Providing End-User Facilities to Simplify Ontology-Driven Web Application Authoring

Abstract

Generally speaking, emerging web-based technologies are mostly intended for professional developers. They pay poor attention to users who have no programming abilities but need to customize software applications. At some point, such needs force end-users to act as designers in various aspects of software authoring and development. Every day, more new computing-related professionals attempt to create and modify existing applications in order to customize web-based artifacts that will help them carry out their daily tasks. In general they are domain experts rather than skilled software designers, and new authoring mechanisms are needed in order that they can accomplish their tasks properly. The work we present is an effort to supply end-users with easy mechanisms for authoring web-based applications. To complement this effort, we present a user study showing that it is possible to carry out a trade-off between expressiveness and ease of use in order to provide end-users with authoring facilities.

1. Introduction

Human-computer interaction research continues to garner a great deal of interest from both academia and industry. Computer applications today are sophisticated in function, with many users. This provides strong motivation for improving user interface designs as well as for making explicit the ground-rules for how people should interact with computers in order to accomplish their daily tasks easily. In fact, the current trend in computing shows a shift from machine-centered automation to user-centered services and tools. Therefore the traditional approach to computing must change from an emphasis on what computers can do to what people can do with computers (Shneiderman, 2003).

Modern computing advocates the idea of people customizing, configuring and also creating software artifacts in order to accomplish their daily tasks. However, professionals such as engineers, scientists and freelances may have concrete domain skills but lack programming abilities (Macías and Castells, 2004). In this regard, further support is needed in order to provide such non-programmer professionals with easy-to-use mechanisms for customizing software artifacts, avoiding the need for them to learn programming languages and specifications that are usually deemed to be irrelevant for their daily work activities. Programming languages must be flexible enough to deal with a wide range of problems; but with flexibility comes complexity, and the result is a learning curve that most users cannot be expected to afford (CACM Special Issue on End-User Development, 2004). Several researchers have sought to reduce the learning burden by creating design environments that do not require users to program per se; instead, they design by instructing the machine to learn from examples (Lieberman, 2001) or by interacting with graphical micro-worlds representing real domains.

It is estimated that over the next few years we will be moving from easy-to-use to easy-to-develop interactive software systems. A study reports that today in the U.S. alone, there are 55 million end-user developers compared with 2.75 million professional software developers (Boehm et al., 1995). This suggests the idea of considering new design strategies that provide the end-user with a different role, one of self-directed

designer rather than simple computer operator. This idea has motivated new interaction paradigms to appear.

Probably the most important approach is End-User Development (EUD) (EUD-NET; Lieberman et al., 2006). This paradigm is focused on a user-centered approach. EUD can be thought of as a set of activities and techniques that allow people (including non-professional developers) to create or modify a software artifact (Klann, 2003). EUD is targeted at meeting the needs previously commented upon, enabling end-users to create their own software artifacts with minimum effort. Do-it-yourself computing is probably one way of regarding this flourishing field (CACM Special Issue on End-User Development, 2004).

Certainly, the World Wide Web can be considered as first and foremost a software platform where end-users carry out authoring tasks every day. Large amounts of web content and functionality are accessed today through web applications. Many companies have created their own web pages and provided users with new capabilities for customizing and configuring the way they work and acquire information. This is mostly due to the fact that the WWW has spread as a relevant information and distribution medium in the last ten years, allowing people to access their saving accounts, purchase goods, learn new things and, in general, interact with the world of shared information every day.

Ways of authoring static web pages have been provided by existing commercial tools for many years. Such tools enable the user to make changes to web pages and upload them to web servers while avoiding having to deal with HTML-based languages. However, to provide the maximal functionality, most of today's web applications are dynamic rather than static. It is estimated that 80% of web pages are dynamically generated by applications and services stored on web servers (Sahuguet and Azavant, 2000). Authoring a dynamic web page is a rather complex task, since it requires programming skill. At present, most web pages built by end-users simply present information; the creation of interactive web sites or of web applications such as online forms, surveys and interactive programs still requires considerable skill in programming and web technology.

Preliminary studies indicate that users' web development activities are limited not because of a lack of interest but rather because of the difficulties inherent in interactive web development (Rode et al., 2006). There are commercial applications, including languages and frameworks such as XSL and JSP/ASP, that can greatly simplify the development and maintenance of dynamic web pages. However, such environments still require advanced technical knowledge, which domain experts, graphic designers or even average programmers may lack. Admittedly, these commercial tools help one to manage projects and they provide code-browsing and debugging facilities, but one still has to deal with the code finally. Additionally, users might want to customize only a particular part of a web application, having no need to deal with relatively complex programmer-targeted development environments when carrying out simpler modifications. An interesting study by Rode and Rosson (2003) revealed that, although much progress has been made by commercial web development tools, most of the end-user tools that they reviewed did not lack functionality but rather ease of use. Rode and Rosson explored many different paths, including extensions to a popular web development tool (Macromedia Dreamweaver), that might offer web application features more suitable to end-users. Although tools such as Dreamweaver and FrontPage have substantial facilities for extending application programming interfaces (APIs), Rode and Rosson found the inflexibility in controlling the users' workflow to be the main hindrance to

adopting these approaches. Currently, none of the commercial tools that they reviewed would work without major problems for the informal web developer. Ideally, following the concept of the 'gentle slope' (MacLean et al., 1990), the skills required to implement advanced features should be proportionate to the complexity of the desired functionality.

In general, no definitive solution has yet been proposed that provides end-users with easy mechanisms for authoring web-based applications dynamically generated by databases, web-based services and ontology-based servers. This problem is motivated by the difficulty of providing what-you-see-is-what-you-get (WYSIWYG) tools for the development of dynamic pages, since it is hard to describe procedural behavior visually. That is an inherent problem that is a particular concern with EUD, since authoring dynamic web pages can be thought of as creating and editing web-based software artifacts.

In this paper we present an approach aimed at enabling end-users to author dynamically generated web-based pages. The approach consists of two tools intended to minimize the effort in authoring dynamic web pages:

- DESK (Macías and Castells, 2003a, 2006) is an interactive authoring tool that allows the customization of dynamic page-generation procedures but does not require authors to have a priori tool-specific skills.
- PEGASUS (Castells and Macías 2001, 2002) generates HTML pages from a structured domain model and an abstract presentation model.

Our approach consists of combining Programming by Example (PBE) techniques (Lieberman, 2001; Cypher, 1993) with a bespoke ontology-based representation of knowledge. DESK acts as a client-side complement of the PEGASUS dynamic web-page generation system. This solution attempts to smooth the gentle slope of complexity in software usage (Macías and Castells, 2004), decreasing general expressiveness by means of a WYSIWYG environment, in favor of increasing the ease of use.

DESK faces the challenge of supporting the customization of page-generation procedures in an editing environment that looks like an HTML editor from the author's point of view. The PEGASUS presentation model specifies which pieces of knowledge should be presented and how a certain unit of information from the domain model is presented to the user. Instead of using PEGASUS's modeling language, authorized users can modify the internal presentation model by editing, in DESK, the HTML pages that PEGASUS generates. DESK follows the PBE approach of inferring changes that affect every class of knowledge from the user's actions on the presentation of a specific unit. DESK widens the spectrum of authors who can participate in an otherwise abstract and complex model-based environment such as PEGASUS. Inversely, our work shows that PBE techniques can benefit from a knowledge-based approach, which provides models of the user interface and explicit domain semantics for the PBE component to reason about. Consequently, our system's main goal is to help users to carry out the authoring task easily. Further, novice web users can benefit from using our system, since no programming languages are required and help is provided throughout the interaction in order to achieve modifications to web pages with minimum effort.

Our system was originally created in 2001 and has been considerably improved since. Initially, the system was intended to deal with adaptive hypermedia courses, preserving the original adaptive nature of the project so far. Regarding that, we have created different adaptive courses, such as one on graph theory, and also other presentations on

painters, an electronic shop and so forth. Our work was implemented using different technologies. We have mainly used Java as a coding language, but also some JavaScript, XML and JSP. We used parsing technologies such as JDOM for dealing with the XML code, and Apache Tomcat to implement the server part.

The rest of this paper is organized as follows. Section 2 describes the PEGASUS mechanism as well as the ontology-based underlying models used in dynamic page generation. Section 3 presents DESK as well as the inference mechanisms used in authoring dynamic web pages generated by PEGASUS. Section 4 presents empirical results about DESK by means of an experiment carried out with users. Next, Section 5 presents the results and the discussion. Section 6 outlines related work and, finally, Section 7 discusses conclusions.

