and also enhanced the state information that can be preserved, which was deemed useful for re-creating on the target device the UI at the state it was before triggering migration. Here we present how the solution proposed has been encapsulated in a multi-agent-based architecture, which supports dynamic UI generation for different platforms (digital TV (DTV), mobile device, desktop) using different implementation languages. With the support of this infrastructure, users can normally access the application and then ask for a migration to any device that has already been discovered in the migration environment. Migration among devices supporting different interaction modalities has been made possible thanks to the use of a logical language for UI description that is independent of the modalities and a number of associated transformations that incorporate design rules and take into account the specific aspects of the target platforms.

In this paper, we first discuss related work, introduce our approach and the logical languages used, and provide more detailed description of our solution through an example. Then, we explain the rules that control the behaviour of the agents that support reverse and forward transformations in order to adapt UIs to the device at hand and to preserve UI state persistence across multiple devices. We conclude by reporting on a user evaluation of the migration approach involving the PDA and DTV platforms, and drawing some conclusions along with indications for future work.

2. RELATED WORK

In recent years, a number of approaches have addressed the problem of interacting with applications in environments characterized by a wide variety of interactive platforms. SUPPLE [3] generates adaptive UIs taking functional specifications of the interfaces, a device model and a user model as input. The remote solver server, which acts as the UI generator, is discovered at bootstrap by the client devices, which can then request interface rendering once it is discovered. In this approach, discovery is limited to the setup stage of the system, and it does not monitor the run-time status of the system, thus losing some of the benefits that could arise from a continuous monitoring activity. Other researchers have investigated the use of overview techniques for supporting adaptation to mobile devices. For example, Lam and Baudish [4] proposed summary thumbnails, which consists in a thumbnail view of the original Web page, but the texts are summarized enabling a good legibility (fonts are enlarged to a legible size and characters are cropped from right to left until the sentence fits on the respective area). WebSplitter [5] aims at supporting collaborative Web browsing by creating personalized partial views of the same Web page depending on the user and the device. Developers have to specify the Web content in XML and define a policy file indicating the tags content that can be presented depending on the user and the device. In addition, they have to define XSL modules in order to transform the XML content into HTML or WML. At run-time, a proxy server generates a presentation depending on user's privilege and the access device. In this approach developers have to manage a plethora of low-level details to specify XML content, policy files and XSL transformations. More generally, all such approaches do not support migration of a UI from one device to another, but provide only solutions for adaptation among different platforms.

In ICrafter [6], services beacon their presence by periodically sending broadcast messages. A control appliance then requests a UI for accessing a service or an aggregation of services by sending its own description, consisting of the UI languages supported (i.e. HTML, VoiceXML) to an entity known as the Interface Manager, which then generates the UI and sends it back to the appliance. ICrafter shares with our system the consideration of a multi-modal approach to user interaction, but it does not support the transfer of the UI from one platform to another, maintaining the client-side state of the interface.

The issues related to device adaptation raised interest in model-based approaches for UI design and generation, mainly because they provide logical descriptions that can be used as a starting point for generating interfaces that adapt to the various devices at hand. In recent years, such interest has been accompanied by the use of XML-based languages, such as UIML [7], UsiXML [8], TERESA XML [9], in order to express the aforementioned logical descriptions. However, most of such approaches focus on providing device-adaptation support only in the design and authoring phase, while we believe that run-time support is equally relevant, since in this way it is possible to dynamically exploit the characteristics of the various devices, which is not a negligible aspect especially when mobile devices are considered.

The personal universal controller (PUC) [10] automatically generates UIs for remote control of domestic appliances. The remote controller device is a mobile device, which is able to download specifications of the functions of appliances and then generate the appropriate UI to access them. XML-based messages are exchanged to request a description of the appliance's features and to signal events to and from the device. PUC applies the model-based approach to generating UIs in mobile devices but mainly for controlling appliances that usually have poor UIs (such as printers, DTV players, …).

Bharat and Cardelli [11] addressed the migration of entire applications (which is problematic with limited-resource devices and different CPU architectures or operating systems) while we focus on the migration of the UI. Newman and others [12] have developed the Speakeasy recombinant computing framework, which is a middleware exploiting mobile code to allow components in different devices to extend one another's behaviour at run-time. However, they have not addressed the issue of adapting the services UI to the interaction resources available in the device at hand, which we address through device-independent languages (and related transformations).

Aura [13] provides support for migration but the adopted solution has a different granularity. In Aura, for each possible service, various applications are available and the choice of a particular application depends on the interaction resources available in the user device. Thus, for a given service, such as word processing, if a desktop PC is available then an application such as Microsoft Word can be activated, whereas in the case of a mobile device a light-weight text editor would be chosen.

Luyten and Coninx [14] present a system for supporting distribution of the UI over a federation or group of devices. *Migratability*, in their words, is an essential property of an interface and marks it as being continuously redistributable. Their paper indicates the suitability of task model-based design of UIs and XML-based UI description languages for supporting UI distribution and migration in device federations. The authors consider migration and distribution of only graphical UIs for desktop and mobile systems, while we provide a solution supporting migration for a broader set of interactive platforms (such as the DTV).

Another aspect that is emerging from the combination of virtual environments, mobile communication and sensors is the AmI vision: a pervasive and non-obtrusive intelligence in the surrounding environment supporting the activities of users. AmI emphasizes aspects of high user-friendliness and support for more intelligent services. Software agent technology is promising in this field, and should have a major role in AmI development thanks to software agent characteristics, such as autonomy and mobility. A software agent is a program acting on behalf of a person or organization. Indeed, one of the central problems in AmI societies, which are highly open and dynamic, is to find rules adopted by the agents encapsulating numerous types of objects and devices in order to guarantee efficient and relevant collective behaviour. To manage the dynamics and the heterogeneity of AmI systems (such as workload, failures and interoperability of the devices, as well as their addition or suppression), these agents must enable the system to adapt to every context. To this regard, in [15] an example (Seta2000) is provided about how an infrastructure based on a multi-agent architecture can facilitate the provision of adaptable ubiquitous Web-based services, by showing how it is able to cope with the complexity and the needed flexibility required in pervasive and personalized systems. However, Seta2000 does not provide support for task continuity and has very limited provision for UI adaptation to the interaction resources of the current device.

In [16] the implementation of the multi-access service platform has been carried out based on the foundation for intelligent physical agents (FIPA)-compliant [17] multi-agent system (MAS) architecture 'Java intelligent agent componentware' (JIAC). The goal is to allow the fast and easy creation of multi-modal UIs based on an abstract description language. Even their approach is not able to support migratory UIs. In particular, their platform has no support for interrogating the state of the UI in the client devices, which is fundamental for obtaining continuity across multiple devices. O'Hare *et al.* [18] discussed the use of intelligent agents as a promising approach to managing both the traditional explicit interaction modality and, where necessary, the implicit interaction modality. Implicit interaction offers software designers an alternative model for capturing user intent and significant opportunities to proactively aid the user in the fulfilment of their tasks. In our case, implicit interaction occurs when the user is nearby an available device and the migration platform can automatically trigger migration to it if the current device for some reason is no longer suitable to continue the user session. Ranganathan and Campbell [19] proposed a middleware for context-aware agents for ubiquitous computing. For this purpose, they use ontologies to describe context predicates, and then agents can acquire various types of contextual information and reason about it in order to adapt their behaviour. In our work, one important aspect of context that is dynamically considered is the set of interaction resources of the device that is the target in the migration process. Our agents are able to dynamically generate first logical descriptions and then the implementation of the UI taking into account them. In [20] the authors present an approach to a reactive behavioural system that not only facilitates building a context representation, but also uses that knowledge to provide relevant services to the user. However, they support some mechanism to control who can use which resources, but not for adapting UIs to the varying interface resources.

In our approach, the smart behaviour of the system is based on software agents that perform a number of intelligent activities. Their roles differ not only in their internal behaviour, but also in the external interaction they have with the other components. For example, one *proactive* agent (device discovery) autonomously and continuously performs the monitoring of the current context; at each context change, it evaluates the new scenario and, if better conditions are recognized, suggests to the user the possibility for migration. Such better conditions could mean for instance that a better device (in terms of interaction resources, and/or processing power, memory and network capability, etc.) has become active and available for migration, or another one has been freed by the previous users. In addition, in our architecture we also have *transformational* and *reactive* agents, whose purpose is to react to requests by, for example, transforming the UI description for a given platform into a description suitable for a platform with

different interaction resources (this is the case, for instance, of the semantic redesign agent). Such transformations are obtained following a number of rules associated with each type of UI element or structure. In the next section we will better describe the activities that are carried out by such agent-based software modules.

