An Interface Agent for the Management of COTS-based User Interfaces

Jose F. Sobrino, Javier Criado, Jesus Vallecillos, Nicolas Padilla and Luis Iribarne
Applied Computer Group, University of Almeria, Almeria, Spain
{jfsobrino, javi.criado, jesus.vallecillos, npadilla, luis.iribarne}@ual.es

Keywords: Interface Agents, Component, User Interfaces, Widgets, Interaction.

Abstract: The great development of the knowledge society on the Internet requires that Web information systems are adapted at runtime to user groups with common interests. Interface agents help us to observe and learn from user preferences making interfaces adaptable to user working habits. We propose an interface agent which works on Web interface based on COTS components, adapting the interface to the user needs or preferences. Our agent runs two main behaviors: observation behavior which analyses the user interaction on the interface and a second behavior which runs the adaptation actions to adapt the user interface at runtime.

1 INTRODUCTION

Due to the rapid expansion of the information and knowledge society on the Internet, Web-based Information Systems (WIS) must be prepared to be easily adaptable, extensible, accessible and manageable at runtime by different people with common interests. In this type of system it is important to have interfaces that facilitate human-computer interaction, promoting the dynamism design of components adapted to the users habits and the development of their work. Interface agents have become a technology widely used in the development of interfaces adapted to the user needs. Such agents have the ability to observe and learn from the preferences and work habits to provide an UI adaptation. Our interface agent works on Web-Ul based on Commercial Off-The Shelf (COTS) interface components of widgets-type. These interfaces offer a great versatility and adaptability making the adaptation an easy process for the interface agent.

Our research proposes an interface agent that looking over the shoulder of the user (Maes, 1994), collecting each event or action that user performs on the components and executing changes in the interface model. Each action performed on a component by user is interpreted and classified by the interface agent, deciding whether the action changes the model. These changes involve an adaptation process. The adaptation process is not performed by interface agent, but this is done within a adaptation engine described in (Rodriguez-Gracia et al., 2012a) and (Criado et al., 2012). This article does not detail this engine, limiting us to use its input/output. After obtaining the adaptation actions that provides the adaptation engine, our interface agent makes the changes on the interface. This type of interface agent that helps to adapt and evolve interfaces is very useful, since it shows to user a friendlier interface when performs his tasks. The use of widgets components gives us a lot of dynamism in interface design due to encapsulation of functionalities and properties.

The rest of the paper is structured as follows. Section 2 describes the type of user interface where our interface agent works. Section 3 details the two main behavior of user interface. Section 4 gives some related work. Finally, in Section 5 describes some conclusions and future works.

2 PREVIOUS SCENARIO

In a world more and more interconnected, the Web interfaces should be flexible and are prepared for an easy adaptation to the new type of users that work on the Internet. However today, Web interfaces are being built statically based on traditional Web development paradigms. We propose to build a Web interface with self-contained visual components, interchangeable and modifiable, varying its appearance or properties at runtime, depending of the user interaction on the components. We assume that there is an UI components market (Heineman and Councill, 2001) (Lau, 2004) that we use as source of our components. These components are stored in public repositories, being prepared to be assembled and put into
operation. These interfaces are increasingly proliferating in interface types smarts for phones and smart TV, which are other particular interface where our approach can also be applied.

Our Web interface bases on the composition at runtime of widgets-type interface components, offering the possibility of reorganizing the composition of the interface at runtime without losing functionality. The components are organized at the interface with a predetermined pattern that will be adapted with the user interaction. Figure 1(a) shows a interface model that represents this concept. Each component of the model can change its visual appearance or disposition within the interface, or even to be replaced by another component with different characteristics.

In Figure 1(b), we show an example of Web interface for the environmental management about a project of the regional government of Andalusia (Spain) in order to get a Web intelligent agent, being our interface agent a first step forward. In this Web interface prototype we can see six components. The first that we can see is the Header that is responsible for controlling the access to the system. The Geography Map visualizes the maps and the other four components correspond to different checkboxes to show data on Geography Map. Once detailed the scenario where the interface agent works, we detail the necessary behaviors to perform the interface adaptation.