2. Ontology-driven specification in web interfaces

A few years ago, the semantic web paradigm proposed new challenges for knowledge representation, web structure and automation. New XML-based languages were created to deal with semantic relationships and contents that are dynamically generated and represented in a web browser. We firmly believe in ontologies as a way to model different aspects of a user interface and to provide conceptual models by which complex relationships can be defined. Such conceptualizations can be used to codify high-level semantic paths for automatic web-based interface generation, further characterization and reverse-engineering purposes (Macías et al., 2006, Macías 2003). Our research experience is in using ontologies to specify knowledge for building data models (domain models) used together with application or presentation models. This has informed our approach towards specifying complex knowledge focused on the interface's domain and presentation models, as well as working with XML-based languages that better fulfill our assumptions about knowledge distribution and sharing. More precisely, we work on combining ontologies with Model-Based User Interface (MBUI) techniques (Paternò, 2001, Puerta and Eisenstein, 1999), which emerged as a solution claiming to overcome several difficulties in automating the process of generating interfaces (e.g. redundancy, lack of encapsulation and reusability). The implicit idea behind MBUI is to split up the conceptual level of a user interface, which leads consequently to the explicit specification of different aspects of the interface itself, such as domain knowledge, presentation, dialog and behavior.

PEGASUS (Presentation modelling Environment for the Generation of ontology-Aware context-Sensitive web User interfaceS) is a domain-independent system that helps to create a dynamic front-end for ontology-driven knowledge-based applications on the web (Castells and Macías, 2002). PEGASUS supports the definition of made-to-measure ontologies for the description of domain knowledge (see Figure 1). This approach is based on MBUI mechanisms that ensure domain independence by separating concept and presentation, so that the system generates web pages on the fly by selecting domain objects and assembling them into HTML documents in response to a user's requests for concrete knowledge units.

(Figure 1)

2.1. The domain model

The domain model in PEGASUS comprises a semantic network of ontology class instances and relations. The domain ontology consists of a set of classes that best fit a specific application domain or that reflect the specific view of a particular author on the

domain. In our approach, ontologies can be defined with a high degree of freedom, with very generic classes such as Catalog or Product, or more specific ones such as E-Mail Clients and Multimedia Tools. This knowledge is captured by defining attributes for classes, and relations between classes (Castells and Macías, 2001).

Figure 2 shows an example of domain ontology containing classes such as those mentioned above. This ontology has been created using one of our authoring tools, PERSEUS, by which the domain models for different PEGASUS applications can be defined easily. PERSEUS (Presentation ontology buildER for cuStom lEarning sUpport Systems) (Macías and Castells, 2001) is an interactive, form-driven tool that was originally used for creating adaptive hypermedia e-learning systems with PEGASUS (Castells and Macías, 2001). The main goal of this tool was the automatic generation of the XML files containing the domain information that will be processed by PEGASUS. PERSEUS allows for custom domain-model designs, where the designer can specify the hierarchical structure of the ontology by creating different classes and relating them to one another by defining dependencies in terms of parent classes and semantic relations. In Figure 2, `Product` corresponds to the current class that is being edited. This class comes from a more abstract class called `DomainObject` and inherits attributes such as `title` and `url`, as well as relations such as `Information`. More specific attributes for that class have been defined, such as `picture` (of a product), `date`, `size` and so on. Furthermore, a new relation called `BelongsTo` has been defined. Relations are one of the most important features of the domain ontology. In our system, relations can be created for a concrete class by defining the type of the class that is related to the current one, activating the `multivalued` property that indicates whether the relation is one-to-many or not, and completing the `title` and the relation's attributes when applicable. For instance, `Catalog` has a one-to-many relation with objects of class `HigherCategory` (i.e. `E-Mail`, `Internet` and so on), whereas `Product` has a one-to-many relation with objects of class `LowerCategory` (`E-Mail-Clients`, `E-Mail-Parsers` and so on).

(Figure 2)

For example, assuming the domain ontology defined in Figure 2 for a software download site such as Tucows, including classes `Product`, `Category`, `HigherCategory`, `LowerCategory`, and `Catalog`, the following instances could be defined:

```
<HigherCategory id="Internet">
  <subCategories>
    <HigherCategory ref="Connectivity"/>
    <HigherCategory ref="Communications"/>
    <HigherCategory ref="E-Mail"/>
    ...
  </subCategories>
</HigherCategory>

<HigherCategory id="E-Mail">
  <subCategories>
    <LowerCategory ref="E-Mail-Clients"/>
    <LowerCategory ref="E-Mail-Parsers"/>
    ...
  </subCategories>
</HigherCategory>
```

```
<LowerCategory id="E-Mail-Clients">
  <products>
    <Product ref="AgileMail_2.0"/>
    <Product ref="AllegroMail_2.0.1"/>
    ...
  <products>
</LowerCategory>

<Product id="AllegroMail_2.0.1"
         license="Shareware" price="39.95">
  <information> <AtomicFragment>
      With AllegroMail, you can set up...
  </AtomicFragment> </information>
</Product>
```

This code corresponds to the instances created by using PERSEUS in Figure 3. PERSEUS can generate the domain model of a PEGASUS presentation automatically. To be precise, the code above corresponds to PERSEUS's output once the designer has generated the corresponding domain ontology. XML attributes such as `license` and `price` correspond to properties of a knowledge unit (of class `Product`), whereas elements such as `subCategories` and `products` are relations with other units (the `ref` attribute corresponds to the unit identifiers). As in Figure 2, Figure 3 shows information about the domain instances listed above. In particular, it depicts the information about `AllegroMail_2.0.1`, which is the instance that is being currently edited. This instance belongs to class `Product`, with attributes that have been instanced with concrete values such as the URL, the picture and the date of the product. Furthermore, relations such as `BelongsTo` have been filled in, creating one- to-many relations between `Product` and `E-Mail-Clients`.

(Figure 3)

For historical reasons, we are at present using our own XML extensions to represent the domain model, but we are planning to move to some of the currently available ontology definition standards such as RDF or OWL (Dean et al., 2002), with minor modifications to our system.

2.2. The presentation model

In contrast to other knowledge-based systems that generate pages automatically, such as Adaptive Hypermedia systems (Brusilovsky et al., 1998; Murray, 1998), PEGASUS provides extensive control over presentation design by using an explicit presentation model, separate from questions of content. The separation of content and presentation is achieved by defining a *presentation template* for each class of the ontology. Templates define those parts (attributes and relations) of a knowledge item that must be included in its presentation and in what order they are to appear, as well as their visual appearance and layout (see Figure 4).

(Figure 4)

In addition to domain objects, style can be managed in our approach by inserting standard JavaServer Pages (JSP) tags through which the designer can freely customize the layout of the page as well as the graphical properties of text and widgets, such as color, background, size, justification, font type and so on. This way, it is possible to write specifications such as the following:

```
<h2> <u> <%= Product.title %> </u> </h2>
```

This means the `title` attribute of a product will be made visual by being underlined and in header2 style. This explicit separation enables graphical aspects and domain contents to be handled more naturally, splitting up design responsibilities depending on the designer's task and/or background. Simpler templates can be elaborated by graphical designers, who need focus only on the presentation's graphical aspects. Designers only have to take care to insert references to domain concepts (such as `Product`, `CategoryofProduct` and so on) into the presentation template. Therefore, content providers need only focus on the structure of the domain ontology in order to create the contents for such references. Finally, the system dynamically generates the objects instanced, using the template created previously.

A template is defined by using an extension of HTML based on JSP that allows the insertion of control statements (between `<%` and `%>`) and Java expressions (between `<%=` and `%>`) in the HTML code. For instance, a template for class `HigherCategory` could be as follows:

```
<% if (availableSpace > 5) { %>                                    1
   <widget type="Table" columns="3"                                2
           dataflow="wrap">                                        3
     <list> <%= subcategories %> </list>                           4
   </widget>                                                       5
<% } else { %>                                                     6
   <table>                                                         7
     <tr><td> <%= id %> </td></tr>                                 8
     <tr><td> <%= subcategories %> </td></tr>                      9
   </table>                                                        10
<% } %>                                                            11
```

The template above specifies that, when there is enough space available (estimated on a scale from 0 to 10), a table should be generated in which a subcategory is presented in each cell, left to right and top to bottom (lines 2 to 5). Otherwise, a table of two rows and a single column is generated (lines 7 to 10) where the category `id` (line 8) and the list of subcategories (line 9) are displayed. In general, the amount of visualization space is estimated by means of user identification, Javascript code and HTTP protocol headers. We obtain updated information about user and platform which takes part of the user model itself. Internally, we have programmed a method that estimates (based on a fixed scale from 0 to 10 that depends on the platform used) the *information capacity* in onscreen of the current platform used (PDA, Desktop, etc.) for each user session. This way, the static attribute `availableSpace` is updated and used to evaluate and insert user and platform conditions into the JSP template.

The expression `<%= subcategories %>` is a reference to the multi-valued relation `subcategories` of the `HigherCategory` being displayed. The relation points to a list of objects of type `Category`, which PEGASUS presents using the appropriate template recursively. The `widget` XML tag is a JSP custom tag used to provide a standard set of HTML widgets such as tables, input types (buttons, combo boxes, etc.) and selection lists. Each widget type has specific mechanisms for displaying domain-model data structures, using different strategies to map complex relations between domain objects to display structures.

