World Wide Web (2008) 11:253–286
DOI 10.1007/s11280-008-0043-3

Intelligent Assistance in Authoring Dynamically Generated Web Interfaces

José A. Macías

Received: 8 May 2007 / Revised: 7 September 2007 /
Accepted: 22 February 2008 / Published online: 19 April 2008

Abstract Since its emergence in the early 1990s, the WWW has become not only an information system of unprecedented size, but a universal platform for the development of services and applications. However, most of the advances in web technologies are intended for professional developers, paying poor attention to end-users with no programming abilities but with explicit needs of creating and customizing web-based presentations. This provides a strong motivation for end-users to act as designers at some point, leading to an emerging role of new computing-related professionals to be considered. This paper is an effort to leverage such difficulties by providing intelligent mechanism to assist end-users in web-based authoring tasks. To carry out such a challenge, intelligent user-monitoring techniques are exploited to obtain high-level information that will be used to infer the user's preferences and assist him throughout the interaction. Furthermore, we report on how *iteration patterns* can be applied to avoid repetitive tasks that are automatically carried out on behalf of the user. In order to bring off a feasible trade-off between expressivity and ease of use, a user experiment to obtain the user's perception and evaluate the hit-rate of our system is also presented.

Keywords intelligent web-based user interfaces · end-user development · web-based programming by example · semantic web interaction · model-based user interface design

1 Introduction

A long-term goal of Human Computer Interaction is to design systems that minimize the gap between the human's cognitive conception of task and its computational representation. In this respect, considerable progress has been made over the last few years. Computer applications include new and more sophisticated user interfaces. This motivated user interaction to be one of the most important concerns in the design of today's software *artifacts*. Consequently, the new computing today is a shift from machine-centered automation to user-centered services and tools. In a nutshell, traditional computing has changed from what computers can do themselves to what people can do with computers [36].

J. A. Macías (✉)
E.P.S. Universidad Autónoma de Madrid, Ctra. De Colmenar Viejo, Km. 15, 28059 Madrid, Spain
e-mail: j.macias@uam.es

1.1 End-user interaction

As software products grow in terms of expressivity, there is a growing need to allow people to customize, configure and also create their own software *artifacts* in order to achieve daily tasks properly. This includes professionals such as engineers, scientists and freelance professionals who have concrete domain skills but generally lack programming abilities [20]. Further support is needed in order to provide non-programmer professionals with easy-to-use mechanisms to build and customize 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 simply cannot be expected to tolerate [9]. 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 [15] 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 reported that in the U.S. alone, there are 55 million end-user developers compared to 2.75 million professional software developers [2]. This advocates the idea of considering new design strategies, providing the end-user with a different role of self-designer rather than being a simple computer operator. Actually, such a trend motivated new interaction-based research to be considered. Probably the most important approach is EUD (End-User Development) [13, 16], which is focused on a user-centered approach. End-User Development can be thought of as a set of activities and techniques that allow people, including non-professional developers, at some point to create or modify a software artifact [14]. Those include intelligent and adaptive approaches such as *Programming by Example*. EUD is targeted at meeting the needs previously commented upon, enabling final users to create their own software artifacts with the minimum effort. Do-it-yourself computing is one way to perceive this flourishing field [8].

1.2 Authoring dynamically generated web-based interfaces

In the last ten years, the web has spread as a relevant information distribution medium. A great proportion of web content and functionality are accessed today through web interfaces. To provide maximal functionality, most web interfaces are dynamic rather than static. Actually, it is estimated that 80% of web pages are dynamically generated [35] by applications and services stored in web servers.

The challenge of authoring static web pages has been practically overcome years ago by well-known commercial tools. Using such applications it is possible to make changes to web pages and in turn upload them into web servers, enabling the user to avoid editing HTML directly. However, authoring dynamic web pages comprises a far more complex concern, since it requires users to have programming skills. At present, most web pages built by end-users simply present information; creation of interactive web sites or web applications such online forms, surveys and interactive web applications 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 [34]. Actually, no definitive solution has still been proposed to provide users with easy mechanisms for authoring web-based dynamic information generated by databases, web-based services, ontologies and other repositories. 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 that domain experts, graphic designers or even average programmers may lack. A small number of development environments have been provided to easily deal with all these technologies. Admittedly, these tools help manage projects and provide code browsing and debugging facilities, but one still has to deal with the code. Additionally, users might want to customize only a concrete part of a web application, having no need of dealing with the whole programmer-targeted development environment when carrying out simpler modifications. An interesting study by Rode and Rosson [33] 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) to offer web application features more suitable to end-users. Although tools like Dreamweaver and FrontPage have substantial extension APIs, Rode and Rosson found the inflexibility in controlling the users' workflow as 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* [23], the skills required to implement advanced features should only grow in proportion to the complexity of the desired functionality. In general, it is difficult to provide what-you-see-is-what-you-get (WYSIWYG) tools for the development of dynamic web pages because it is difficult to describe procedural behavior visually. This is an inherent problem concerning most authoring environments, which implicitly reduce the user's ability to create and modify (web-based) software artifacts.