3 COTS-BASED INTERFACE AGENT MODEL

In our system, the interface agent is responsible for observing the user interactions with the interface and implementing the necessary changes to adapt the interface to the user needs. In Figure 2, we can see the user interaction with the interface components performing the tasks assigned. Our interface agent observes the actions sequences that executes the user on the components (Section 3.1) and transforms them in an observer model (Section 3.2). This model is the input of the adaptation module, which makes the appropriate adjustments depending on the user actions. The result is a set of adaptation actions (Section 3.3) that the interface agent executes on the interface.

![Figure 2: Interface Agent.](image)

We will explain with more detail the UI adaptation at runtime in the Figure 3. The interface agent observes the user interaction with the Observe Interaction. In the next step, Manage Interaction compiles all the user actions creating sequences of actions. These two steps are performed continuously, storing these action sequences in the Functional History (FH). The Identify Plans module classifies the user actions stored in FH, deleting the actions which do not represent changes in the model. Create Observer Model is responsible for transforming the actions sequences performed by the user in an observer model for the adaptation module. When the adaptation module ends,
Get Adaptation Actions stores the adaptation actions and the Manage Adaptation Actions completes the required fields in order to execute the actions in the interface. The Apply Adaptation Actions executes the actions and adapts the interface components to the user actions. In the following sections we will explain each step with more detail.

3.1 User Interaction

Our interface agent operates as show in Figure 4, collects each user interaction. We define a input as a tuple \( \langle e, obj, comp \rangle \) where \( e \) is the observed event, \( obj \) is the object on which was made the event and \( comp \) is the component. These inputs are analyzed in observation module and are sent to intention detection module. This module creates the user intention based on the input information considering the component and the event made. An intention is represented by a tuple \( \langle e, obj, comp, int \rangle \), where \( e, obj, comp \) are elements inherited to observation module and \( int \) is the intention may be created of inputs. Each intention is sent to create action module which determines the action based on the user intention and stores this action in the Functional History (FH). This module defines an action as a pair \( \langle n, comp \rangle \) where \( n \) is the name of the action and \( comp \) is the component which performs the action. The process returns cyclically to observation module completing a cycle.

A possible example would be to execute the necessary actions to display the information in an area of the Geography Map component (sequence (1)).

\[
\text{input} = \langle \text{Click, Layer, Map} \rangle \\
\text{input} = \langle \text{Move, Layer, Map} \rangle \\
\text{input} = \langle \text{DblClick, Layer, Map} \rangle \\
\text{intention} = \langle \text{Click, Layer, Map, Click} \rangle \\
\text{intention} = \langle \text{Move, Layer, Map, Select} \rangle \\
\text{intention} = \langle \text{DblClick, Layer, Map, Action} \rangle \\
\text{action} = \langle \text{ClickLayer, Map} \rangle \\
\text{action} = \langle \text{SelectLayer, Map} \rangle \\
\text{action} = \langle \text{ShowInfo, Map} \rangle
\]
### Table 1: Partition Algorithm Pseudocode.

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For all ( a ) in ( FH ) do</td>
</tr>
<tr>
<td>2</td>
<td>If ( a ) is a component contained in ( COTS.GUI ) then</td>
</tr>
<tr>
<td>3</td>
<td>For all ( P_i ) in ( COTS.PL ) do</td>
</tr>
<tr>
<td>4</td>
<td>If ( a ) is ( P_j ) then</td>
</tr>
<tr>
<td>5</td>
<td>Add ( a ) to ( P_j )</td>
</tr>
<tr>
<td>6</td>
<td>If ( P_j = P_i ) then</td>
</tr>
<tr>
<td>7</td>
<td>Return ( P_i )</td>
</tr>
<tr>
<td>8</td>
<td>Else</td>
</tr>
<tr>
<td>9</td>
<td>Remove ( a ) from ( P_j )</td>
</tr>
<tr>
<td>10</td>
<td>PartitionActionIn(( FH, P_j, P_i ))</td>
</tr>
<tr>
<td>11</td>
<td>If ( P_j = P_i ) then</td>
</tr>
<tr>
<td>12</td>
<td>Return ( P_i )</td>
</tr>
<tr>
<td>13</td>
<td>Else</td>
</tr>
<tr>
<td>14</td>
<td>Return ( P_i = \emptyset )</td>
</tr>
<tr>
<td>15</td>
<td>End if</td>
</tr>
<tr>
<td>16</td>
<td>End if</td>
</tr>
<tr>
<td>17</td>
<td>End for</td>
</tr>
<tr>
<td>18</td>
<td>End if</td>
</tr>
<tr>
<td>19</td>
<td>End if</td>
</tr>
<tr>
<td>20</td>
<td>End for</td>
</tr>
</tbody>
</table>