The resulting page for the Internet category can be seen in Figure 5, where the outer table results from lines 2 to 5 of the template, and the inner tables correspond to lines 7 to 10 applied to subcategories of Internet software. For the sake of brevity, a few details such as cell background colors and the tabbed bar have been omitted from the template code. However, readers can refer to references (Macías and Castells, 2003, 2001) to find out more detailed explanations about templates.

Besides including templates, the PEGASUS presentation model enables the construction of presentation rules such as:

```
<Rule>
  <test condition="availableSpace <= 1 "/>
  <presentation>  <%= this.asLink() %>   </presentation>
</Rule>
```

(Figure 5)

In Figure 5, this rule controls the presentation of third-level subcategories, such as "E-mail Clients", as a link.

Adaptivity is carried through by inserting conditions into the presentation model's templates, into presentation rules, and into relations between domain objects. These conditions can test properties of the user model (overlay model and user profile), properties of the data, characteristics of the platform, and any other aspect that can influence presentation, such as task requirements, user's goals, usage modes (e.g. exploration vs. selective search), etc.

At run-time, the user interacts with the application through a web browser. Interaction with an application built with PEGASUS consists of navigating through the semantic network of domain objects. Each time the user moves to an object, PEGASUS responds by generating an HTML page (see Figure 6). In doing so, the system:

i) resolves the user's request by determining the actual object to move to;

ii) locates the instance in the domain model;

iii) updates the domain and user models; and

iv) generates the HTML presentation, applying the pertinent rules and the template that corresponds to the object class.

In the generated pages, links do not point to other pages but refer, explicitly or descriptively, to other domain objects.

(Figure 6)

From the PEGASUS point of view, the unit of interaction with the user is the HTTP request. User-model updates are carried out by taking into account only the information extracted from the client's requests. Platform and user-interface characteristics are captured client-side through JavaScript code that the system inserts in the generated HTML pages, and the information is returned to the server as part of the HTTP request when the user clicks on links and buttons. This assumption greatly simplifies the system architecture and the integration with external tools and modules. On the other hand, it means that the system is not explicitly aware of user activity between two requests, and the presentation is not updated during that interval. A finer but far more complex and bandwidth-sensitive approach could be supported by generating Java user-interface components (applets) that interact with the user and communicate directly with the server to query and update the domain and user models.

All in all, PEGASUS's underlying models are flexible enough to represent interface information with a high degree of expressiveness. Designers incorporate such knowledge by writing it by hand with an ontology-based standard editing tool or by using one of our previous tools, such as PERSEUS. However, a designer wanting to customize or further change a presentation generated by PEGASUS would have to follow the reverse path from the generated web page to the underlying models, dealing with procedural information, the domain and presentation models of the applications, and figuring out correspondences and mappings from the domain ontology to the presentation objects. Obviously, this is a difficult challenge to face, since dealing with procedural, presentation and domain knowledge together is not an easy task even for advanced programmers. When procedural information needs to be considered, data-driven design approaches such as PERSEUS are insufficient. Any solution proposed should be able to provide mechanisms to support customization by the end-user, where ease of use should be the primary concern.

We conceived DESK as an end-user authoring tool for dealing easily with web customization. DESK provides automatic support for creating designs involving domain, presentation and procedural information under the same authoring environment. Therefore, the user does not need to get involved with the reverse path that the system follows automatically to carry out the required modifications.

3. Providing end-users with authoring support

Our approach focuses on the EUD paradigm to deploy ontology-based MBUI techniques that relieve users from having to deal with specification languages. To this end, it accepts a reduction in the expressiveness of the MBUI approach in order that users do not have to manipulate declarative specifications for the interface. For a successful trade-off between expressiveness and complexity, the system must provide a low-level abstract design environment, such as a WYSIWYG interface that provides end-users with a real representation of the interface. Such environments help users to easily manipulate the interface's objects without using complex specification languages, and provide a realistic depiction at every step of what the user is attempting to do. However, creating an application from scratch through a WYSIWYG environment is not easy, since a great deal of implicit information from the underlying application models is often required (Macías and Castells, 2003b).

With DESK, the user can modify the design of dynamic web documents by editing the page that PEGASUS generates, instead of by directly manipulating its modeling language. DESK identifies domain values, model fragments, and presentation constructs in the HTML code, from which it infers meaningful transformations. The user only knows about the web document and need not be aware of the underlying models and languages.

Figure 7 shows how DESK works. DESK has both client-side and server-side components. The client-side component looks like a conventional HTML web-based authoring tool, where the user navigates through dynamic web pages generated by PEGASUS (1) and edits those (2) in a WYSIWYG environment. The tool monitors the user's activity and generates a monitoring model containing user actions along with its context for characterizing each action conveniently. Then this information is sent to DESK's server-side component(3), which processes the monitoring model, infers changes (4), generates suitable feedback and sends it back to the user (5). Finally, DESK applies the inferred changes to the PEGASUS's underlying models (6). Affected

web pages will be dynamically regenerated and will appear suitably modified whenever the user navigates through them.

(Figure 7)

3.1. Characterization of user intent

The DESK authoring tool uses a set of heuristics consisting of advanced ontology-based searching algorithms for obtaining both syntactic and semantic information in order to infer user intent. Syntactic information is obtained by the client-side component by means of *low-level heuristics* (H_L), whereas the server-side component obtains semantic information by applying *high-level heuristics* (H_H). This distinction is because semantic information is only available at the server-side where underlying high-level models are stored, whereas the client-side component is mostly provided with syntactic information about the user's modification to HTML objects. However, both syntactic and semantic information are used together to provide further accuracy when addressing ambiguity and analyzing context, thus obtaining more precise and meaningful information about the user's intent. In general terms, DESK heuristics deploy available knowledge from PEGASUS's domain ontology in order to map the user's modifications to appropriate domain structures.

At the client-side, DESK records all basic user editing actions accomplished in the HTML code (insert text, change text style, etc.) and attempts to find out the syntactic context by applying low-level heuristics. In turn, contextual information and user actions are packed into *constructor primitives* to form the monitoring model (Figure 8).

(Figure 8)

Low-level heuristics determine the syntactic context for every user action (Macías and Castells, 2005). Syntactic context is useful to obtain local information about where the changes take place in the HTML code, thus providing further support for disambiguation. Later, this information will be used on the server-side.

Low-level heuristics are grouped into several modules.

- The context-location module finds out the nearest syntactic context for each modification. Candidate contexts include references to surrounding HTML objects, and text fragments that could be useful in order to identify mappings among HTML code and domain objects (see the example of enumerated code in Section 3.2)
- The special-structure location module identifies presentation structures (e.g. tables, selection lists, etc.) in which a modification occurs. This module knows about items, cells, rows and columns, as well as how data structures are related to different presentation widgets.
- The monitoring-model generation module generates a structured monitoring model containing information extracted from previous modules—that is to say, user actions and the surrounding context. This module transforms atomic syntactic actions into meaningful editing primitives, including contextual location and information about the HTML object's structures.

3.2. Semantic transformations

Once the monitoring model has been created, it is sent to the server for further processing. Figure 9 shows the back-end architecture of DESK. The client-side sends the monitoring model to DESK's server-side component, where inference takes place.

(Figure 9)

Server-side processing is mainly focused on inferring semantic information that will eventually be used to update PEGASUS's underlying models. To this end, high-level heuristics (Macías and Castells, 2005) have been defined. These determine semantic context by examining the application's domain model. The system handles this by processing the domain ontology in order to find out relationships between the syntactic changes and the domain objects.

High-level heuristics are also grouped into several modules.

- The context-location module finds the semantic context by processing the domain ontology. This is probably the most important module and is also the first to be invoked. It is targeted at identifying domain objects by both analyzing the content of the monitoring model and processing the domain ontology. More precisely, an algorithm executes a loop to find whether an element of the monitoring model matches an ontology object or whether it has instead to be identified by the context (analyzing other surrounding objects).
- The presentation-context module takes into account the information reported by the context-location module to create references to presentation objects included in the presentation model of PEGASUS. These references will then be used to identify changes that concern how the domain objects will be visualized. Since the user can make changes to domain and presentation objects separately, the system must identify correctly whether a change affects the presentation level (lexical changes such as style, position, color and so on) or the domain level (changes concerning domain objects and relationships).
- The disambiguation module is called whenever an ambiguous situation appears. This is when two or more references for the same user modification are found during context searching by previous modules. To solve this problem, the disambiguation module takes into account contextual information stored in the monitoring model by means of the low-level heuristics. Such contextual information will be analyzed to disambiguate references and decide which is the appropriate one to select. When the ambiguity cannot be solved, the system prompts the user for help.