3. OUR APPROACH

Migration is the result of state preservation and adaptation to the device interaction resources. Such features have to be supported while users interact with the applications made available by the intelligent environment. For this purpose, our migration architecture supports a number of reverse and forward transformations that are able to transform existing desktop Web applications for various interaction platforms and support task continuity. The basic assumption is that there exists a huge amount of easily accessible content for desktop Web applications, which can be processed and transformed to support migratory interfaces, even across non-Web implementation languages. The client devices subscribe to the migration service by running a migration client agent that provides information regarding the device characteristics. The devices access Web applications through the migration server, which includes proxy functionalities. Migration can be triggered either by the user (through the migration client that allows the specification of the target device by either a graphical interface or scanning RFID tags associated with the target device) or it can be automatically triggered by the smart environment when some specific event (such as very low battery or connectivity) is detected (see Fig. 1), or in a mixed solution in which the environment suggests possible migrations based on the devices available and the user decides whether or not to accept them. In the case of automatic identification of the migration target, there is an agent in the migration infrastructure that is able to

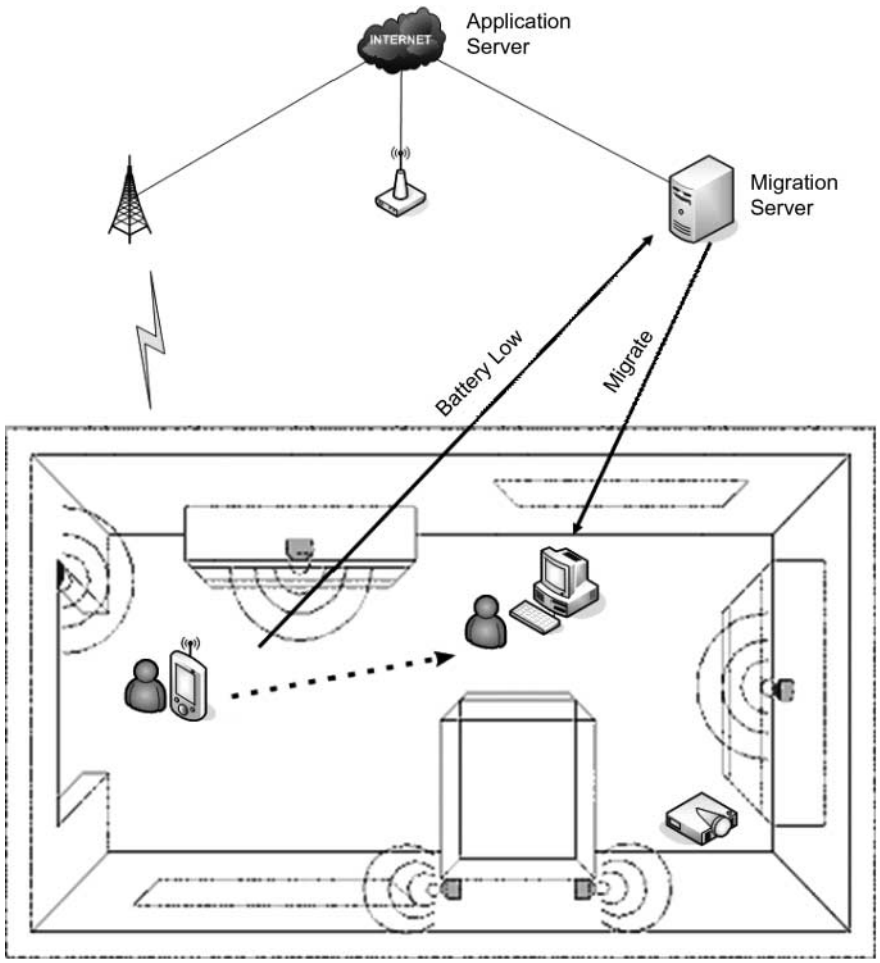


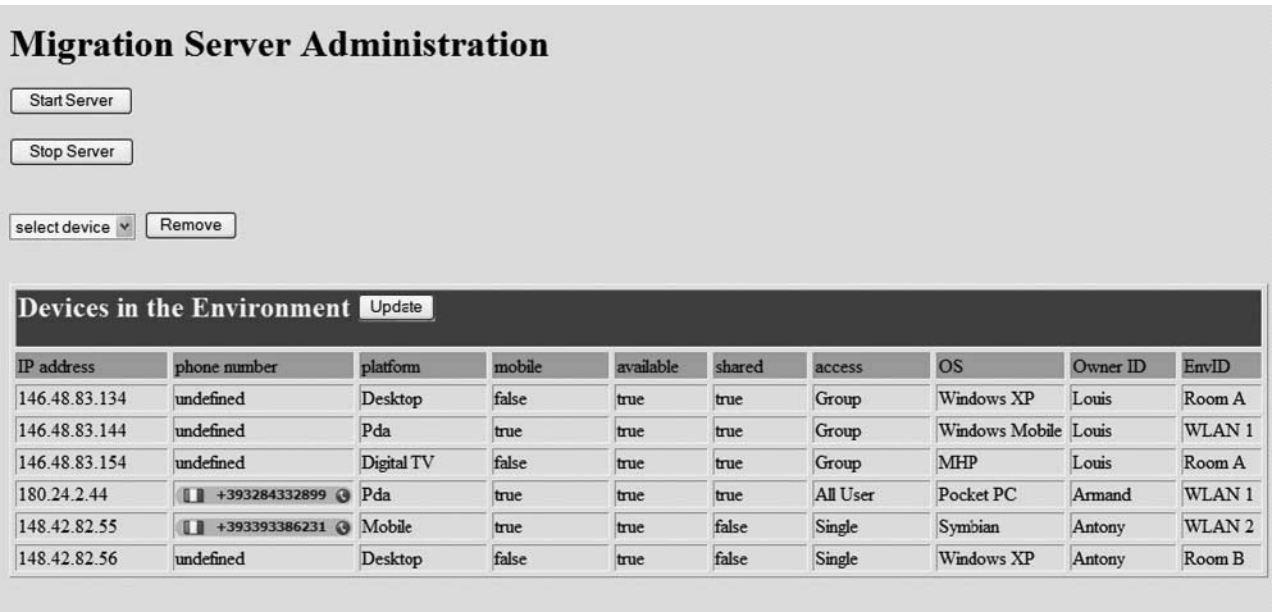
FIGURE 1. The migration environment.

analyse the characteristics of the available devices and identify the most suitable according to a number of rules. Such rules consider the device descriptions, which include, for example, an indication whether the device is personal (in this case specifying the associated user) or can be used by members of a group or by all members; the state of the device (for example, if it is a single user device and already taken by another user then it cannot be considered available for migration); information regarding where the device is located (in the case of stationary devices it is the corresponding room). Figure 2 shows the information that the migration server dynamically provides regarding the devices that are available for migration.

The migration platform has been designed using an agent-based, service-oriented architecture, and implemented using Web services. This means that the main functionalities have been encapsulated in the following agents that can communicate with the external world through XML-based interfaces:

- *Device discovery*, which is the module in charge of discovering the devices currently available for migration;
- *Trigger manager*, which decides when the migration has to be activated;
- *Reverse engineering*, which builds logical descriptions from the desktop Web implementations;
- *Semantic redesign*, which transforms the logical description of the source UI into the logical description for the target device;
- *State mapper*, which associates the state of the source UI to the logical description for the target device;
- *User interface generator*, which generates the UI implementation for the target device;
- *Migration device agent*, which runs on each device notifying its presence/availability and provides information on the state of the UIs running on its device;
- *Migration manager*, which orchestrates the general behaviour of the system.

When the user accesses the application through an interaction platform other than the desktop, the server transforms its UI by building the corresponding logical description and using it as a starting point for creating the implementation adapted to the accessing device. In addition to interface adaptation, the environment supports task continuity. To this aim, when a request for migration to another device is triggered, the environment detects the state of the UI, which depends on the user input (elements selected, data entered) and identifies the last element accessed in the source device. Then, a logical version of the interface for the target device is generated, and the state detected in the source device version is associated with the target device version so that the user inputs (selections performed, data entered) are not lost. Lastly, the UI implementation for the target device is generated and activated remotely at the point corresponding to the last basic task performed in the initial device. In the process of creating an interface version suitable for a platform different from the desktop, we use a semantic redesign agent. This part of the migration environment automatically transforms the logical description of the desktop version into the logical description for the new platform in order to provide a description of the UI suitable for the new platform. To this aim, intelligent rules are used for adapting the description of the UI to the new platform taking into account its capabilities (e.g. using



IP address	phone number	platform	mobile	available	shared	access	OS	Owner ID	EnvID
146.48.83.134	undefined	Desktop	false	true	true	Group	Windows XP	Louis	Room A
146.48.83.144	undefined	Pda	true	true	true	Group	Windows Mobile	Louis	WLAN 1
146.48.83.154	undefined	Digital TV	false	true	true	Group	MHP	Louis	Room A
180.24.2.44	+393284332899	Pda	true	true	true	All User	Pocket PC	Armand	WLAN 1
148.42.82.55	+393393386231	Mobile	true	true	false	Single	Symbian	Antony	WLAN 2
148.42.82.56	undefined	Desktop	false	true	false	Single	Windows XP	Antony	Room B

FIGURE 2. Migration server monitoring the devices available.

constructs that are suitable for the new platform) but ensuring at the same time that the original goal is maintained. This solution allows the environment to exploit the semantic information contained in the logical description and obtain more meaningful results than transformations based only on the analysis of the specific UI implementation languages. In this case the semantic information is related to the basic tasks that the UI elements are expected to support.

The current architecture of the migration environment, described through the inter-agent communications is introduced in Fig. 3. Each client device has a software agent, which serves, amongst other things, to announce the availability of the device for the migration (1–2 arrows in Fig. 3) to the device discovery manager agent, which in turn updates the list of available devices (with related information) in order to keep it up-to-date, and then sends this information to the migration manager (3a). This communication is also used to provide information regarding events that occur in the devices during the user session and can be relevant for migration (for example, a device's battery is running out of charge). When a browser on a client device (e.g. a PDA) tries to access the desktop version of a Web page the request is filtered by a proxy server, which accesses the application server to obtain it and then asks the migration manager to start the process for delivering the appropriately modified page version to the PDA. This means performing the reverse engineering of the desktop version and then the semantic redesign of the resulting concrete description for the considered platform (the PDA in this case). As a result of this process, the version suitable for the PDA is delivered to the client device. This process is iterated for each new page that is requested by the device.

The migration can be triggered either by a user request or proactively by the trigger manager agent (note that in Fig. 3 the same label (4) has been used for both kinds of trigger), which can identify situations where it would be better for the user to change device. At this point, the migration manager asks the reverse engineering agent to get the (5a) Web page (desktop version), in order to produce the corresponding concrete description (5b). Once this logical description is obtained, the migration manager asks the semantic redesign agent (6a) to perform the semantic redesign of that page for the target platform. Then, the semantic redesign agent performs such transformation to create a logical description for the target platform (6b), whose state needs to be updated. This is done by the state mapper agent (7a and 7b). As soon as the state mapper has retrieved the logical presentation and has updated it with the latest state information, the state mapper sends the new concrete user interface (CUI) for the target device to the migration manager. Then, the migration manager sends this CUI to the UI generator module (8a and 8b), which transforms this logical description into a final UI. The result of the process (namely: the final UI) is then sent to the target client (9) where it is rendered and made available to the user.