### Process PartitionActionIn

1. For all \( a \) in \( FH \) do
2. For all \( a \) in \( P_i \) do
3. If \( a_i = a \) then
4. Add \( a_i \) to \( P_i \)
5. Else
6. Return \( P_i = \emptyset \)
7. End if
8. End for
9. If \( P_i = P_j \) then
10. Return \( P_i \)
11. Else
12. Return \( P_i = \emptyset \)
13. End if
14. End for

If we examine our algorithm, each action \( a \) is obtained from the \( FH \) queue and is never considered any more. Then it checks the component where the action has been executed in the set of components \( (COTS.GUI) \) from GUI. This component \( (COTS) \) returns all associated plans \( (COTS.PL) \) and it checks if it corresponds to the first actions sequence that forms a plan of actions \( (P_i) \). If the action \( a \) corresponds to the first action of a plan, this action is added to a temporal plan \( P_i \). If the plan has an action of sequence, the algorithm finishes returning the executed plan. If the plan has two actions or more, the algorithm uses the Partition-ActionIn algorithm. This algorithm tries to complete the actions sequence included in the selected plan. That is, the algorithm goes straight on checking the actions obtained from \( FH \) until completing the plan. If at any time, the action obtained from \( FH \) does not match with the action to be executed, then the algorithm finishes returning an empty plan. If the actions sequence obtained from \( FH \) matches perfectly with the sequence of the plan, the algorithm returns the executed plan.

For example, we display the information in a sector of Geography Map component. We define an action \( a_{ij} \) where \( i \) is the number associated with the component plan and \( j \) is the sequence of actions of each plan. We suppose that in \( FH \) stores the following collection of actions \( \{a_{11}, a_{21}, a_{12}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}\} \), where \( \{a_{31}, a_{32}, a_{33}\} \) are the actions of example:

\[
\begin{align*}
    a_{31} &= \text{ClickLayer, Map} \\
    a_{32} &= \text{SelectLayer, Map} \\
    a_{33} &= \text{ShowInfo, Map}
\end{align*}
\]  

(4)

We suppose that all actions have been carried out on a component which only has three associated plans \((PL_1, PL_2, PL_3)\) where \( PL_3 \) is the corresponding to the previous example. When the algorithm is performed, the action \( \{a_{11}\} \) is associated to \( PL_1 \). The sequence \( \{a_{21}, a_{21}\} \) does not match the \( PL_2 \) since \( a_{21} \) is not an action of the selected component. \( \{a_{22}, a_{23}\} \) does not correspond with the beginning of a plan component. \( \{a_{31}, a_{32}, a_{33}\} \) corresponds to \( PL_3 \). \( PL_1, PL_3 \) are plans component. \( a_{22} \) is an action that does not belong to any plans. Using the partition algorithm, we get an effective segmentation. The sequence \( \{a_{22}\} \) would be checked if it belongs to some other plans, being discarded at the end. Once the interface agent has identified a plan, it generates a model that is sent to the adaptation engine. This process is described below.

### 3.2 Observer Model

In order to develop a right adaptation, it is also necessary to get other elements of the UI proposed by (Criado et al., 2012) and observing system requirements that are described in (Troya et al., 2010). All these variables make up our observer meta-models showed in Figure 5, which shows the meta-model of the relationship between observed actions and monitored context variables. This meta-model is required by the adaptation module. This defines three types of observers: ComponentObserver, ObserverObserver and ContextObserver. The first type monitors variables of the components on the system. The second type provides information about other observers, and the third is responsible for observing the context variables. We add a new ComponentObserver completing the observation of the user interaction (inputs), which is represented by PlanObserver. The observation of these types of observer gives a complete view of everything that happens in the graphic user interface.