One of the most important concerns of the high-level heuristics is to process the domain model in order to obtain meaningful information for characterizing user changes. Figure 10 shows an example of how such a process is carried out. Let us suppose that the user edits the title of an e-mail client called "Allegro Mail", adding the word "Client" at the end. The following information is created in the monitoring model to codify this modification:

```
<InsertText>
  <Text> Client </Text>                                          1
  <Context start="12" end="18" before="" after="Item 2”>         2
    <Text> Allegro Mail </Text>                                  3
  </Context>
</InsertText>
```

The code above shows how the system recognizes the insertion of the word `Client` (line 1) and also the context where the insertion takes place—that is, from position 12 up to position 18 (line 2) of the first line (`before = ""`; means that before that point there is nothing) and just before a given `Item 2` (`after = "Item 2"`), following the existing paragraph `Allegro Mail` (line 3). It is worth noting in Figure 10 that the line `Allegro Mail` was generated by a `<%=subcategories("vertical")%>` instruction in the presentation template (step 0). Such a command establishes different categories of email products (in this case) to be visualized vertically by a selection list (step 1). In order for the system to detect the proposed modification, it processes the monitoring-model code above and attempts to find out where "Allegro Mail" software appears by matching that string with the existing domain objects. Eventually, an occurrence is found, as the `title` attribute of object `EMC1` seems to contain such a string (step 2). The system starts to analyze the object affected and then realizes that it belongs to the category `eMail-Clients`. The system searches the domain model again to find where the object `EMC1` occurs, and discovers that it is included in the relation `BelongsTo` of the object `E-Mail` (step 3). Analyzing this object and searching the domain model once again, the system finally finds the class `Software`, where the relation `BelongsTo` appears (step 4). Since the system follows up every relation coming from the first occurrence, it is possible to determine the logical path for every modification. In this way, carrying out a bottom-up search and keeping the information found during the process the system can characterize the change in the domain model. In this particular case, the characterization carried out by the system can be summarized as: "The user has modified the title of a «lower category» e-mail client product that belongs to a «higher category» called e-mail, included in the software catalog of the electronic shop."

(Figure 10)

The changes our system can detect may also involve presentation styles in the JSP template. In order to detect those, the system first characterizes the object involved in the modification as explained above. It then searches the presentation model of the corresponding class in which the object appears. This is the task of the presentation-context module, which matches the characterized object with its representation in the presentation template, replacing, removing or adding the new style attributes.

For instance, let us suppose that the presentation template contains the code `<h1> <%= Product.title %> </h1>`, giving the product's title a heading style of `h1`. If the user decides to change the style to `h2`, the following line depicting such a modification will appear in the monitoring model:

```
<ChangeStyle old="h1" new = "h2">
  <Text> Allegro Mail Client </Text>
  <Context start="1" end="20" before="" after="Item 2"/>
</InsertText>
```

Once the context-location module has characterized the object `Product` and its attribute `title`, the presentation-context module searches the presentation template (class `Product`) for such a reference (`Product.title`) and replaces the existing attribute (`h1`) by the new one (`h2`), resulting in the following line in the presentation template: `<h2> <%= Product.title %> </h2>`.

This process can be generalized easily for every HTML structure (such as a table or selection list) and widget. Therefore, monitoring-model primitives can reflect changes

and additions in style and page layout detected anywhere in the presentation template. Additionally, ambiguities are also addressed through the disambiguation module. That is, if the same object reference appears twice or more in the same presentation template, contextual information is analyzed. The contextual information appears in every primitive generated in the monitoring model (see previous examples of code), reflecting the start and end positions and the objects appearing immediately before and after it. Thus the system can determine the right object to change, with minimum ambiguity.

The process of running high-level heuristics enriches the monitoring model with information resulting from the characterization explained above—that is, semantic knowledge such as concrete domain-object names, attributes and semantic relationships. Finally, a specialized module for managing changes processes the (enriched) monitoring model again in order to accomplish the changes to PEGASUS's underlying domain and presentation models, sending back in turn a detailed report and prompting the user for help if needed. In Figure 10, the attribute `title` of object `EMC1` is readily modified by such changes to the monitoring module.

Admittedly, this is only a simplified example of how the system works. The next section introduces a further example to illustrate how the system deals with far more complex presentation structures.

3.3. A worked example

At first sight, the DESK client looks like a standard tool for editing HTML pages and navigating through them. However, internally it manages a structured model of the user interaction (the monitoring model). As well as being used to record the user's changes to dynamic web pages, the knowledge codified in the monitoring model is also employed to help the user accomplish cumbersome tasks automatically. This process is carried out by analyzing the monitoring model's actions carefully and using a reactive assistant to act as a surrogate for the user when necessary.

Figure 11 shows the user interface of the DESK client, where the web page depicted in Figure 5 is being modified as follows: a) the text "Applications" is to be inserted beside the "Internet" literal, and b) a few items from an HTML table have been cut and pasted into both a combo box and a selection list. In general, items can be added or removed from presentation structures by using a pop-up window that is made visible by double-clicking on the widget.

(Figure 11)

It is worth noting in Figure 11 that the user is attempting to replace an existing table containing product categories by a combo box. Such a widget contains one of the higher categories and also a selection list for selecting the subcategories of one of the elements chosen from the combo box. DESK automatically detects the user's intent and suggests that the table should be replaced by the combo box and selection list. Figure 12 shows the result of such a process, once DESK has changed the presentation model on the server-side (the internal mechanisms of DESK's client- and server-sides are detailed in later sections). As we can see, the inserted text "Applications" appears twice in the presentation template. This is because the change concerns an attribute called `title` that is included in the domain ontology, and therefore is rendered on both the tabbed pane and the page title. This eventually results in the literal "Internet" being replaced by

"Internet Applications" at both those locations. Furthermore, the previous table has been replaced by a combo box and a selection list in the new version of the presentation.

(Figure 12)

It is worth emphasizing that the changes that have been accomplished are far from being merely syntactical. The presentation depicted in the previous example was dynamically generated, coming from the domain and presentation models stored on the server. Consequently, the changes made to widgets, as well as the automatic transformation shown, are automatically carried out by the system, which makes semantic assumptions about the presentation's widget structures and deals with mappings to domain objects. As we explain in the following sections, semantic information combined with the user's syntactic actions offers interesting possibilities for user assistance, enabling us to handle many transformations automatically.

3.3.1. Assisting the user

As we mentioned before, the monitoring model is one of the chief elements in DESK. Aimed at tracking user interaction for further semantic processing, the monitoring model is updated continually, reflecting every user action. Besides monitoring the user's actions, the monitoring model is also taken into account on the client-side in order to analyze syntactic actions and provide users with help when authoring a web page.

The DESK client features a mechanism intended to recognize the user's intent and provide appropriate help. This component knows about presentation structures, and allows for user actions to carry out automatic transformations. Rather than continuously checking for concrete user actions on presentation structures, which would be very inefficient, DESK includes a pre-activation agent (DESK-A) that checks against more general conditions and detects *iteration patterns*; see Macías and Castells (2005) for further detail. Only one agent is needed in order to check the monitoring model and detect different types of actions. This agent is activated when certain actions (e.g. copying elements from one widget into another) are detected. The agent looks for partial clues that alert the system to execute specific heuristics that trigger a more detailed analysis of actions and objects involved. The agent can be configured manually by defining its behavior in the form of rules. The agent's behavior is configured by a set of transformation hints such as the following:

```
<TransformationHint searchLength="100">
  <widget type="Table"
          changeTo="ComboBox,List" />
  <Condition action="Creation"                                    1
             object="ComboBox" />
  <Condition action="Creation"                                    2
             object="List" />
  <Condition action="PasteFragment"                               3
             from="Table" to="ComboBox"
             repeat="3" />
  <Condition action="PasteFragment"                               4
             from="Table" to="List"
             repeat="3" />
  <Condition fact="Relation" from="ComboBox"                       5
             to="List" />
```

```
</TransformationHint>
```

This hint activates a specific heuristic for transforming a table into a combo box and a selection list when the following conditions are satisfied:

1. a combo box has been created
2. a selection list has been created
3. *(and 4.*) domain fragments have been pasted from a table into a combo-box and a selection list (at least three times in each one)
5. there is an existing relation between the information pasted (in terms of domain knowledge) into each widget.

The `searchLength` attribute represents the number of actions in the monitoring model that the agent will consider at any one time. This parameter is useful for tracking back the user's actions related exclusively to the theme of one particular transformation (more than one transformation can be nested in the monitoring model). Once activated, the agent runs transformation heuristics to carry out more elaborate tests to work out how the transformation will be applied. This involves recognizing iteration patterns and coordinating data flow among presentation structures.