1.3 The approach

This paper presents an approach aimed at authoring dynamically generated web-based pages by end-users. This approach consists of two tools intended to minimize the effort in authoring such web-pages. On the one hand, DESK [17, 18] is an interactive authoring tool that allows the customization of dynamic-page generation procedures with no a-priori tool-specific skill requirements from authors. On the other hand, PEGASUS [4, 5] generates HTML pages from a structured domain-model and an abstract presentation-model. The approach consists of combining intelligent GUI-design techniques such as Programming By Example (PBE) [9, 15] with a bespoke ontology-based representation of knowledge. DESK acts as a client-side complement of the PEGASUS dynamic web-page generation system. Such a solution attempts to smooth the gentle slope of complexity in software usage [20], decreasing the general expressivity 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 rendered and how a certain unit of information from the domain model is presented to the user. Instead of using the PEGASUS modeling language, authorized users can modify the internal presentation model by editing in DESK the HTML pages generated by PEGASUS. DESK detects *iteration patterns* and follows the Programming By Example approach to infer changes that affect every class of knowledge from the user's actions. 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.

In a nutshell, the main contribution of this work is providing an environment that automatically customizes web presentations from changes performed by users. Furthermore, an intelligent agent is also provided to monitor the user's activity and build a high-level task model to reason about and infer the user's intent. Inferred information is exploited to personalize web presentations, which will be dynamically generated depending on previous user changes. The conceptual separation between content and presentation, which is implicitly carried out by the system using an ontology-driven approach, provides useful mechanisms to characterize the user's intent. This involves exploiting semantics to better characterize user preferences at interaction. Additionally, a trade-off between expressiveness and easy-of-use should be considered. In this respect, we carry out such a compromise by supporting less expressive but likely far more easy-to-use authoring facilities, which are mostly intended to help and assist non-skilled end-users in authoring complex and cumbersome tasks.

This paper is organized as follows. Section 2 describes PEGASUS's mechanisms as well as the ontology-based underlying models used in dynamic page generation. Section 3 presents DESK and the inference mechanisms used in authoring the adaptive dynamic web pages generated by PEGASUS. Section 4 discusses an experiment carried out by users utilizing DESK. Section 4 also reports related work. Finally, Section 5 provides the conclusions to complete the structure of this paper.

2 Knowledge representation and automatic generation of web interfaces

Ontologies can be regarded as an effective way to model different aspects of a user interface and to provide conceptual models in which complex relationships can be defined. Such a conceptualization can be used in order to code high-level semantic paths for automatic web-based interface generation, further characterization and reverse-engineering purposes [19]. The author's research experience is in using ontologies to specify knowledge for building data models (domain models) used together with application or presentation models. Our previous experiences helped us to address the problem of specifying complex knowledge focused on the interface's domain and presentation models, as well as working with XML-based languages that better fulfill the assumptions about knowledge distribution and sharing that we have implicitly presented in this work. Therefore, this work is focused on combining ontologies with Model Based User Interface (MBUI) techniques [30, 32], which emerged as the 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 user and platform, domain knowledge, presentation, dialog and behavior.

PEGASUS (*P*resentation modeling *E*nvironment for the *G*eneration of ontology-*A*ware context-*S*ensitive web *U*ser interface*S*) is a domain-independent system that helps to create a dynamic front-end for ontology-driven knowledge-based applications on the web [5]. PEGASUS supports the definition of made-to-measure ontologies for the description of domain knowledge. 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 (i.e., instances of the domain ontology) and assembling them into HTML documents in response to a user’s requests for concrete knowledge units. Since PEGASUS’s ontological processing and specification is not the main concern in this paper, we summarize in next subsections the most important concerns about this system for the sake of brevity. Instead, we will later focus on DESK and the *iterative pattern* detection, which is the strength of this paper. Interested readers can refer to [4, 5] in order to find further detail about PEGASUS.