Table 1 shows an excerpt of pseudocode describing the rule that is adopted by the migration manager agent after a client requests a migration of a page (*active_URL)* towards to *target_Platform* device (the case of a DTV has been detailed in Table 1, but other platforms like PDA, cellphone, etc. have also been considered). As you can see, the migration manager agent communicates with other agents (implemented through Web Services, see e.g. calls to "RedesignWS" and "ReverseWS" in Table 1) for performing the necessary transformations. In

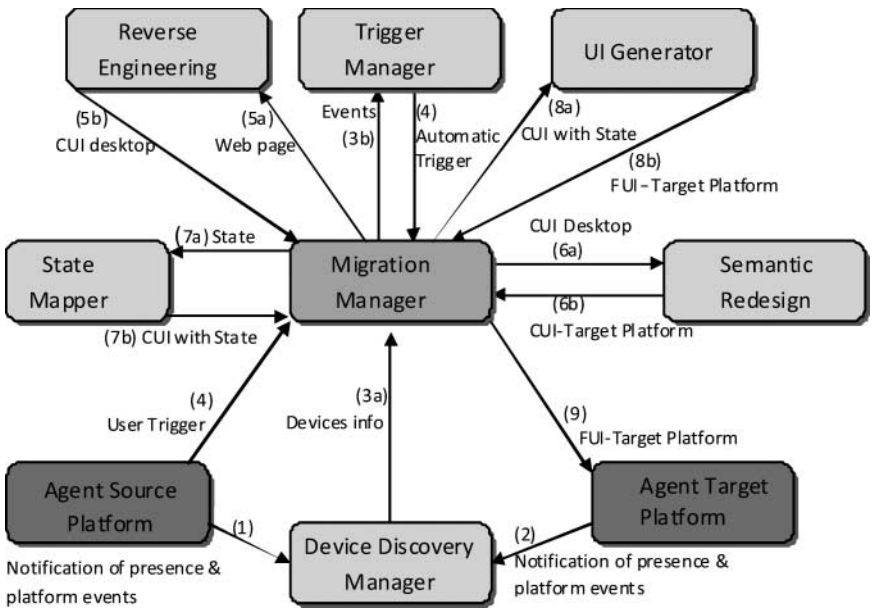


FIGURE 3. The main agents of the migration platform.

TABLE 1. Pseudocode for a rule of the migration manager agent, after a migration request is received.

```
Rule migrateTowardsTV {
  //the migration manager starts migration considering
  //the received request and the current state
  Migration.migrate(source_Platform, target_Platform,
active_URL, state_Data)
targetDeviceType =
retrieveDeviceType(target_Platform)
  //retrieving information on target device
  page=RetrieveDesktopPage(activeURL);
  //retrieve the desktop version of the active URL (page)
  if (targetDeviceType == TV
  then
  {
  CUI_Desktop = ReverseWS(page);
  //activate the reverse engineering agent on the page
  // redesign the desktop page for the new platform: a new
CUI is produced
  targetPlatform_CUI = RedesignWS(Desktop, TV,
CUI_Desktop);
  // activate the state mapper agent for updating
targetPlatform_CUI with state_Data
  StateMapperWS(targetPlatform_CUI, state_Data);
  finalPage = callGeneratorWS(targetPlatform_CUI);
  // call the generation of final UI for the new platform
  UpLoadPage(finalPage); // upload the final page
  }
}
```

particular, we use two levels of UI logical descriptions according to the CAMELEON reference framework [21]: the *concrete one*, which is dependent on the platform type, and the *abstract one*, which describes interfaces in a completely platform-independent way. In practice, the concrete level acts as an intermediate level between the implementation and the abstract, which is more semantic-oriented description.

For example, when deciding how to associate the state of the source UI to the target device, our migration infrastructure looks for the abstract elements corresponding to the implementation elements that have been modified in the source version, then identifies the elements used in the target version for the implementation of such abstract elements, and lastly associates the updated state to these elements.

Likewise, the migration infrastructure identifies the point at which the target UI should be activated: the migration environment identifies the last input entered and the corresponding abstract element, and then looks for its implementation in the target version. The presentation including such element will be activated in the target device, in order to allow the users to continue the tasks in the new device from the point where they left off in the source device, so allowing the users to have the feeling of seamlessly interacting with the devices existing in the ambient.

Since the elements of the concrete description are defined taking into account the features of the corresponding platforms, but are independent of the implementation language, our approach also supports interoperability between various implementation languages. Thus, for example, a Web application can be transformed into a Java application for the DTV. In this case the generation of the UI implementation involves the generation of a file in a Java version for DTVs representing an Xlet, which is an application that is immediately compiled and can be interpreted and executed by the interactive DTV decoders.

4. UI LOGICAL DESCRIPTIONS

To represent the logical description of UIs we use a revised version of the TERESA XML language [9]. In this language, an abstract UI is composed of a number of presentations and connections among them. Figure 4 shows a graphical representation of the highest levels of the TERESA XML abstract language. While each *presentation* defines a set of abstract interaction techniques perceivable by the user at a given time (which might be elementary interactors or compositions of such interactors), the *connections* define the dynamic behaviour of the UI, by indicating what interactions trigger a change of presentation and what the next presentation is. The abstract element types are divided into two groups: interaction (single choice, multiple choice, numerical edit, text edit, position edit, activator, navigator, interactive description), and only output (text, object, description). In addition, at the abstract level we also describe how to compose the basic interface elements through some composition operators (see Fig. 5). Such operators can involve one or two expressions, each of them can be composed of one or several interactors or, in turn, compositions of them. In particular, the composition operators have been defined taking into account the type of communication effects that designers aim to achieve when they create a presentation [22]. They are (see Fig. 5): Grouping, indicating a set of interface elements logically connected to each other; Relation, highlighting a one-to-many relation among some elements, one element has some effects on a set of elements. In a grouping two additional properties can be defined: ordering, some kind of ordering among the set of corresponding elements can be highlighted; hierarchy, different levels of importance can be defined among the set of corresponding elements. All such abstract elements are described through an XML device-independent language.

Figure 5 shows the details of the specification of the *interactor_composition* element as well as the *interactor* element. The XML elements are represented by rectangles drawn with a continuous line, while the attributes are represented by dotted rectangles. For sake of readability several

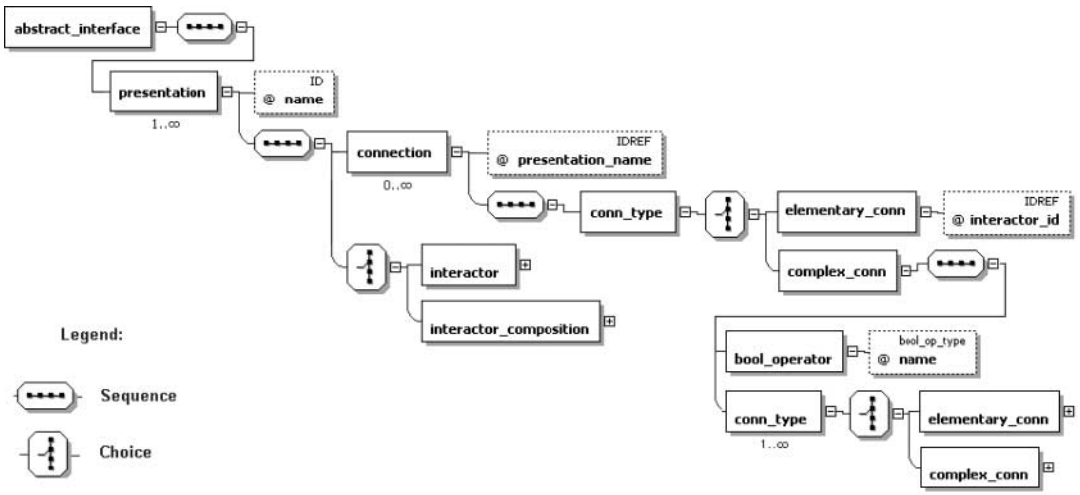


FIGURE 4. The highest levels of an abstract UI.

attributes are not shown. While at the abstract level no particular platform is considered, at the concrete level there is a further refinement of the abstract UI depending on the type of platform considered, which allows adding further details on how to support a certain task. For example, in the case of a graphical desktop platform a navigator can be implemented either through a textlink, or an imagelink or a simple button, and in the same way, a single choice object can be implemented using either a radio button or a list box or a drop-down list. Likewise, a username on a GUI can be rendered through a textual label, whereas onto a vocal platform it will be rendered through a vocal utterance. This information (how a specific piece of content will be rendered) is exactly the kind of support that the concrete level offers to the specification of the UI. Indeed, this level defines in which way the UI object will be presented, although the implementation of such an object is not yet specified.

For instance, at the concrete level, for a textual label, it is not yet specified whether it will be implemented in XHTML or JAVA: this information is defined in the final level of the UI, which is its implementation. The same holds for the composition operators, so the grouping operator can be refined at the concrete level by a number of techniques including both unordered lists by row and unordered list by column, fieldsets (which are rectangles including the composed elements), bullets, and colours. The smaller capability of a mobile phone might not allow implementing the grouping operator by using an unordered list of elements horizontally listed in a row, and then this technique might not be available on this platform. In a vocal device, a grouping effect can be achieved through inserting specific sounds or pauses or using a specific volume or keywords delimiting the grouped elements. Another advantage of this approach is that maintaining links among the elements in the various abstraction levels allows the possibility of linking semantic information (such as the activity that users intend to do) and implementation levels, which can be exploited in many ways. A further advantage is that if a new implementation language needs to be addressed, only the transformation from the corresponding concrete level to the new language has to be added. This is not difficult because the concrete level is already a detailed description of how the interface should be structured.