As already mentioned, the monitoring model comprises a sequence of instructions that reflect actions performed by the end-user. The following monitoring-model fragment shows two different primitives extracted from the previous example: the insertion of the string `Applications` and the transformation of a table into a combo box and a selection list:

```
<InsertText>                                                    1
  <Text> Applications </Text>
  <Context start="09" end="21"                                  2
           before="T01" after="TB01">                           3
    <Text> Internet </Text>
  </Context>
</InsertText>
<ChangeWidget>
  <From type="Table" id="T01"                                   4
        relation="subCategories"
        class="HigherCategory"
        objectID="Internet"/>
  <To   type="ComboBox" id="C01"                                5
        relation="subCategories"/>
        class="HigherCategory/>
  <To   type="List" id="L01"                                    6
        relation="subCategories"
        class ="LowerCategory" />
</ChangeWidget>
```

As for the text insertion (*1*), it is worth noting how DESK uncovered contextual information about the change (*2*), that is, where the information is located: starting at the ninth position besides the string "Internet", and ending at position twenty-one. Contextual semantic (*3*) reflects the fact that the insertion has been accomplished between the table `T01` and the tabbed bar `TB01` (DESK internally assigns an identifier to every widget when parsed). With regard to the transformation from a table to a combo box and a selection list, the code that the transformation heuristic generates

comprises a high-level instruction that includes domain semantics and relationships between the widgets involved. This way, the code above reflects how a table (*4*), identified by `T01` and generated by the relation `subCategories` of class `HigherCategory` and domain object `Internet`, is transformed into the combo box `C01` (*5*) and the selection list `L01` (*6*), keeping the same domain relationship (`subCategories`) and belonging to different domain classes (`HigherCategory` for the combo box and `LowerCategory` for the selection list).

In general, the DESK agent can deal with different types of change by configuring the agent's behavior in order to carry through meaningful transformations by using the monitoring model. Interested readers can refer to Macías (2003) and Macías and Castells (2005) in order to find further cases of transformations that have been omitted from this paper for the sake of brevity.

3.3.2. Deploying semantics in DESK

Once the monitoring model has been sent to the server-side DESK component, the system carefully analyzes its content, instruction by instruction. Continuing with the example of the modifications presented above, the first instruction corresponds to the text insertion (the string "Internet"). For each instruction, the DESK server uses high-level heuristics to search the domain model for information matching the domain objects, thereby adding (in the text-insertion example) the following semantic:

```
<Context class="Category" attribute ="id"
         objectID="Internet"/>
```

In this case, the server-side DESK component has found a correspondence with the domain model, and the system processes the domain-model object that has the identifier "Internet" (which is an instance of `Category`). As a result, the system adds the name of the class, the attribute and the object as semantic context, changing the content of the `id` attribute to "Internet Applications" in the domain ontology as well.

In the second example (the transformation of a table into a combo box and a selection list), the system notices that the change affects the presentation rather than the domain model, and no contextual information is added this time. Instead, the table is substituted by a combo box and a selection list in the presentation template for the class `HigherCategory`. After this modification, the new presentation template is as follows:

```
<% if (availableSpace > 5) { %>
   <widget type = "ComboBox">
     <items> <%= subCategories %> </items>
     <selectedItem> <%= selectedID %> </selectedItem>
   </widget>
   <widget type = "List">
     <items>
       <%= subCategories.item(SelectedID).subCategories %>
     </items>
   </widget>
<% } else { %>
   <table>
     <tr> <td> <%= id %>            </td> </tr>
     <tr> <td> <%= subcategories %> </td> </tr>
```

```
   </table>
<% } %>
```

The variable `SelectedID` represents an input parameter used for widgets that involve selection at runtime, such as the combo box. This parameter is internally generated and managed by the system, depending on the number of input values needed for each widget.

4. User study

The main goal of our work is to provide easy-to-use mechanisms for customizing dynamic web pages. To achieve this, a methodical approach for generating and authoring dynamic web pages has been proposed and fully implemented. While most commercial and other existing approaches are focused on dealing with static aspects or force the user to create code at some point, our approach protects the user from having to use a programming language when authoring dynamic web pages. To carry out such a challenging brief, the system features a reverse-engineering mechanism that helps end-users carry out modifications in a WYSIWYG environment. This enables the system to accomplish the changes by automatically modifying the underlying models on the server, thus providing the end-user with a new web page with minimal effort.

An empirical study has been carried out to evaluate and assess the quality of the approach presented here. This section reports on an experiment carried out with real users that has helped us evaluate DESK's authoring mechanisms. Next, in Section 5 we discuss the results and analyze DESK's functionality.

For the study, we recruited 12 participants from heterogeneous scientific backgrounds from our academic institution. The participants were given a 10-minute general introduction to the goal of the study. This experiment started with the premise that users were expected to have no or minimal skill in web programming, but to have a basic ability to handle web navigation. Post-study interviews revealed that only 5 participants had any web programming experience, which was limited to creating and modifying simple HTML pages manually. However, all of them had significant experience in WYSIWYG web authoring and navigation. It is worth mentioning that the authoring tool was not initially described to the participants, in order to observe how they dealt with DESK and, much more important, whether the authoring tools reminded them of others that they might have used.

In general, the main objectives of this study were:

a) Evaluate DESK's ease of use.

b) Observe whether users easily took control of the authoring mechanisms provided by DESK. It was important at this point to measure whether users felt familiar with the authoring tool, observing whether DESK reminded them of other similar tools they had used, such as commercial tools intended for static web authoring.

c) Observe user' expectations and frustrations in web page authoring with DESK.

d) Measure DESK's hit rate in inferring user intents from their actions monitored throughout the experiment.

In order to fulfill those objectives, a two-stage study was designed, consisting of two different tasks to be carried out by each participant:

1) Firstly, the users were asked to use DESK to author the dynamic presentation on scuba diving depicted in Figure 13. We wanted a very simple web page to be used for the test, in order that we could measure quick responses from both user and system rather than using a complex designs that would take far more time and effort. To this end, we used PERSEUS to create an ontology of a scuba-diving course, and then we created a simple template to generate the dynamic contents for the test. To carry out the changes, each user was provided with the same list of ten modifications to be accomplished. The modifications were not ordered explicitly, and were clearly specified in terms such as "Replace the text *X* by *Y*" and "Apply bold style to text *X*". At the beginning of the session, the user had 5 minutes to read the list of modifications carefully and ask any questions. The changes proposed can be summarized as follows:

    - Replacing different texts
    - Transforming a bullet list into a table
    - Adding a new element to the table created
    - Modifying text attributes (color, justification and so on)
    - Inserting new text
    - Removing existing text
    - Moving HTML objects.

    Each user started with the scuba-diving web page generated by PEGASUS. In addition, a printed paper copy containing the modifications was also provided. The task was then to modify the given page to obtain a final version with all the changes applied. The lack of a specified order in which the modifications should be made helped us measure the accuracy of inference, the expressiveness and the freedom of design provided by DESK, placing no restriction on the way users carried out the customizations from the initial design. Different users could carry out the modifications by following different steps and thereby expecting the system to respond in different ways. The main objective of this first part of the study was to obtain the maximum information about the operation of the system's inference mechanisms. The variety of modifications that were proposed helped us observe different aspects of the system's behavior and inferences made, such as:

    - How the system identified different domain objects by using contextual information extracted from user modifications.
    - How the user was assisted in the automatic transformation carried out. In this case, it was interesting to observe the system's behavior in transforming a bullet list into a table that was generated automatically by the system using the mechanisms explained in previous sections.
    - How the system can move, add, remove or insert new domain elements while keeping the contextual information and relating such modifications to the correct domain objects.
    - How the system can find attributes of domain objects related to style modifications in the presentation's templates.
    - How the system can control consistency with the new elements created by the automatically suggested transformations, identifying presentation

structures and enabling the user to add new components. The system was expected to discover where to add the new content in the domain ontology. In this study, the user added new content to the table automatically created by the system.

(Figure 13)

2) For the second part of the study, we used two different questionnaires to evaluate human reactions to the interaction with DESK, covering the topics of satisfaction, ease of use and user's expectations. Users were asked to fill out a questionnaire based on User Interface Satisfaction (Chin et al., 1988) and another based on Perceived Usefulness and Ease of Use (Davis, 1989). The questions in both questionnaires were selected and customized to mainly focus on DESK, avoiding asking participants to respond to unrelated questions. The main objective of this second part of the study was to obtain maximum information about users' perceptions when working with DESK.

5. Results and discussion

The first part of the experiment revealed interesting aspects about both DESK and the users' behavior. For each user intervention, we studied data extracted from internal system variables and DESK's monitoring model, with the aim of analyzing DESK's accuracy and behavior. Specifically, we studied the following parameters:

a) The time the user took to carry out all the changes
b) The number of primitives generated in the monitoring model
c) The inference hit rate (in inferring user intents).

Although the study generated a great deal of information, we summarize here the most important results obtained.

(Table 1)

Table 1 shows the numerical values obtained by the test. The first column shows the number of primitives generated during the user interaction and recorded on the monitoring model. This number differs from one user to another, as *max* and *min* values indicate. This is principally due to the fact that DESK offers enough expressiveness for a task to be accomplished in different ways, and so the number of primitives depends on the steps that each user followed to achieve the changes proposed. The average number of primitives generated was 200. In the second column, the hit rate shows 95% success in inferring users' intents. This implies that DESK achieved most changes successfully when carrying out the reverse-path analysis. Any errors were mainly due to ambiguities when inferring user intents (Macías, 2003) and they will be considered for future improvements. The final column shows the time that users spent in accomplishing the modifications. As we can see, participants spent an average of 5 minutes and 39 seconds on this part of the experiment. As the standard deviation shows, the spread of times is not very significant, since all participants were able to use standard web tools and therefore quickly became familiar with DESK's features. This corroborates one of our initial assumptions, since users perceived that DESK is similar to other static web tools but includes powerful mechanisms to modify dynamic web pages automatically.