2.1 PEGASUS’s domain model

The domain model in PEGASUS comprises a semantic network of ontology classes, 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 the presented approach, ontologies can be defined with a high degree of freedom, with very generic classes like Catalog, Product, or more specific, like E-Mail Software, and Multimedia Tools. All this knowledge is captured by defining attributes for classes, and relations between classes [4].

As an example, a designer could build a domain ontology in PEGASUS for a software download site like Tucows, defining ontology classes like Product, Category, HigherCategory, LowerCategory, and Catalog. This way, instances such as Internet, E-Mail, E-Mail Clients and Allegro Mail 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="Agile Mail"/>
    <Product ref="Allegro Mail"/>
    ...
  <products>
</LowerCategory>

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

XML attributes like license and price correspond to properties of a knowledge unit (of class Product), whereas elements like subCategories and products are relations with other units (the ref attribute corresponds to the unit ID's). While the current version of the approach uses ad-hoc XML extensions to represent the domain model for historical reasons, it is planned to move to some of the currently available ontology definition standards like RDF or OWL [12], with minor modifications to the system.

2.2 PEGASUS's presentation model

In contrast with other knowledge-based systems that achieve automatic page generation; e.g., Adaptive Hypermedia systems [3, 28], PEGASUS provides extensive control over presentation design, by using an explicit presentation model, apart from contents. The separation of content and presentation is achieved by defining a *presentation template* for each class of the ontology. Templates define what parts (attributes and relations) of a knowledge item must be included in its presentation and in what order, as well as their visual appearance and layout. 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, Categoryof-Product 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. Templates are defined by using an extension of HTML based on JavaServer Pages (JSP), that allows inserting 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
```

This template indicates that when there is enough available space (which is estimated on a scale from 0 to 10) a table is created 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. 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 like tables, input types (buttons, combo boxes, etc.), and selection lists.