5. AN EXAMPLE APPLICATION

The example considered in this case is a shopping application. The scenario is a user returning home from work, who starts to prepare an online shopping list through a mobile device (while s/he is on the bus or train) and then when s/he gets home, s/he may check what is in the house and realizes that some items are still missing. Then, s/he completes the list by interacting with the DTV while sitting comfortably on the sofa. Thus, using the PDA, the user can access the page dedicated to the products and specify the category of interest ('meat' in our case, see Fig. 6a). Depending on the selected category, the application allows a further refinement of the selection. In our example, users are allowed to select the kind of meat they want to buy by choosing among beef, poultry and pork. Then, a number of options are displayed together with the associated amounts (see Fig. 6b), and users can start to select what they want to buy.

When the user gets home, the smart environment (through the proactive trigger manager agent) suggests migrating the UI to other devices that have been recognized as available in the new environment.

Indeed, the agent-based architecture has recognized a situation where the interactions could be made more comfortably (e.g. the user could use the desktop PC that has a larger screen, or the TV while sitting on the sofa) and it suggests such possibility to the user. Then, if the user decides to migrate the UI to the DTV, s/he can continue editing the shopping list through a larger screen without having to save the

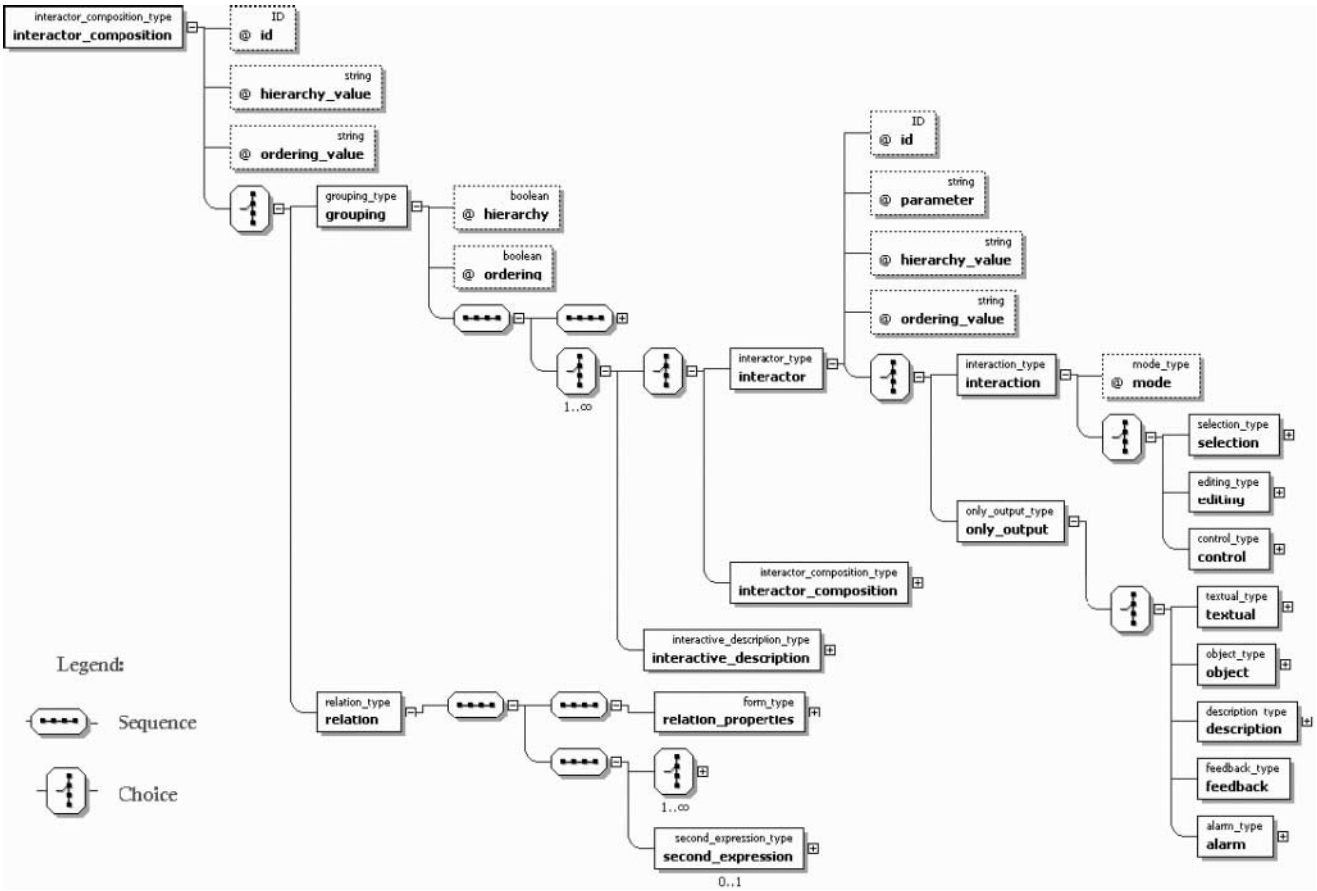


FIGURE 5. Specification of an interactor composition in TERESA XML.

current selections from the PDA and login to the application again from the new device.

The combination of agents involved in this phase of migration (i.e. reverse engineer, semantic redesign, state mapper and UI generator) will be in charge of providing the user with the new page on the target device, with each agent adopting its own rules. Such rules have already been summarized in Section 3, and will be described in greater detail in the following. More specifically, in the example considered the reverse engineer agent will create the logical description of the desktop page version, then the semantic redesign module will adapt it for the new platform (the DTV), the state mapper will associate the current state (which was captured on the PDA device), and the UI generator agent will provide the final implementation for the target device (DTV). After the interface migration, the user will find the items that were entered earlier through the PDA (see Fig. 7, the request for three beef steaks, which was made using the handheld device) and edit them or add new ones until satisfied, when s/he can send the request to buy them. Figure 8 shows how the general approach introduced in previous sections actually works for the considered shopping example. The process shown in Fig. 8 assumes that a request has been made from a PDA to access a XHTML desktop page (arrow (0)). Then, the migration platform (identified by the dashed square in Fig. 8) captures such request and reverse-engineers the page (see arrow (1)) by producing a logical specification (*Shopping-desktop.LUI*) of the original page. This specification contains *logical* information that regards both the abstract and the concrete level; therefore, we have decided to refer to it by using a '.LUI' extension. The *Shopping-desktop.LUI* specification is then transformed through a semantic redesign step (arrow (2)) into a logical specification for the PDA platform (*Shopping-pda.LUI*). The latter specification is then converted into an implementation adapted for such device (*Shopping-PDA.html*) through a forward engineering step (arrow (3)). When a migration request is sent from the PDA (arrow (4)), indicating that the target device is the DTV, the state of the UI is retrieved and sent to the migration platform.

At the same time, the logical specification of the desktop interface (*Shopping-desktop.LUI*) is accessed again in order to exploit a more 'semantic', higher-level description of the UI, to be adapted for the new device through a semantic redesign stage (arrow (5)). The goal of such a phase is to keep the logical structure of the source and target interfaces

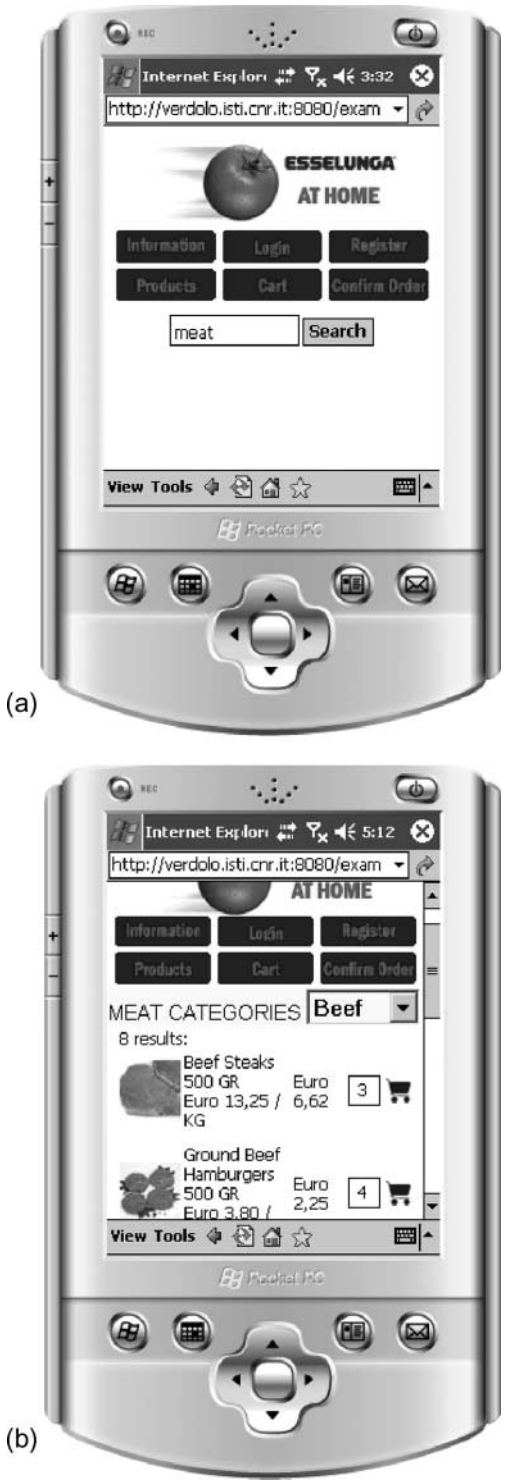


FIGURE 6. The PDA interface of the shopping application: (a) the products page and (b) the purchase page.

consistent in terms of basic tasks supported and, at the same time, adapt the UI to the different interaction resources of the target device. The result of the semantic redesign stage is *Shopping-tv.LUI*. The migration platform retrieves from the

FIGURE 7. The DTV application interface after migration.