The evaluation of the questionnaires also revealed interesting conclusions likely to be considered in the second part of the study. One of the parameters measured was the predictability of the authoring tool. This value is reflected in Table 2. The predictability

is a value ranged between 0 (min) and 5 (max) that users perceived when observing the final design inferred by the system. This variable can be considered as a way to estimate both frustration and expectation. A small value reflects the fact that the final design inferred by DESK did not agree with the user's intent, whereas a large value reflects the opposite. In most cases, expectation can be considered proportional to frustration. If the user's expectation is high and the system does not respond as desired, frustration will be also high. Table 2 indicates a good level of predictability for DESK, meaning that the final design inferred by DESK matched what users wanted and so in most cases they ended up with a low rate of frustration. We can also conclude that on average the authoring tool inferred the changes to the dynamic presentation that the user expected.

(Table 2)

DESK's ease of use is probably one of the foremost points to consider. Results obtained from the questionnaires greatly encourage a view of DESK as an easy-to-use authoring tool that can reduce drastically the 'gentle slope' of complexity. Figure 14 displays users' opinions about DESK's ease of use, showing that two users agreed that DESK is easy to use, and the rest (i.e. ten users) that DESK is very easy to use. The range of possible answers was "Very Easy", "Easy", "Normal", "Difficult", and "Very Difficult".

(Figure 14)

Another relevant concern is how users perceive DESK as an authoring tool for helping to solve their daily tasks. Figure 15 shows the participants' opinions about DESK's usefulness, where one user agreed it was very high, eight - high, two - normal, and only one user agreed it was low. The range of the possible answers was "Very High", "High", "Normal", "Low", and "Very Low".

(**Figure 15)**

The results obtained support the initial hypothesis. Most users thought of DESK as an easy-to-use authoring tool, very similar in some ways to other static authoring tools they may have used, but with an extra and powerful capability of authoring dynamically generated web pages. In this experiment, open questions also revealed that most users considered DESK to be a useful tool that can be applied to daily tasks such as authoring personal agendas and CVs, dealing with database-generated pages, managing dynamic on-line courses and teaching information, managing collaborative documents, authoring student forums and laboratory web pages, and so forth. Bearing in mind such opinions, we affirm that there is an obvious and increasing need to provide end-users with easy mechanisms for dealing with dynamically generated web contents in real time.

6. Related work

Our research aims to provide a set of PBE techniques for authoring domain-independent web-based user interfaces and dealing with high-level user tasks, and different domains have been considered in order to evaluate the tool. From this point of view, DESK is comparable to other approaches such as *Predictive Interfaces* (Darragh and Witten, 1991) and *Learning Information Agents* (Bauer et al., 2000), where the system observes and monitors the user's interaction with the software environment. These approaches help the user by predicting and suggesting some commands to carry out tasks automatically.

Monitoring user actions is a common practice to provide intelligent interaction between users and web applications. One example of this approach is AVANTI (Paramythis et al.,

2001), in which the system tracks the user interaction in a web browser designed, for universal accessibility, as a front-end to the AVANTI information system. This tool uses Unified User Interfaces (U2Is) from Unified User Interface Development methodology (U2ID) to achieve self-adaptability, monitoring any kind of user during the navigation, including elderly and disabled people. In this respect, DESK too is a web browser and also an authoring tool that also monitors user actions. By contrast, though, DESK uses configurable and parametrical information from different knowledge models in order to obtain meaningful information about user intents. Furthermore, an intelligent agent tracks user steps during the interaction to infer atomic changes. In this way, our authoring tool supplies the user with an easy-to-use web browser and WYSIWYG editing tool to automate changes to web presentations.

Turquoise (Miller and Myers, 1997) is an intelligent browser and editor for the web that allows users to create dynamic pages by example rather than by writing program code. With such a tool, users without programming experience can create scripts that combine data from several web pages, automate repetitive browsing or editing tasks, convert other data formats into HTML, and process submitted forms. Scripts are demonstrated by familiar browsing and editing actions, which Turquoise records and generalizes into a program. Like DESK, Turquoise is based on the PBE paradigm, where the system infers procedural information from examples of what the user wants to achieve. Turquoise operates by inferring scripts from user actions, and copies HTML contents into a special window. Besides copying and pasting elements, DESK allows for a wide spectrum of actions in a complete WYSIWYG environment, inferring high-level intents from such actions by using knowledge extracted from the underlying models. Ontology-driven mechanisms enable domain-independent user actions to be processed, and modifications to be mapped to concrete objects in the domain ontology with the aim of extracting meaningful information about real user intents. Similarly, in Scrapbook (Sugiura and Koseki, 1998) users can demonstrate which portions of web pages they are interested in by creating a personal page—that is to say, by selecting data through a web browser and copying it into the single personal page. Web data is copied directly from Netscape Navigator using internal programming interfaces. Once the personal page is created, the system automatically updates it by extracting the user-specified portions from the latest web pages. Thus, the user can browse the required information on a single page and avoid repetitive access to multiple web pages. By contrast, DESK provides the user with expressiveness enough to edit the HTML document freely. Rather than copying and pasting particular HTML fragments from other pages, DESK supports a PBE–WYSIWYG environment for inferring changes to the domain and presentation model separately. Such expressive power requires the use of an ad-hoc browser instead of a standard one, but provides improved control of the interaction with the user as much as possible.

In the TriIAs system (Bauer et al., 2000), the user interacts with a web-based application such as a travel-planning agent that is intended to create a schedule for a trip, satisfying all the constraints entered by the user. TriIAs calls an information extraction trainer that is able to learn new wrappers for extracting relevant pieces of information from web documents. Wrappers (Musela, 1999) provide a uniform access to information repositories such as databases, files and so forth. DESK's extraction of structured information, such as context semantics used by and generated from a semi-structured document (HTML and JSP code), is very similar to the way in which wrappers operate. In DESK, an agent tracks the user during the interaction, relieving him or her from having to carry out repetitive tasks (e.g. copying items from one widget

into another). In that sense, our agent uses enriched semantic information present in the monitoring model, as well as semantics coming from our ontology and domain knowledge, making the inference step easy without the necessity of using extraction languages.

WebSheets (Wolber, 2002) is another WYSIWYG authoring tool based on the PBE paradigm. This tool enables the creation of dynamic web pages with the aim of accessing and modifying databases. WebSheets uses Query By Example, by which queries are generated in the authoring environment from user actions, using an SQL-like language to request information from databases. By contrast, in DESK the domain information is stored using an ontology-based semantic model rather than a relational model. To this end, our authoring tool can handle complex changes made by the user, relating domain objects and widget structures, so that the variety of changes can be quite heterogeneous, and all of them can be applied to both domain and presentation models.

LAPIS (Miller, 2003) is a web scraper that enables high-level conceptual information to be rendered by means of a pattern library and using a simple web browser. LAPIS parses the HTML and transforms tag- and link-level elements into conceptual representations that help end-users to understand web information easily. As LAPIS does, DESK parses HTML code and characterizes information from web pages by using a data model. However, DESK provides the user with WYSIWYG mechanisms for authoring web pages, also analyzing user actions as part of the characterization process for inferring user intents.

Considering EUD-intended model-based tools, there have been interesting contributions during the last years. WebRevenge (Paganelli and Paternò, 2002) can track the reverse path of a web page. WebRevenge generates a CTT task model (Paternò, 2001) by analyzing the interaction and the interface's elements. WebRevenge works together with TERESA (Mori et al., 2002), an authoring tool for modeling applications from CCTT-based task models. TERESA handles the forward engineering and WebRevenge the reverse path, in order to provide support for web-based migration of applications to different platforms. By contrast, DESK is intended to help the user during interaction with a system rather than when using it as a multi-model generation system. DESK also takes into account both user interaction and an ontological data model, and information extracted from both is used together to infer further modifications. DESK uses a low-level task model rather than a CTT-based one, where interface objects, domain information and user actions are embedded to enrich the monitoring model with semantics used for further characterizing the user's intent. On the other hand, Bouillon et al. (2002) presents a model-based approach for web engineering. This work consists of an architecture for reverse-engineering web pages, with a view to applying forward engineering subsequently. Reengineering methods are then applied to produce new user interfaces for multiple contexts of use, thus creating a capability for the rapid production of user interfaces for different computer platforms, access devices, etc. In this respect, DESK provides a reengineering mechanism by following the inverse of the path traversed by PEGASUS, starting from the HTML code and going back to the constructor models. Such an approach allows for the mapping of final user changes to concrete domain objects with the maximum domain independence.