Each widget type has specific mechanisms to display 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 1, 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 (a few details like cell background colors and the tabbed bar have been omitted in the template code for the sake of brevity).

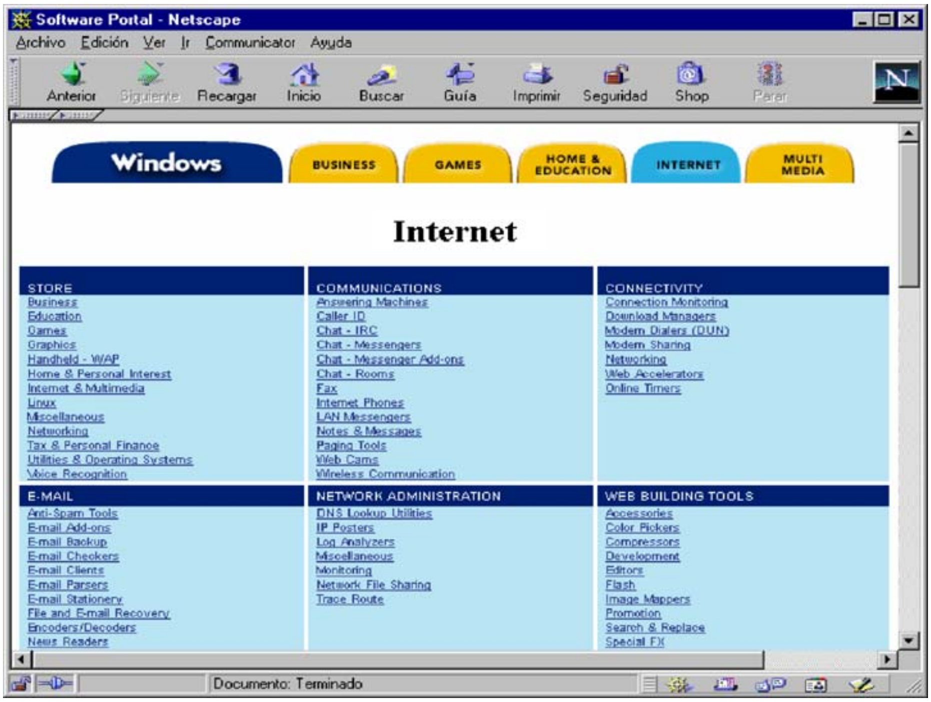
Besides templates, the PEGASUS presentation model also includes presentation rules like the following:

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

In Figure 1, above rule is responsible for presenting 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, properties of the data, characteristics of the platform, and any other aspect that should influence presentation, like task requirements, user's goals, usage modes (e.g., exploration vs. selective search), etc.

At runtime, the user interacts with the application through a web browser. Interaction with an application built with PEGASUS consists of navigating through the semantic



Internet

STORE	COMMUNICATIONS	CONNECTIVITY
Business	Answering Machines	Connection Monitoring
Education	Caller ID	Download Managers
Games	Chat - IRC	Modem Dialers (DUN)
Graphics	Chat - Messengers	Modem Sharing
Handheld - WAP	Chat - Messenger Add-ons	Networking
Home & Personal Interest	Chat - Rooms	Web Accelerators
Internet & Multimedia	Fax	Online Timers
Linux	Internet Phones	
Miscellaneous	LAN Messengers	
Networking	Notes & Messages	
Tax & Personal Finance	Paging Tools	
Utilities & Operating Systems	Web Cams	
Voice Recognition	Wireless Communication	
E-MAIL	**NETWORK ADMINISTRATION**	**WEB BUILDING TOOLS**
Anti-Spam Tools	DNS Lookup Utilities	Accessories
E-mail Add-ons	IP Posters	Color Pickers
E-mail Backup	Log Analyzers	Compressors
E-mail Checkers	Miscellaneous	Development
E-mail Clients	Monitoring	Editors
E-mail Parsers	Network File Sharing	Flash
E-mail Stationery	Trace Route	Image Mappers
File and E-mail Recovery		Promotion
Encoders/Decoders		Search & Replace
News Readers		Special FX

Figure 1 Web page generated for an instance of type HigherCategory.

network of domain objects. Each time the user moves to an object, PEGASUS responds by generating an HTML page (see Figure 2). In doing so the system a) resolves the user's request by determining the actual object to move to, b) locates the instance in the domain model, c) updates the domain and user models and d) 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.