source device (in this example the PDA) the *state* resulting from the user's interactions (arrow (6)) through a client-side AJAX script included in the page by the migration platform. It associates such state with the new logical description created for the target platform (*Shopping-tv.LUI).* This description is then used to generate (arrow (7)) the final implementation of the UI for the target device, which consists in a pre-compiled java class (*Shopping-TV.class*) that is activated in the target device (arrow (8)). Then, the application is interpreted by the set-top box and, lastly, rendered on the target platform (the DTV). It is worth pointing out that while translating the UI from a source platform to a target platform, it is necessary to maintain support for the intended interactions in the target platform, in order to ensure that the UI remains semantically consistent during this transformation. The reverse engineering agent is in charge of mapping the elements written in the desktop implementation onto corresponding higher-level logical objects, which better identify their associated semantics (namely, the task supported). Indeed, since the different abstract elements are classified according to their semantics and this semantics is shared across the different platforms, it is possible to retain the intended objective of an interaction technique available in the source platform by supporting its semantics through an appropriate technique available on the target platform. The advantages of this mechanism can be easily seen in our example. The semantic redesign of the UI supported by the migration approach leads to some changes in how the UI is rendered when moving from the PDA to the DTV. For instance, on the PDA, due to the smaller screen size, the selection of the different types of products is supported by a combo-box (which displays only one choice at a time, see Fig. 6), while the larger screen size of the DTV allows for supporting the same task through a radio-button (which shows all the choices at the same time but for this reason requires more screen space, see Fig. 7). Another difference between the source device (PDA) UI and the target device (DTV) is the fact that the semantic redesign enables supporting an additional task on the DTV, namely the possibility of getting further details on

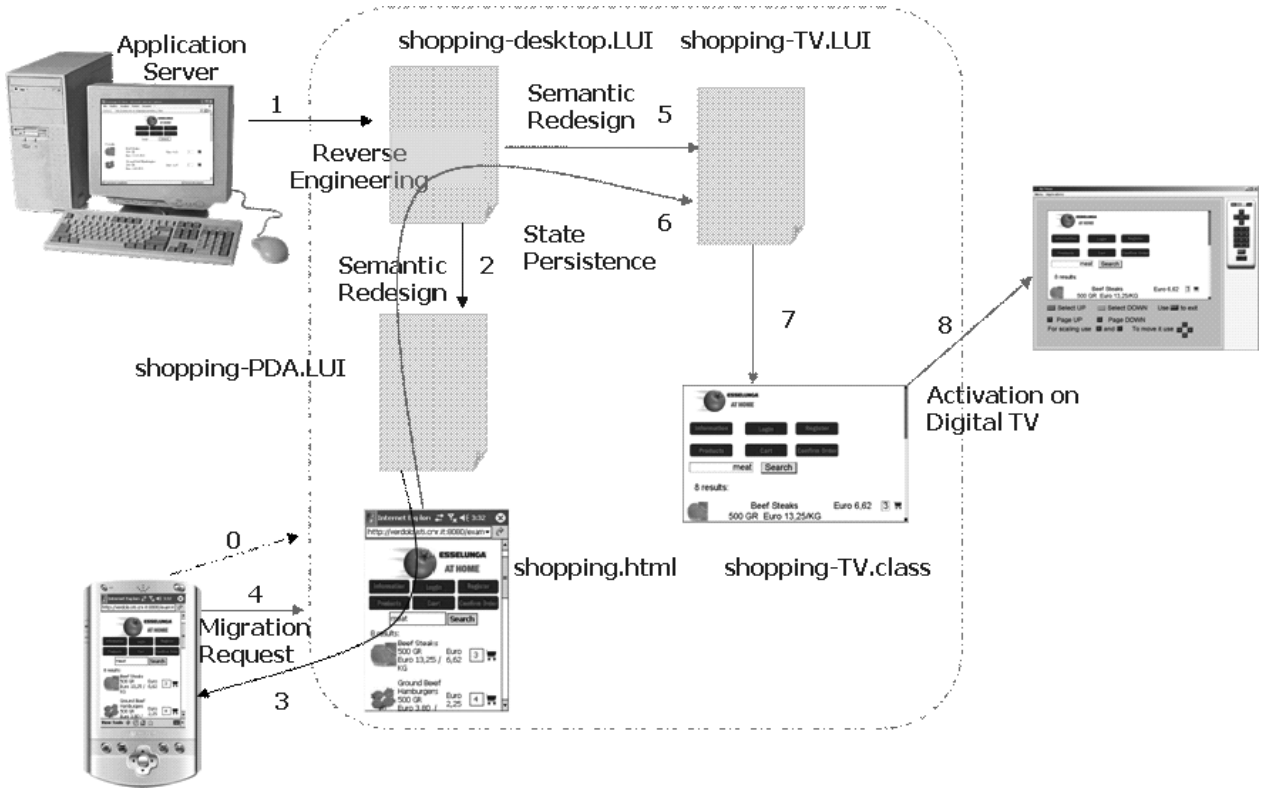


FIGURE 8. The migration transformations in the example.

the different products (see 'Details' button in Fig. 7), which was not supported on the source device (the PDA). Its availability in the target device is made possible by the presence at the abstract level of a logical description of the activities that should be carried out depending on the platform considered, so that only the tasks that are deemed significant on each specific platform are supported. In the above case, because of its small screen size, providing such details was deemed unsuitable for a PDA and is thus not supported. Therefore, it is the semantic redesign agent that has to determine from the logical description which tasks are inappropriate for the new platform and remove the associated interaction object (s).

The semantic redesign of the migration approach performs further changes not only on the type of elementary objects of the UI, but also in structuring its layout: as you can see comparing Figs 6 and 7, the larger screen space available on the DTV allows for arranging buttons in a single row, which is not the case for the PDA device.

6. REVERSE ENGINEERING AGENT

The main purpose of the reverse engineering part is to analyse the implementation of the existing Web application desktop version, capture the logical design of the UI (in terms of basic tasks supported and the ways to appropriately structure the UI in order to accomplish them), which will then be used as the starting point for the design and generation of the interface for the target device. Some work in this area has been carried out previously. For example, WebRevEnge [23] automatically builds the task model associated with a Web application, whereas Vaquita [24] and its evolutions build the concrete description associated with a Web page.

The reverse engineering agent can reverse both single XHTML pages and whole Web sites, including the associated CSS stylesheets. When a Web page is reversed into a presentation, its elements are reversed into different types of concrete interactors and combination of them by recursively analysing the DOM tree of the X/HTML page. In order to get it, well formed X/HTML files are needed. However, since many of the pages available on the Web do not satisfy this requirement, before reversing the page, the W3C Tidy parser (http://tidy.sourceforge.net/) is used for correcting features like missing and mismatching tags and returns the DOM tree of the corrected page, which is analysed recursively starting with the body element and going in depth. Depending on the type of node analysed, the reverse engineering algorithm follows one of the following branches:

- The X/HTML element is mapped onto a concrete interactor. This is a recursion endpoint. The appropriate interactor element is built and inserted into the

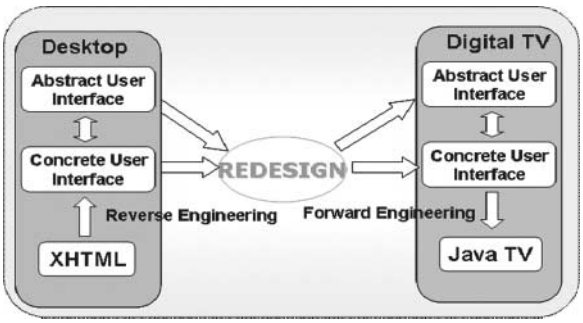


FIGURE 9. Architecture of the adaptation part for DTV.

XML-based logical description. For example, DOM nodes corresponding to the XHTML tags <img>, <a> and <select> cause the generation of concrete objects of type respectively *image*, *navigator* and *selection*. The properties of the objects in the Web implementation are also used to fill in the attributes of the corresponding concrete UI elements, so that this information can be elaborated for producing an appropriate element in the target device, out of the peculiarities used in the source Web page (for instance, the *italic* attribute of a text concrete element is set to true although in the X/HTML implementation it might appear as either <i> or).

- The X/HTML node corresponds to a composition operator. In this case, the proper composition element is built and the function is called recursively on the X/HTML node subtrees. The subtree analysis can return both elementary interactors and composition of them. In both cases the resulting nodes are appended to the composition element from which the analysis started. For example the node corresponding to the tag <form> is reversed into a *Relation* composition operator and <ul> into a *Grouping*. Depending on the analysed node, appropriate attributes are also stored in the resulting element at the concrete level (e.g. typical X/HTML desktop <ul> lists will be mapped at the concrete level in a *grouping* expression using *bullets* listed following a *vertical* positioning).
- The node does not require the creation of a new element in the concrete specification (e.g. if in the Web page there is a definition of a new font, no new element is added in the CUI). If the node has no children, no action is taken and it is a recursion endpoint (e.g. this can happen with line separators such as
 tags). If the node has children, each child subtree is recursively reversed and the resulting nodes are collected into a *grouping* composition, which is in turn added to the result.

In the reverse process, the environment first builds the concrete description and then the abstract one. In TERESA XML the concrete descriptions are a refinement of the abstract one, which means that they add a number of attributes to the higher-level elements defined in the abstract descriptions. Thus, the process for reversing a concrete description into the corresponding abstract one consists in removing the lower-level details from the interactor and composition operators specification, while the structure of the presentations and the connections among presentations remain unchanged. In practice, there is a many-to-one relation between the elements of the concrete UI and the abstract UI (both for the interactors and the composition operators): the concrete UI indicates several ways to refine and abstract element for the platform under consideration. Therefore, it is easy to derive the abstract logical objects corresponding to the different concrete interaction objects. For instance, considering the desktop platform, we can have at the concrete level a *text_link*, an *image_link* and a *button*, which are all possible refinement options for a *navigator* interactor. However, since all such elements share the same objective (navigating between different parts of the UI), the result of reversing each of these concrete elements will be an abstract *navigator* object.

7. SEMANTIC REDESIGN AGENT

The semantic redesign agent changes the logical description of a UI for a given platform into a logical description for a different platform. The aim is to support a similar set of tasks and communication goals but provide input for obtaining an implementation that adapts to the interaction resources available.

In particular, the semantic redesign agent analyses the input from the desktop logical descriptions and generates an abstract and concrete description for the target platform, from which it is possible to automatically generate the corresponding implementation. Figure 9 shows the process in the case of adaptation from desktop to DTV.