This article will not detail the adaptation operation. For a more detailed we suggest the reading of (Criado et al., 2012) and (Rodriguez-Gracia et al., 2012).
2012b). We will send created observer model to adaptation module and will return a set of adaptation actions that the interface agent interprets and executes in the Web interface. In the next section we will detail these adaptation actions.

3.3 Adaptation Actions

When the adaptation module finishes, the interface agent receives the adaptation actions that uses to carry out the adaptation of the UI. The Figure 6 shows the adaptation actions meta-model.

![Figure 6: Adaptation Actions Meta-model.](image)

As we can see there are four types of adaptation actions, Insert, Delete, Replace and Modify. The adaptation action element has as main attribute the component ID which executes the action. Each of one of these adaptation actions are performed under a certain order. It also checks that all adaptation actions complete each adaptation action with specific attributes of Web environment. Each of one of the adaptation actions are converted by the interface agent to Web specific language in order to be executed directly on the interface.

In the table 2 we can see an adaptation action in XML that modifies the Geography Map component. Each component is identified by its ID. This operation changes the frame size of the component executing the new conditions of the user action. It is executed directly by the interface agent at runtime, modifying the visual appearance of the UI without the intervention of the user.

<table>
<thead>
<tr>
<th>Action</th>
<th>ModifyAction</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;widget id=&quot;map01258&quot; operation=&quot;modify&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;closePosition value=&quot;relative&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;closeTop value=&quot;-22px&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;closeLeft value=&quot;208px&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;height value=&quot;100px&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;iframelHeight value=&quot;100px&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;iframelWidth value=&quot;100px&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;width value=&quot;230px&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/widget&gt;</td>
<td></td>
</tr>
</tbody>
</table>

4 RELATED WORK

Some approaches take into account variables associated with the actions as (Maes, 1994) (Brown et al., 1998). The main problem with this approach is that it ignores the user tasks and that the values of variables are pre-fixed. There are studies that attempt to predict the user intentions with non-probabilistic methods using logical methods as (Kautz et al., 1991) (Goulti-aeva, 2006). This approach can not decide what measure the evidence supports the hypothesis particular user intention. Other works like (Mott et al., 2006) (Charniak and Goldman, 1991) use n-gram models and Bayesian networks to probabilistically improve the prediction of user intentions.

Another approach to interface agents to consider is to control the design and creation of UI. SurfAgent (Somlo and Howe, 2003) is an information agent that builds a user profile by using examples provided by the user documentation. (Li, 2009) is a method of constructing UI based on a model agent type Model-View-Controller showing the pattern design and development method of UI based on agents an information retrieval system. (da Silva et al., 2000) describes the solutions offered by a mobile agent system (AgentSpace) showing two complementary ways to create UI with the mobile agent. (Arias and Daltrini, 1996) shows a framework for the conceptual and detailed design of the UI that helps people involved in the task of designing the UI, helping to create a custom interface. All these papers are focused on the design and control of interfaces, but do not take into account the user interaction.

5 CONCLUSIONS AND FUTURE WORK

Our paper presents an interface agent that observes to
the user, collecting all events and actions performed on the widget-type interface component, and promotes changes in the interface model, making that the interface evolves and adapts to the characteristics and needs of the user work. We have shown the process to obtain the user actions in form of plans and how the interface agent creates an observer model. This observer model is sent to the adaptation engine that generates a set of adaptation actions to change the interface. Our interface agent completes and executes these adaptation actions transparently, making that the interface is adapted to the actions performed by the user. Our interface agent combines capabilities and other features of interface agents, allowing to obtain the user intention based on the interaction among components, and at the same time is capable of changing the visual appearance of the interface as a result of the user interaction.

Finally, we are working to provide the system with social features through user groups, adding to the system abilities to work with multiple users in a cooperative manner, showing different interfaces to different users who are doing work cooperatively.

ACKNOWLEDGEMENTS

This work has been supported by the project JUNTA ANDALUCIA (proyecto de excelencia) TIC-6114, and the EU (FEDER) and the Spanish Ministry MINECO under grant of the projects TIN2010-15588 and TRA2009-0309 and Ingenieros Alborada IDi.

REFERENCES