From the PEGASUS point of view, the unit of interaction with the user is an HTTP request. User model updates are carried out by taking into account only the information extracted from client's requests. Platform and user interface characteristics are captured at the 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 an 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. By contrast, it means that the system is not explicitly aware of user activity between two requests, and 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 handle 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 [17], which is intended to create specific domain knowledge (class, objects and relationships) for PEGASUS. PERSEUS (Presentation ontology buildER for cuStom lEarning sUpport Systems) is an interactive form-driven tool that was originally used for creating PEGASUS adaptive hypermedia e-learning systems [4] by automatically generating the XML domain-information to be processed by PEGASUS.

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

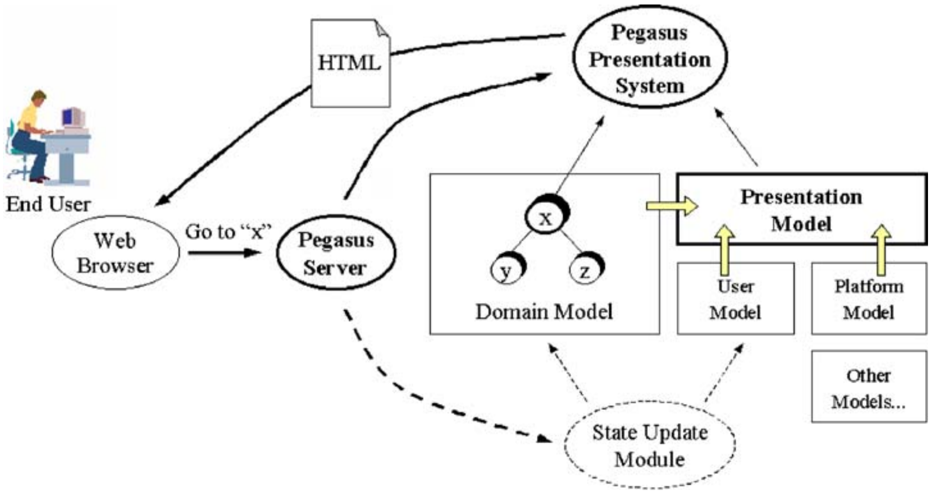


Figure 2 PEGASUS architecture.

domain ontology to the presentation objects—i.e., those used in the presentation template to render domain 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.

DESK was conceived to leverage the authoring mechanisms and deal with web customization easily. DESK fulfils the assumptions made and provides with automatic support for accomplishing designs involving domain, presentation and procedural information under the same authoring environment. Therefore, the user does not need to get involved in the reverse path that the system follows automatically to carry out the required modifications.

3 Authoring dynamic web pages through DESK

In this approach, EUD paradigm is particularly considered in order to deploy ontology-based MBUI techniques that relieve users from having to deal with specifications. 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 [22]. 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 [19].

Users can modify the design of dynamic web documents through DESK, 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 3 depicts how DESK works. DESK has both client-side and server-side components. The client-side 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 PEGASUS's underlying models (6). Affected web pages will be dynamically regenerated and will appear suitably modified whenever the user navigates through them.

3.1 Inferring user intentions

DESK authoring tool uses a set of suitable heuristics consisting of advanced ontology-based search 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

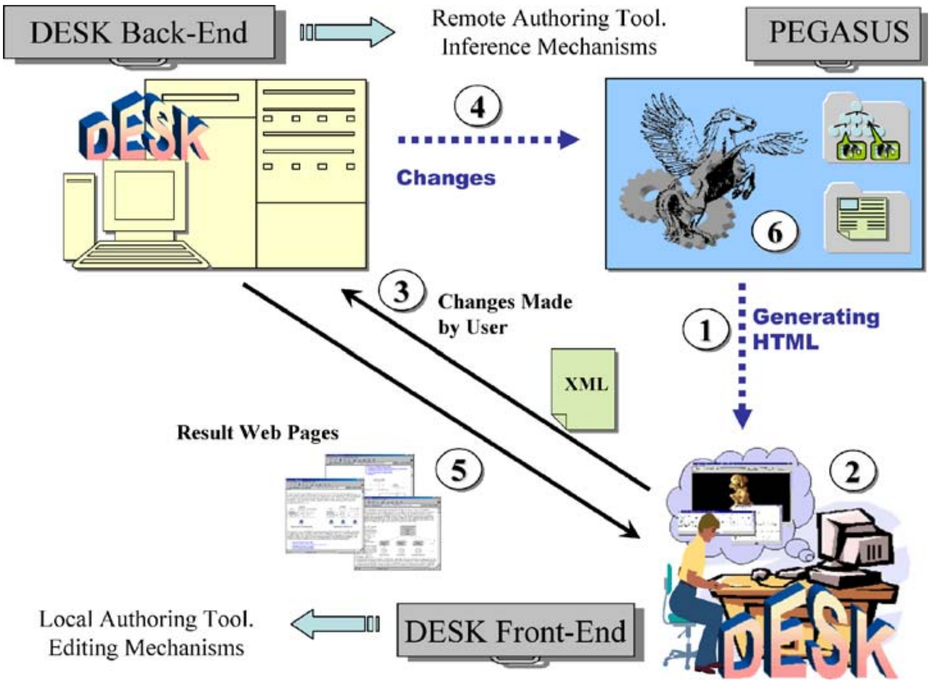


Figure 3 DESK mechanism overview.