Figure 10 shows the various phases of semantic redesign in the case of desktop-to-mobile transformations. There are three main steps: (i) transforming the desktop logical interface into a mobile logical interface; (ii) calculating the resulting cost in terms of resources and (iii) splitting the logical interface into presentations that fit the cost sustainable by the target device. In the first transformation the concrete elements of the desktop description are substituted by concrete elements supported by the mobile platform (for example, a radio-button with several elements can be replaced with a pull-down menu, which occupies less screen space, see Fig. 10). In this transformation, further rules are applied to adapt the elements of the UI to the characteristics of the new platform even when the transformation from the source platform to the target platform does not change the type of interactor. For instance, images originally displayed in the source (desktop) platform are resized according to the screen size of the target (mobile) device, while keeping the same aspect ratio. In some cases they may not be rendered at all because the resulting resized image would be

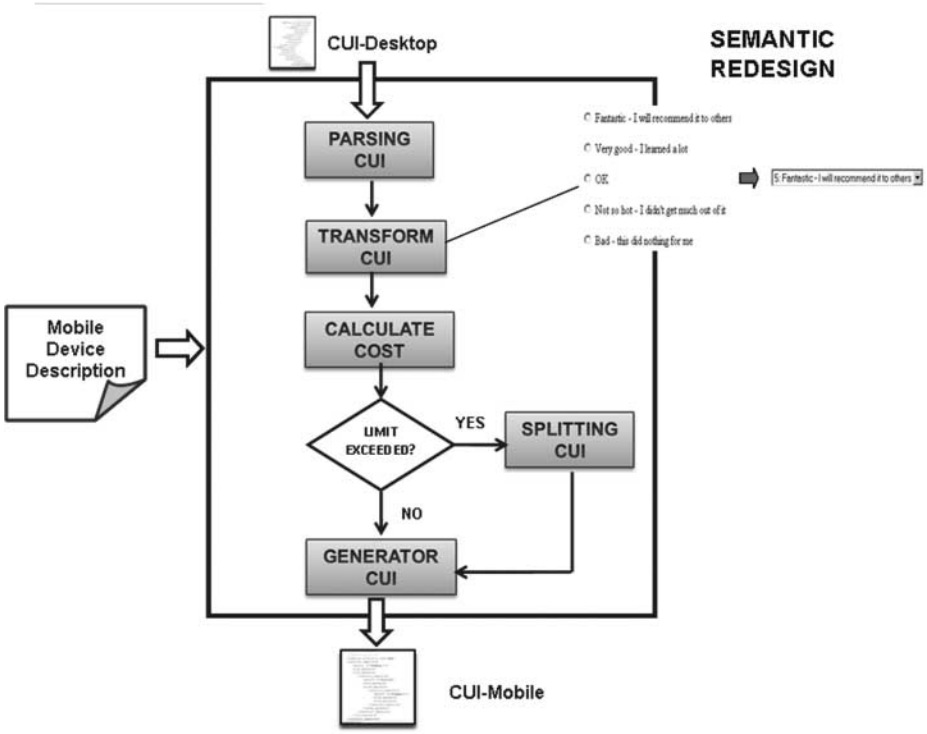


FIGURE 10. Desktop-to-mobile semantic redesign.

too small or the mobile device does not support them. Text and labels can be transformed as well, since they may be too long for mobile devices. In converting labels, we use conversion tables to identify shorter synonyms or abbreviations.

In order to automatically redesign a desktop presentation for a mobile device, we need to consider semantic information and the available resource limitations. If we only consider the physical limitations, we may end up dividing large pages into smaller ones that are not meaningful, since they result from considering only some aspects, e.g. the available space in the screen of the target device. To overcome this problem, we also consider the composition operators indicated in the logical descriptions. Indeed, our algorithm tries to maintain in the same presentation interactors that are collected together through some composition operators. For instance, suppose that a user has to specify her personal information for receiving updates/news from a low cost airline newsletter: in this case all the UI objects supporting the editing/showing of the user's personal information are logically grouped together. Such a logical grouping should be reflected into an adequate presentation in the final UI in such a way to convey the semantic relationship that exist among the various objects also through opportune visualizations so that the user can easily understand that they altogether contribute to achieve the same specific (sub-)goal and they are all grouped together (e.g. on GUI a typical technique is using a graphical fieldset for grouping together all the grouped fields). Thus, the environment aims to preserve the communication goals of the designers and obtain interfaces that are easy to use because each presentation is composed of objects that are semantically related to each other in that they all contribute to achieve a specific goal (or subgoal). In addition, the division of pages according to the logical completion of a task (or a subtask) also allows for maintaining the consistency of UIs through different devices, which is especially convenient for users who interact with the same application through different devices (as happens with migratory UIs). Page splitting requires a change in the navigation structure with the need for additional navigator interactors for accessing the newly created pages. More specifically, the algorithm for calculating the costs and splitting the presentations accordingly is based on the number and cost of interactors and their compositions. The cost is related to the interaction resources consumed, e.g. number of pixels, font sizes, etc. After the initial transformation, which replaces the desktop concrete elements with mobile concrete elements (for example, a text area for the desktop platform could be transformed into a simpler text edit on the mobile platform), the cost of each presentation is calculated. If it fits the cost sustainable by the target device, then no further processing is required. Otherwise, the presentation is split into two or more pages following this approach: the cost of each composition of elements is calculated. The one with the highest cost is associated to a newly generated presentation and is replaced in the original presentation with a link to the new presentation. Thus, if the cost of the original presentation after

this modification is under the maximum allowed cost, then the process terminates, otherwise it is recursively applied to the remaining compositions of elements. In the case of a complex composition of interface elements, which might not be entirely included in a single presentation because of its high cost for the target device, the algorithm aims to distribute the interactors equally amongst presentations of the mobile device. Figure 10 shows the various phases of the semantic redesign in the case of desktop-to-mobile transformation.

The cost that can be supported by the target mobile device is calculated by identifying the characteristics of the device through the *user agent* information in the HTTP protocol, which can be used to access more detailed information in a local XML repository with device descriptions obtained through WURFL (wurfl.sourceforge.net/), a device description repository containing a catalogue of mobile device information. Initially, we considered UAProfiles but sometimes such descriptions are not available or are wrong and they require an additional access to another server where they are stored. As already mentioned, examples of elements that determine the cost of interactors are the font size (in pixels) and number of characters in a text, and image size (in pixels), if present. One example of the costs associated with composition operators is the minimum additional space (in pixels) needed to contain all its interactors in a readable layout. This additional value depends on the way the composition operator is implemented (for example, if a grouping is implemented with a fieldset or with bullets the costs are associated with the space taken by the surrounding rectangle or the bullets). Another example is the minimum and maximum interspace (in pixels) between the composed interactors.

The semantic redesign module can take into account the different features of the modalities that can be supported. For example, in vocal interfaces, it is important that the system always provides feedback when it correctly interprets a vocal input and it is also useful to provide meaningful error messages in the event of poor recognition of the user's vocal input. At any time, users should be able to interrupt the system with vocal keywords (for example 'menu') to access other vocal sections/presentations or to activate particular features (such as the system reading a long text).

8. STATE MAPPER AGENT

The state refers to the information entered by the user, but also other pieces of information that can be important for the user session (such as the history and cookies). State persistence is necessary to support task continuity across multiple devices. For this purpose, when clients access the Web pages, their requests pass through a proxy server, which downloads the pages from the application servers, and also annotates them with scripts able to capture the UI state. We can distinguish two types of scripts:

(1) Scripts that are executed any time a new page is downloaded and accessed by the user, when s/he navigates normally through the various pages (which means: without triggering any migration). Such scripts have to execute two basic functionalities: (i) through a polling-based monitoring mechanism, implemented through an AJAX script, they determine whether a migration was triggered by the migration client and (ii) through a piece of invisible script code (since it uses a IFRAME element) get the cookies and the current history, store their contents, and continuously update such information to maintain the state consistent in an automatic and transparent way. This information (e.g. the history, the cookies) is useful when the migration target still uses a Web-based implementation language.
(2) Scripts that are executed only when there is a migration request. Indeed, when a migration trigger is generated, an AJAX callback function is automatically activated and sends the DOM (containing the state of the current page) collected through a specific script, together with the content collected through the IFRAME previously mentioned (e.g. history, cookies). In particular, when a request for migration to another device is triggered, the environment detects the state of the UI as modified by the user input (elements selected, data entered, etc.), and identifies the last element accessed in the source device. This is obtained by the JavaScript functions that are automatically inserted by the proxy server and are able to manage the collection of the information describing the state of the migrating page by accessing its DOM. The information is collected in a string formatted following an XML-based syntax and sent to the server through an AJAX script.

The source device in the migration must be running a light software (the migration client), whose purpose is to allow the user to trigger the migration and select the migration target. When the migration is triggered, the IP of the source and target device are sent to the migration manager. The script included in the Web page also contains an AJAX script that performs a polling in order to know whether any migration has been triggered for that application. In the positive case, the Web application is informed and activates the state information transmission. This mechanism was chosen because only an application running on the browser in the client device can access the application DOM, and the AJAX Script can transmit the data without requiring any explicit action from the user.

Then, the migration platform will first associate the content state of the page on the source device to the concrete description of the version for the target device, and, in the case of migration to a Web-based target, it adds a new portion of invisible content containing the AJAX functions to re-create the additional state features (such as history and cookies) on the target client device in the corresponding generated implementation. The

objective of the state mapper agent is to update the CUI for the target device (which has been produced by the semantic redesign module) with latest information regarding the state of the UI contained in the DOM file of the source page just before migration. The corresponding elements in the two files are easy to identify because each object of the CUI has a unique identification label (ID), which is the same of the corresponding XHTML/DOM element from which that CUI element was generated by the reverse engineering process. One possible complicating factor is when the semantic redesign has transformed a specific concrete object C1 (for a specific platform) into a different concrete object for the target platform, C2. In this case, since the same ID is maintained among the two concrete objects C1 and C2, the association between the concrete object and the corresponding DOM element is still straightforward (the same ID is maintained). Nevertheless the state mapper may require a further step, that is, adapting the value of the DOM element to specify the new concrete object. For instance, it might happen that, as a result of the semantic redesign process, a radiobutton element was translated into a pulldown menu element. Therefore, the values included in the specification of the radiobutton element (e.g. the different items of the radiobutton) have to be appropriately adapted and used to fill in the specification of the pulldown menu element.