The use of a data model was already present in the earliest PBE systems. In a very simple form, Peridot (Myers, 1998) lets the user create a list of sample data in order to construct lists of user-interface widgets. In Gold (Myers et al., 1994) and Sagebrush (Roth et al., 1994) the user can build custom charts and graphics by relating visual

elements and properties to sets of data records. The data model in Peridot consists of lists of primitive data types. Gold and Sagebrush assume a relational data model. We also have previous experience in developing PBE- and model-based approaches such as HandsOn (Castells and Szekely, 1999). In HandsOn, the interface designer can manipulate explicit examples of application data at design-time to build custom dynamic displays that depend on application data supplied at run-time. HandsOn is based on the presentation model of MASTERMIND (Szekely et al., 1995), where the designer can build presentation objects by means of direct manipulation in a visual environment. Our view in this regard is that it is interesting to lift these restrictions and support richer information structures. We firmly believe in ontologies as a medium to specify different features of interfaces and, specifically, web-based interfaces. Ontologies provide a conceptual model in which complex relationships can be defined in order to codify high-level semantic paths for further characterization and reverse-engineering purposes (Macías et al., 2006). Our research experience is in using ontologies as a medium to specify knowledge for building data models (i.e. domain models) used together with application or presentation models. This makes it possible for our approach to specify complex knowledge focused on the interface's domain model and also work with XML-based languages that better fulfil the assumptions about distribution and sharing (Macías and Castells, 2003b).

As for commercial web-development tools, Microsoft FrontPage and Macromedia Dreamweaver can be considered as the most popular ones. These tools offer a high functionality and provide environments intended to deal with different web-based languages such as HTML, CSS, XSL, XML, JSP, ASP and so forth. Although these tools also come with multiple tool bars and debugging facilities, they are not intended for end-users. In order to modify procedural, content and presentation information, the user has to act at some point as a skilled designer, dealing with web-based languages (or at least with a visual representation of them) and being subjected to the authoring formalisms. Some studies (Rode and Rosson, 2003) revealed that, although much progress has been made by commercial web development tools, most of the end-user tools that they reviewed (including Microsoft FrontPage and Macromedia Dreamweaver) lacked not functionality but ease of use. In general, the cognitive load in carrying out editing tasks using such environments is very high, because these commercial tools are mostly intended for professional designers rather than end-users. Although providing the highest functionality is a first-order concern in commercial authoring environments, end-users might just want to accomplish customization and easy changes to concrete parts of a dynamic web interface. This implies reducing expressiveness in favor of increasing ease of use, something that is barely visible in existing commercial authoring tools today. In DESK, we provide easy mechanisms for authoring dynamic web pages, relieving the user from having to deal with programmatic representations. DESK includes less functionality than commercial tools in favor of increasing ease of use. Our tool features intelligent mechanisms intended to fulfill end-user needs, automatically modifying the underlying ontologies in PEGASUS and traversing the reverse path with no user intervention. This way, end-users can easily customize and make partial changes to dynamic web pages. Furthermore, end-users are provided with assistance during the authoring process. Therefore, users only have to achieve syntactic changes in a WYSIWYG environment, taking no notice of specification languages and of procedural information that is automatically tackled by the system.

7. Conclusion

An increasing proportion of web content and services are accessed today through dynamically generated web pages. Dynamic web pages enable user interfaces to be generated automatically, splitting content, structure and layout, which are processed on the fly depending on application data or state, user input, user characteristics, and any contextual condition that the system is able to represent. However, the development and customization of dynamic pages is a complex task that requires advanced programming skills. Most tools and technologies targeted at authoring the (semantic) dynamic web still require advanced technical knowledge that domain experts, content producers, graphic designers or even average programmers usually lack. Commercial development environments have been provided for these technologies, and they help to manage projects and provide code browsing and debugging facilities; but they are intended for expert developers rather than end-users. Consequently, web applications are expensive to develop and customize for end-users and often are of poor quality, which is currently an important hurdle for the end-user development of web applications.

Many informal user studies have revealed that the web development tool that users envision is typically "Word for Web Applications", expressing a preference for a desktop-based tool that embraces the WIMP, drag-and-drop, and copy-and-paste metaphors, and offers wizards, examples and template solutions (Rode et al., 2006). The research we present here is an effort to face such a challenge. It aims at combining the ease of use of an interactive authoring tool with the power of the model-based approach, providing an integral solution to enable end-users to modify dynamic web applications based on the semantic web.

DESK provides the designer with an intuitive authoring environment capable of addressing complex web page designs. Our authoring tool is based on the Programming by Example paradigm, where the user supplies the system with an example of what he or she wants to get and the system infers the changes to dynamic page generation procedures automatically. From monitoring user actions, DESK obtains information that will be processed together with semantic domain knowledge. Such information will be used to infer the knowledge necessary to provide the user with assistance during the authoring process. Changes are automatically carried out in the server by using both domain and presentation knowledge from PEGASUS. DESK tries to infer maximal information from user actions and from existing semantic knowledge that is independent from the application domain. To test assumptions about our approach's ease of use, we have carried out a user test. This experiment shows that is possible to reduce the 'gentle slope' of complexity by supplying an easy-to-use WYSIWYG user interface, but has revealed some limitations on expressive power, owing to the fact that DESK is focused on concrete WYSIWYG representations rather than abstract ones.

All in all, the main goal of our work is not to provide a universal solution to the issue of end-user authoring, but to find out how far one can go without leaving the WYSIWYG approach. More precisely, one of the most important aims of our work is to provide a useful authoring tool capable of inferring correct actions in a reasonable number of cases. DESK can successfully infer the following types of change in documents:

- Insertion, deletion and modification of HTML fragments within the presentation template and generation rules.
- Insertion, deletion and modification of HTML tags surrounding domain elements within the presentation template and generation rules.

- Creation, deletion, modification and automatic transformation between widgets (e.g. combo boxes, selection lists, and tables).
- Insertion, deletion, modification and moving around of dynamic text and multimedia references from the domain model.

However, the information that DESK processes must be supplied by the PEGASUS system, and hence DESK is not able to create a web page from scratch. Since the DESK authoring tool is based on PEGASUS's ontological facilities, it is not able to modify arbitrary dynamic web pages generated from other sources. Besides, to provide with intelligent support we have programmed heuristics that at present can be considered as domain-independent built-in algorithms intended to deal with our ontological RDF-like language. Such algorithms are conceptually grouped into 6 different modules (as explained in sections 3.1 and 3.2) that comprise low and high level heuristics. Concerning implementation, each module consists of pure Java algorithms. Therefore, new improvements require modification of the internal Java code by a programmer. Rule programming is about 50% of the existing java code. Nevertheless, our contribution can be regarded as a PBE semantic web approach to take on board when designing web-based generation systems, in order to improve the interaction with end-users as much as possible. In particular, we have made minimal assumptions about the user's skills in web-based languages, supplying an EUD solution that involves an automatic process of reverse engineering intended to reduce interaction efforts. Our work is based on well-known disciplines such as the Programming by Example and Model-Based User Interfaces paradigms. In our opinion, PBE and MBUI techniques can be combined together to relieve the user from having to deal with web-based languages and complex development environments not intended for end-users. Certainly, this implies some reduction in the expressive power of the MBUI approach, since end-users do not need to manipulate declarative specifications, but rather to devote all their effort to modifying the application interface to fulfill their expectations in software customization. In general terms, we believe that the user should not be aware of the interface's internal specification processes. This led us to research on formal mechanisms in order to implement intelligent authoring tools that help users modify dynamic web-based pages and thereby provide them with an approach intended to deal with their daily, non-programming-oriented, creative problem-solving activities.

As for future work, we plan to improve DESK to deal with more sophisticated cases of inference in order to provide end-users with further assistance. The results obtained from the user experiment will be also considered when making improvements, refining the heuristics for dealing with new transformations and advanced widget manipulation. We will also consider improving user and platform models for further customization. So far, we have dealt with a user model that just stores basic information (personal user information and platform characteristics), and with applying conditions inserted into the presentation template. However, one significant improvement would be to have different domain models (considering separate XML files) for each user, as well as studying advanced characteristics and navigation goals.

Acknowledgements

The work reported in this paper is being partially supported by the Spanish Ministry of Science and Technology (MCyT), projects TIN2005-06885 and TSI2005-08225-C07-06.

References

Bauer, M., Dengler, D. and Paul, G., 2000. Instructible Information Agents for Web Mining. In Proceedings of the International Conference on Intelligent User Interfaces (January 9-12, pp. 21-28, New Orleans, USA).

Boehm, B.W., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R., 1995. Cost models for future software life cycle processes: COCOMO 2.0. Annals of Software Engineering Special Issue on Software Proces and Product Measurement. J.D. Arthur and S.M. Henry, Eds. Baltzer AG Science Publishers, Amsterdam, The Netherlands.

Bouillon, L., Vanderdonckt, J. Eisenstein, J., 2002. Model-Based Approaches to Reegineering Web Pages. Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design - TAMODIA 2002, 18-19 July 2002, Bucharest, Romania, 86-95.

Brusilovsky, P., Eklund, J., Schwarz, E., 1998. Web-based Education for all: a Tool for the Development of Adaptive Courseware. Computer Networks and ISDN Systems, 30, 1-7.