9. UI GENERATOR AGENT

This module is in charge of building the UI in an implementation language suitable for the target device, starting with a concrete description of the UI for the platform considered. Indeed, depending on the considered interaction platform and the specific implementation language, a particular transformation is triggered. Table 2 provides some pseudocode excerpts describing the algorithm underneath the UI generator transformational agent for the DTV platform.

9.1. Generating UIs for DTVs

In order to show that our approach can support even non-Web implementation languages we have considered Java for the DTV. In terms of UIs, this platform poses different issues with respect to traditional desktop systems. Indeed, DTV might be considered quite similar to a graphical desktop (due to the dimensions of the available screen), but we have to take into account that TV users have no mouse or keyword but just a TV controller and that the user typically sits at a certain distance from the TV screen, which is generally greater than the distance between a user and a PC screen. As a consequence, a specific kind of font—Tiresias—was used, due to its high readability on TV displays. In addition, in order to guarantee the best readability of the text, quite high font dimensions were selected (ranging between 24 and 36 points), avoiding the smaller ones, which do not guarantee sufficient visibility.

TABLE 2. Pseudocode for the UI generator agent.

```
GeneratorWS(TransformedCUI) {
  document = DocumentBuilder(TransformedCUI); //parse
XML definition
  //start parsing from root element (which is 'presentation')
of XML definition
rootElement = getRootElement(document); //create instance
of DTVPresentation that represents presentation node
presentation = createDTVPresentation(rootElement);
for each child (rootElement) {
  if (child is CompositionNode) {
    generateCompositionNode(child, presentation); }
  else { generateNode(child, presentation);
    }
  }
compileClass();
return class;
}
generateCompositionNode(node, pres) {
  if (node Type is GROUPING) {
    pres.addGrouping();
    for each node child {
      if (child is CompositionNode){
    generateCompositionNode(child, presentation);}
    else {
    generateNode(child, presentation);
    }}}}
else if (node Type is RELATION) {
    pres.addRelation();
    for each node child {
          if (child is CompositionNode)
{generateCompositionNode(child, presentation);}
          else {generateNode(child, presentation); }
    }}
}

generateNode(node, pres) {
    if (node Type is SINGLE_SELECTION) {
      pres.addSelection(RADIO_BUTTON, node); //Single
Selection - Radio Button }
    else if (node Type is MULTIPLE_SELECTION) {
      pres.addSelection(CHECK_BOX, node); //Multiple
Selection - CheckBox");}
    else if (node Type is TEXT_EDIT) {
      pres.addTextEdit(TEXT, node); //Text Edit}
    else if (node Type is NAVIGATOR) {
      generateNavigatorNode(pres, node);}
    else if (node Type is ACTIVATOR) {
      generateActivatorNode(pres, node);}
    .....
}
```

In this case, the generation of the UI involves the creation of a file in a Java version for DTVs (Xlet). Such Xlets bear

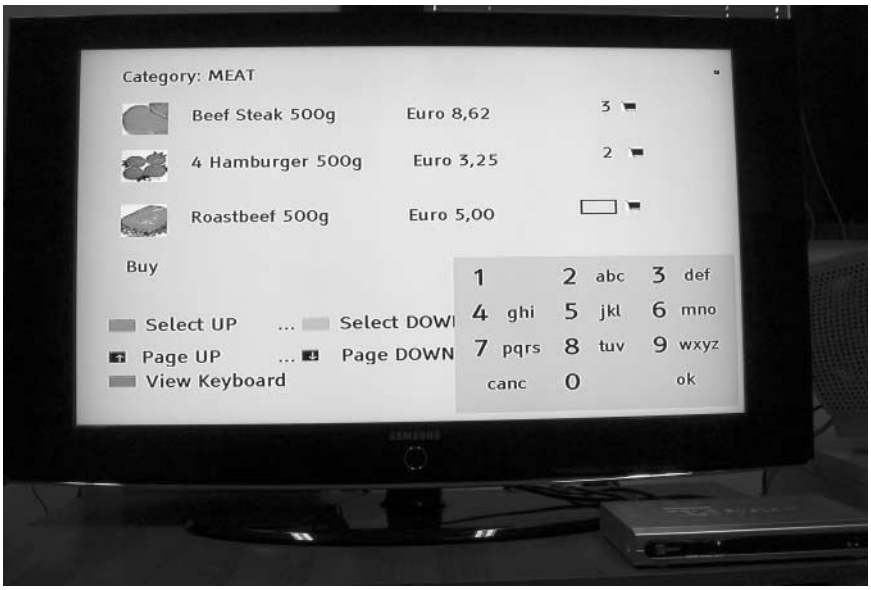


FIGURE 11. The UI on a large screen DTV.

a strong resemblance with common Java applets, with the difference that, instead of the Web browser (which executes the applets) there is the Multimedia Home Platform (MHP, http://www.mhp.org/), an open middleware system standard for interactive digital television, enabling the execution of interactive, Java-based applications on the digital receiver (set-top box). Once the Xlet is generated, it is supposed to be downloaded on the set-top box. If a real set-top box is not available, an emulator can be used to execute the interactive Xlet. For our examples, we initially used the XletView emulator (http://xletview.sourceforge.net) for testing MHP Xlets on a PC, and then switched to the use of a real set-top box connected to a 42″ TV. In our prototype we use a set-top box telesystem TS7.2 DT, which supports MHP 1.0.2 (see Fig. 11). However, there is a great difference between Xlets and Java applets regarding the UI part: Xlets do not include the AWT package, which contains several widgets for Java programs, such as radio buttons, check boxes, buttons, etc. Thus, we also developed a toolkit [25] that provides such techniques and simplifies the UI generation for the DTV platform. The library also includes a class that provides the methods to include the widgets in the application without having to specify all the details every time. Then, there is a class for each interactor implementing its appearance and behaviour. The implementation of the appearance is not trivial because the basic Xlets just provide primitives for drawing rectangles and showing images. Thus, for example, 3D effects are obtained by partially overlapping white rectangles on top of black rectangles, and selected/deselected elements in radio buttons or check boxes are obtained using different images according to the current state. The semantics of the interactor implementations is managed through specific event handlers that allow them to update the state and their appearance accordingly. Thus, for example, in the case of a radio button the corresponding event handler keeps track of the currently selected element (using state variables) and updates the graphical view of the interaction object.

The generated UI for the DTV platform takes into account the specific characteristics of this platform. Indeed, in the case of DTV (example in Fig. 11), adequate space is reserved in the screen for presenting the buttons of the virtual controller through which the user interacts with the e-commerce application. Actually, in the CUI for the TV platform, the semantic redesign module adds an additional set of grouped elements corresponding to the commands needed to control the generated UI. Afterwards, the UI generator has to translate this newly added grouping of concrete UI objects into the Java implementation and then, after the UI generation, the buttons of the virtual controller are rendered in the final UI, using a suitable implementation language for the DTV platform. In addition, while some buttons of the virtual controller are permanently displayed on the screen, others are added dynamically. For instance, when the pointer goes over an area that is editable, the button 'View Keyboard' is visualized in order to enable the opening of a Virtual Keyboard (shown in the bottom-right part of the screen, see an example in Fig. 11) for editing the selected field. However, if the pointer is not over an editable area, the button 'View Keyboard' is not shown at all.

10. USER EVALUATION

We performed an evaluation of our prototype along with the application introduced in the previous section to assess the

migration approach. The test involved 12 people, with an average age of 30 (ranging between 22 and 50), recruited in the institute community (11 males and 1 female). Only three had undergraduate education, while the others had a university degree (or higher). Before starting the exercise, the users read a short text presenting a brief overview of the migration approach (goals, features, etc.). Then, they were provided with a short description of the scenario of the evaluation exercise: the shopping application described in the previous section. In particular, the users were instructed to carry out the following tasks using the software prototype: register with the supermarket Web site via the PDA, and then start to buy some products belonging to a certain category of products available in the supermarket (choosing from meat, pasta, vegetables). Afterwards, the users were instructed to complete the purchase by buying products of a different category and using the DTV. The basic transformations involved in the migration scenario were those described in Fig. 8 and involving the PDA as source device and DTV as target device.

After the exercise, users were asked to fill in a questionnaire, which was divided into two parts: in the first part general information was requested from the user (age, education level, level of expertise on using PDA/DTV systems, etc.), while the second part was devoted to questions more related to the evaluation exercise.

The test subjects reported to be, on average, quite expert in using PDA systems ($M = 2.58$ in a 1–5 scale in which 1 is the worst score and the 5 is the best one; SD $= 1.24$), but not particularly expert in using DTVs ($M = 1.75$; SD $= 0.97$). Users experienced some difficulties using the DTV, and this was generally self-explained and brought back by them to the inexperience in using the DTV emulator, which was used in the test. However, after having understood the correct way of using the emulator most of these problems diminished and users managed to perform the requested tasks.