Castells, P. and Szekely, P., 1999. Presentation Models by Example. En: Duke, D.J., Puerta A. (eds.). Design, Specification and Verification of Interactive Systems. Springer-Verlag, pp. 100-116.

Castells, P. and Macías, J.A., 2002. Context-Sensitive User Interface Support for Ontology-Based Web Applications. Poster Session of the 1st. International Semantic Web Conference (ISWC'02), Sardinia, Italia; June 9-12th.

Castells, P.and Macías, J.A., 2001. An Adaptive Hypermedia Presentation Modeling System for Custom Knowledge Representations. Proceedings of WebNet - World Conference on the WWW and Internet. Orlando, Florida; October 23-27. Published by AACE, pp. 148-153.

Chin, J.P., Diehl, V.A. and Norman, K.L., 1988. Development of an Instrument Measuring User Satisfaction of the Human-Computer Interface. Proceedings of ACM CHI'88 Conference on Human Factors in Computing Systems, pp. 213-218.

Communications of the ACM. Special Issue on End-User Development, 2004. September, Volume 47, Number 9.

Cypher A., 1993. Watch What I Do: Programming by Demonstration. The MIT Press.

Davis, F.D., 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. MIS Quarterly, Vol 13, No. 3 (Sep. 1989), pp. 319-340.

Dean, M., D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein., 2002. OWL Web Ontology Language 1.0 Reference W3C Working Draft 29 July. Available at http://www.w3.org/TR/owl-ref.

Darragh, J. J. and Witten, I.H., 1991. Adaptive predictive text generation and the reactive keyboard. Interacting with Computers 3, no. 1:27-50.

Klann, M., 2003. End-User Development Roadmap. In Proceedings of the End User Development Workshop at CHI Conference (Ft. Lauderdale, Florida, USA. April 5-10).

Lieberman, H., Paternò, F., and Wulf, V. (eds), 2006. End-User Development. Human Computer Interaction Series. Springer Verlag.

Lieberman, H., 2001. Your Wish is my Command. Programming By Example. Morgan Kaufmann Publishers. Academic Press, USA.

McLean, A., Carter, K., Lövstrand, L., Moran, T., 1990. User-Tailorable Systems: Pressing Issues with Buttons. ACM. Proceedings of CHI 1990: 175-182.

Macías, J.A., Puerta, A. and Castells, P., 2006. Model-Based User Interface Reengineering. HCI Related Papers of Interacción 2004. Jesús Lorés y Raquel Navarro (eds.). Springer-Verlag Volume, pp 155-162.

Macías, J.A., and Castells, P., 2005. Finding Iteraction Patterns in Dynamic Web Page Authoring. Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction jointly with The 11th International Workshop on Design, Specification and Verification of Interactive Systems (EHCI-DSVIS 2004). Tremsbüttle Castle, Hamburg, Germany. July 11-13. Rémi Bastide, Philippe Palanque and Jörg Roth (Eds.). Lecture Notes in Computer Science, Volume 3425, pp 164 – 178. Springer-Verlag.

Macías, J.A. and Castells P., 2004. An EUD Approach for Making MBUI Practical. Proceedings of the First International Workshop on Making model-based user interface

design practical: usable and open methods and tools. Hallvard Trætteberg, Pedro J. Molina, Nuno J. Nunes (eds). Joint Conference of Intelligent User Interfaces (IUI) and Computer Aided and Design of User Interfaces (CADUI). Funchal, Madeira, Portugal. January 13.

Macías, J.A. and Castells, P., 2003a. Dynamic Web Page Authoring by Example Using Ontology-Based Domain Knowledge. In Proceedings of the International Conference on Intelligent User Interfaces (IUI) (Miami, Florida, USA. January 12-15).

Macias, J.A. and Castells, P., 2003b. Using Domain Models for Data Characterization in PBE. In Proceedings of the End User Development Workshop at CHI Conference (Ft. Lauderdale, Florida, USA. April 5-10).

Macías, J.A., 2003. Authoring Dynamic Web Pages by Ontologies and Programming by Demonstration Techniques. PhD. Thesis. Departamento de Ingeniería Informática. Escuela Politécnica Superior. Universidad Autónoma de Madrid. September. http://www.ii.uam.es/~jamacias/tesis/thesis.html.

Macias, J.A. and Castells, P., 2001. An Authoring Tool for Building Adaptive Learning Guidance Systems on the Web. Active Media Technology. J. Liu et al. (Eds.). Lecture Notes in Computer Science, LNCS 2252. Springer-Verlag, pp. 268-278.

Miller, Rober C., 2003. End User Programming for Web Users. In Proceedings of the End User Development Workshop at CHI Conference (Ft. Lauderdale, Florida, USA. April 5-10).

Miller, R., Myers B., 1997. Creating Dynamic World Wide Web Pages By Demonstration. Carnegie Mellon University School of Computer Science, CMU-CS-97-131 and CMU-HCII-97-101.

Mori, G., Paternò, F. and Santoro, C., 2002. CTTE: Support for Developing and Analysing Task Models for Interactive System Design. IEEE Transactions in Sotware Engineering. IEEE Press. Vol. 28, No.8, pp. 797-813, August.

Muslea, I., 1999. Extraction Patterns for Information Extraction Tasks: A Survey. Proceedings of AAAI Workshop on Machine Learning for Information Extraction. Orlando, Florida.

Myers, B.A., 1998. Creating User Interfaces by Demonstration. Academic Press, San Diego.

Myers, B. A., Goldstein, J., and Goldberg, M, 1994. Creating Charts by Demonstration. Proceedings of the CHI'94 Conference. ACM Press, Boston, April.

Murray, T., 1998. Authoring Kowledge Based Tutors: Tools for Content, Instructional Strategy, Student Model, and Interface Design. Journal of the Learning Sciences 7, 1, 5-64.

EUD-NET. Network of Excellence on End-User Development. http://giove.cnuce.cnr.it/EUD-NET.

Paganelli, L., Paternò, F., 2002. Automatic Reconstruction of the Underlying Interaction Design of Web Applications. Proceedings of the SEKE Conference, pp. 439-445. ACM Press, Ischia.

Paramythis, A., Savidis, A., Stephanidis C., 2001. AVANTI: a universally accesible web browser. Proceedings of the HCI'2001 (New Orleans, USA, August), Lawrence Erlbaum Associates, Publishers, 91-95.

Paternò, F., 2001. Model-Based Design and Evaluation of Interactive Applications. Springer Verlag.

Puerta, A.R.; Eisenstein, J., 1999. Towards a General Computational Framework for Model-Based Development Systems. Proceedings of the International Conference on Intelligent User Interfaces (IUI). ACM Press, New York.

Rode, J., Rosson, M.B. and Pérez, M.A., 2006. End-User Development of Web Applications. Lieberman, H., Paternò, F., and Wulf, V. (eds): End-User Development. Human Computer Interaction Series. Springer Verlag.

Rode, J. and Rosson, M.B., 2003. Programming at Runtime: Requeriments & Paradigms for nonprogrammer Web Application Development. IEEE 2003 Symposium on Human-Centric computing Languages and Environments New York, pp. 23-30.

Roth, S. F., Kolojejchick, J., Mattis, J. and Goldstein, J., 1994. Interactive Graphic Design Using Automatic Presentation Knowledge. CHI'94 Conference. Boston, April, pp. 112-117.

Sahuguet, A.; Azavant, F., 2000. Building Intelligent Web Applications Using Lightweight Wrappers. Data and Knowledge Engineering.

Szekely, P., Sukaviriya, P., Castells, P. Muthukumarasamy, J.; Salcher, E., 1995. Declarative interface models for user interface construction tools: the MASTERMIND approach. Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction. Yellowstone Park, USA, August, pp. 120-149.

Shneiderman, B., 2003. Leonardo's Laptop. The MIT Press.

Sugiura, A., and Koseki, Y., 1998. Internet Scrapbook: automating Web browsing tasks by programming-by-demonstration. Proceedings of the 7th International WWW Conference (Brisbane, Australia, April). Elsevier Science.

Wolber, D., Su, Y., Chiang Yih., 2002. Designing Dynamic Web Pages and Persistence in the WYSIWYG Interface. Proceedings of the International Conference on Intelligent User Interfaces (IUI'2002). San Francisco, California, USA. January 13-16, pp. 228-229.

Figure Captions

Figure 1. Knowledge Representation in PEGASUS

Figure 2. Domain ontology in PERSEUS

Figure 3. Domain instances in PERSEUS

Figure 4. Hypermedia document generation from domain objects by applying presentation templates and rules

Figure 5. Web page generated for an instance of type `HigherCategory`

Figure 6. PEGASUS Architecture

Figure 7. DESK mechanism overview

Figure 8. DESK client-side

Figure 9. DESK server-side

Figure 10. An example of the characterization of a change

Figure 11. DESK authoring tool

Figure 12. Resulting web page after the changes have been processed

Figure 13. DESK snapshots of the web page used for the user test

Figure 14. Ease of use of DESK perceived by users

Figure 15. Usefulness of DESK perceived by users