Focussing more properly on the migration, users found the device discovery application very intuitive and clear in presenting the devices on which the migration can occur ($M = 4.67$; SD $= 0.65$). In addition, the UI redesigned for the DTV, apart the above-mentioned issues, was found clear and intuitive both in navigation ($M = 4$; SD $= 1.04$) and in quality of presentations ($M = 4.25$; SD $= 0.75$). As a further comment, some users appreciated the consistency shown between the UIs of the two devices, which they found helpful in avoiding users' disorientation, especially when the user feels quite unfamiliar with the target device of the migration (as in this case, with the DTV). Indeed, the easiness in continuing the task from the point where the interaction was left before migrating received quite satisfactory consents ($M = 3.42$; SD $= 1.08$). Moreover, the information that was filled in the application before activating the migration was easily re-found by the users when reconnecting with the different device, as they reported in the questionnaire (10 out of 12 reported the maximum value for rating this aspect and the other two judged it in positive way as well, with a final average value of 4.75; SD $= 0.62$). It is worth noting that this aspect is really important in giving the users the effect of a 'continuous' interaction regardless of the device and in recalling them the previous context (especially important when the migration extends over a long period of time). Furthermore, all the users were able to complete the tasks assigned during the evaluation exercise. If we consider as a metrics for evaluation the completion speed, we have to say that when the user decides to migrate to another device, such a decision can be triggered by different reasons. Indeed, there might be reasons of better adequacy of the new device (e.g. benefits in terms of user interaction comfort, etc.), which the user judges more beneficial with respect to the consequences of the possible disruption included in changing the interaction device (e.g. the time needed for performing the migration itself and/or the time needed by the user for adapting his/her model of the application to the features of the new platform). In this situation, especially if the user has already some familiarity with migration features and then s/he already has an idea of which result the migration will produce, the completion speed can still be one of the goals of the users who want to migrate the UI in order to complete more quickly their tasks, especially when such tasks require long time. Indeed, for instance, when migrating from a cellphone to a desktop platform, the possibility for having multiple activities concurrently visualized in the same presentation will probably speed up the completion of the task with respect to navigating through a number of different presentations (as it might occur when using a platform with small capabilities in screen size). However, there are situations in which the completion speed might not be the primary user goal, since the user wants to migrate in order to be able to spend more time on the current activity. In this case, s/he migrates for deliberately being able, to some extent, to increase the task completion time. Such situations occur, for instance, when the user wants to have a larger display and wants to comfortably sit on the sofa in order to linger on the activity currently performed, which is selecting the destination of the next holidays, in order to better enjoy the experience. In this situation triggering migration might increase significantly the completion time, without being a sign of dissatisfaction for the user. The other situation is when the migration is automatically suggested by the intelligent environment. This is driven by external conditions (for instance, the battery consumption of the source device is very low and then continuing the interaction with the source device will be simply not possible in a matter of minutes), or by conditions that have been specified by the user as suitable conditions for migration: they are continuously monitored by dedicated software agents, which suggest the user a possible migration when such conditions are verified.

All in all, users judged the migration intuitive, easy to use and useful ($M = 3.33$; SD $= 1.07$). They judged the application particularly useful for tasks that are likely to require long interactions. Therefore, some of them expressed the willingness in seeing the application of this approach also to different

application domains (e.g. games). Lastly, most of them foresee a better understanding of the potentialities of this approach in the short/medium-term future, in parallel with the further diffusion and penetration of interactive platforms in people's everyday life and the need of seamless access to services, regardless the place (and the time) where (and when) the interaction takes place.

11. CONCLUSIONS AND FUTURE WORK

We have presented an intelligent multi-agent environment able to support migration of UIs for various platforms including DTV, mobile devices and desktop systems.

It is worth pointing out that although in this paper we consider migration from PDA to the DTV, the approach can be extended for any platform, providing that a Web desktop version of the considered application exists and that suitable software agents (migration client, UI generator, etc.) are provided taking into account the characteristics of the devices involved (available interaction resources, implementation languages supported, etc.).

In the current state of implementation, the environment is able to support migration of any Web application written in XHTML and CSS but it cannot reverse engineer Java applets or Flash applications contained in Web pages, which are thus automatically filtered before reversing the Web pages. The environment is then able to generate interfaces after migration implemented in several languages, depending on the target platform: XHTML, VoiceXML, X+V, Java (and support for others such as C#, SVG and SMIL is under development). This solution can be useful for supporting various types of interactive applications able to follow and support the user moving through different devices in the home and outside not only for common everyday tasks, such as shopping, bids for auction on line, games, making reservations, but also for business applications.

We plan to extend this solution to also support migration in multi-user environments.

ACKNOWLEDGMENTS

We thank Renata Bandelloni for work on previous versions of the migration architecture.

FUNDING

This work has been partly supported by the EU ICT STREP FP7-ICT-2007-1 N.216552 OPEN (Open Pervasive Environments for migratory iNteractive Services) project (http://www.ict-open.eu), although the views expressed are those of the authors and do not necessarily represent the project.

REFERENCES

[1] ISTAG (2003). *Ambient intelligence: from vision to reality*. ftp://ftp.cordis.lu/pub/ist/docs/istag-ist2003_draft_consolidated_report.pdf.

[2] Bandelloni, R., Mori, G. and Paternò, F. (2005). Dynamic Generation of Migratory Interfaces. *Proc. Mobile HCI 2005*, Salzburg, Austria, September 19–22, pp. 83–90. ACM Press, New York.

[3] Gajos, K., Christianson, D., Hoffmann, R., Shaked, T., Henning, K., Long, J.J. and Weld, D.S. (2005). Fast and Robust Interface Generation for Ubiquitous Applications. *Proc. UBICOMP'05*, Tokyo, Japan, September 11–14, pp. 37–55. Springer, Berlin.

[4] Lam H. and Baudisch P. (2005). Summary Thumbnails: Readable Overviews for Small Screen Web Browsers. *Proc. ACM CHI'05*, Portland, OR, April 2–7, pp. 681–690. ACM Press, New York.

[5] Han, R., Perret, V. and Naghshineh, M. (2000). WebSplitter: Orchestrating Multiple Devices for Collaborative Web Browsing. *Proc. CSCW 2000*, Philadelphia, PA, December 2–6, pp. 221–230. ACM Press, New York.

[6] Ponnekanti, S.R. Lee, B. Fox, A. Hanrahan, P. and Winograd T. (2001). ICrafter: A Service Framework for Ubiquitous Computing Environments. *Proc. UBICOMP 2001*, Atlanta, GA, 30 September–2 October, pp. 56–75, *Lecture Notes in Computer Science* 2201. Springer, London.

[7] Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S. and Shuster, J. (1999). UIML: An Appliance-Independent XML User Interface Language. *Proc. Eighth Int. World Wide Web Conf.*, Toronto, Canada, May 11–14, pp. 1695–1708. Elsevier.

[8] Limbourg, Q. and Vanderdonckt, J. (2004). UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence. *Engineering Advanced Web Applications*, pp. 325–338. Rinton Press, Paramus.

[9] Mori G., Paternò F. and Santoro C. (2004). Design and development of multi-device user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Eng.*, **30**, 507–520.

[10] Nichols, J. Myers B.A., Higgins M., Hughes J., Harris T.K., Rosenfeld R. and Pignol, M. (2002). Generating Remote Control Interfaces for Complex Appliances. *Proc. ACM UIST'02*, Paris, France, October 27–30, pp. 161–170, ACM Press, New York.

[11] Bharat K.A. and Cardelli, L. (1995). Migratory Applications. *Proc. UIST95*, Pittsburgh, PA, November 15–17, pp. 133–142. ACM Press, New York.

[12] Newman, M.W., Izadi, S., Edwards, W.K., Sedivy, J.Z. and Smith, T.F. (2002). User Interfaces When and Where They are Needed: An Infrastructure for Recombinant Computing. *Proc. UIST'02*. Paris, France, October 27–30, pp. 171–180.

[13] Garlan, D., Siewiorek, D., Smailagic, A. and Steenkiste, P. (2002). Project Aura: toward distraction-free. *IEEE Pervasive Comput.*, **1**, 22–31.

[14] Luyten, K. and Coninx, K. (2005). Distributed User Interface Elements to support Smart Interaction Spaces. *Proc. IEEE Symp. Multimedia,* Irvine, CA, December 12–14, pp. 277–286.

[15] Ardissono L., Goy A., Petrone G. and Segnan, M. (2005). A multi-agent infrastructure for developing personalized web-based systems. *ACM Trans. Internet Technol.* **5**, 47–69.

[16] Rieger, A., Cissee, R., Feuerstack, S., Wohltorf, J. and Albayrak, S. (2005). An Agent-Based Architecture for Ubiquitous Multimodal User Interfaces. *Proc. 2005 Int. Conf. Active Media Technology*, Kagawa, Japan, May 19–21, pp. 119– 24, IEEE Press.

[17] FIPA (2002). *Agent management specification*, *FIPA00023*. http://www.fipa.org/specs/fipa00023.

[18] O'Hare, G.M.P., Keegan, S. and O'Grady, M.J. (2006). Interaction for Intelligent Mobile Systems. *Proc. KES 2006*, Bournemouth, UK, October 9–11, pp. 686–693, Lecture Notes in Artificial Intelligence 4252. Springer, Berlin.

[19] Ranganathan, A. and Campbell, R.H. (2003). A Middleware for Context-Aware Agents in Ubiquitous Computing Environments. *Proc. Middleware 2003*, Rio de Janeiro, Brazil, June 16–20, pp. 143–161, Lecture Notes in Computer Science 2672. Springer, Berlin.

[20] Hanssens, N., Kulkarni, A., Tuchida, R. and Horton, T. (2002). Building Agent-Based Intelligent Workspaces. *Proc. Int. Conf. Internet Computing*, Las Vegas, NV, June 24–27, pp. 675–681, CSREA Press.

[21] Deliverable D1.1 (2002). *The CAMELEON reference framework*. CAMELEON Project.

[22] Mullet, K. and Sano, D. (1995). *Designing Visual Interfaces*. Prentice-Hall, Englewood Cliffs, NJ.

[23] Paganelli, L. and Paternò, F. (2003). A tool for creating design models from web site code. *Int. J. Softw. Eng. Knowl. Eng.*, **13**, 169–189.

[24] Bouillon, L. and Vanderdonckt, J. (2002). Retargeting Web Pages to other Computing Platforms. *Proc. WCRE'2002*, Richmond, VA, 29 October–1 November, pp. 339–348. IEEE Computer Society Press, Los Alamitos.

[25] Paternò, F. and Sansone, S. (2006). Model-Based Generation of Interactive Digital TV Applications. *Proc. MoDELS'06, Workshop on Model Driven Development of Advanced User Interfaces*, Genoa, Italy, October 2, ISSN 1613-0073.