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 precise and meaningful information about the user's intent. In general terms, DESK heuristics deploy available knowledge from the 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* (H_L). In turn, contextual information and user actions are packed into *constructor primitives* to form the monitoring model (Figure 4). Low-level heuristics determine the syntactic context for every user action [21]. Syntactic context is useful to obtain local information about where the changes take place in the HTML code, thus providing with 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 nearest syntactic context for every modification. Candidate context includes references to other surrounding HTML objects and text fragments that could be useful in order to identify mappings among HTML code and domain objects.
- *The special-structure location module* identifies presentation structures (e.g., tables, selection lists, etc.) in which a modification occurs. This module knows

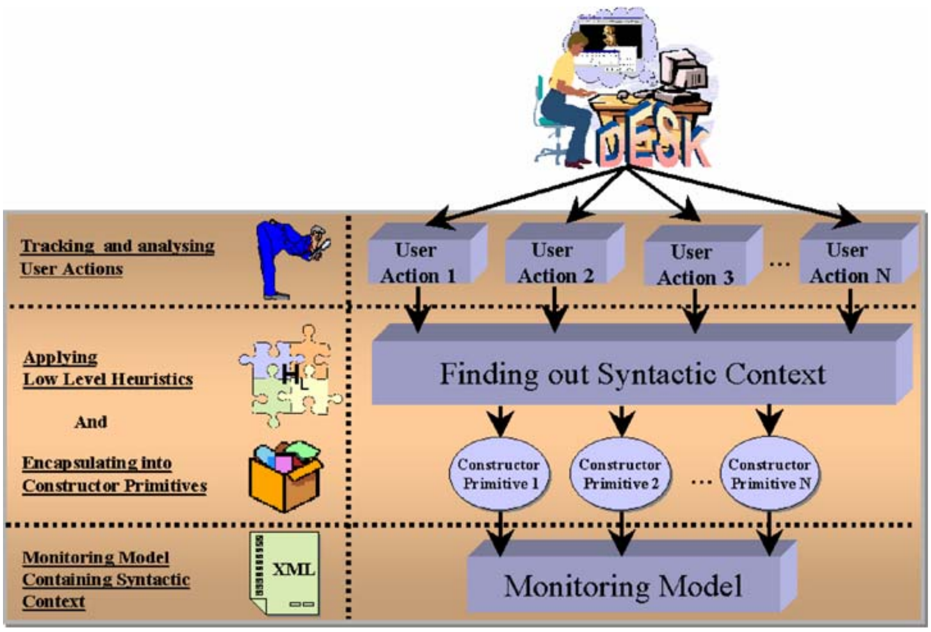

DESK
Tracking and analysing User Actions
User Action 1
User Action 2
User Action 3
...
User Action N
Applying Low Level Heuristics
And
Encapsulating into Constructor Primitives
H_L
Finding out Syntactic Context
Constructor Primitive 1
Constructor Primitive 2
...
Constructor Primitive N
Monitoring Model Containing Syntactic Context
XML
Monitoring Model

Figure 4 DESK client-side.

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 from previous modules—that is to say, user actions and 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 Inferring meaningful transformations

Once the monitoring model has been created, it is sent to the server for further processing. Figure 5 shows the back-end architecture of DESK. The client-side sends the monitoring model to DESK's server-side component, where inference takes place.

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* (H_H) have been defined [21]. 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 out 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

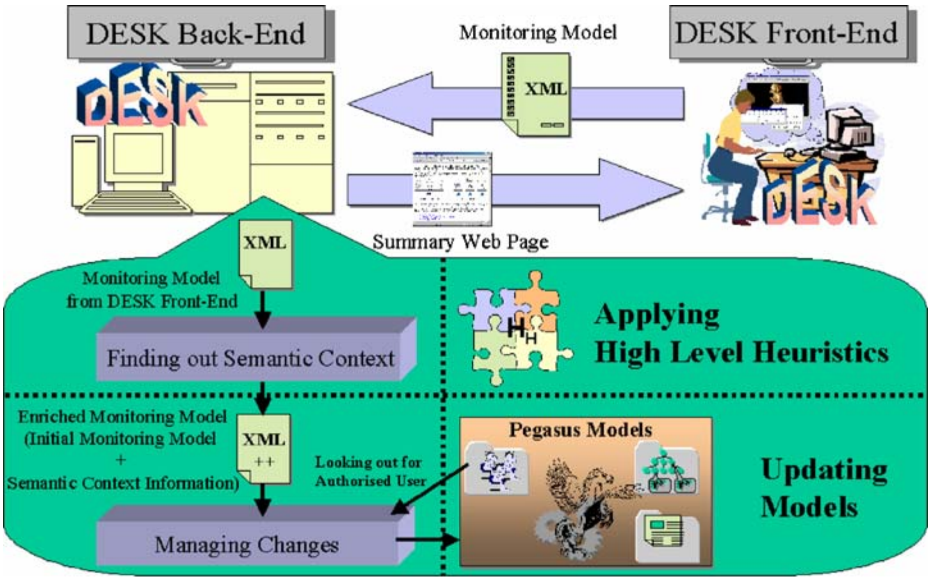

DESK Back-End
Monitoring Model
XML
DESK Front-End
Summary Web Page
Monitoring Model from DESK Front-End
Finding out Semantic Context
Applying High Level Heuristics
Enriched Monitoring Model (Initial Monitoring Model + Semantic Context Information)
XML ++
Looking out for Authorised User
Pegasus Models
Updating Models
Managing Changes

Figure 5 DESK server-side.

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 attributes 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. As an example, let us suppose that the user edits the title of an e-mail client called "Allegro Mail", adding the word "Client" at the end.

Before	After
• Allegro Mail	• Allegro Mail Client
• Item 2	• Item 2
• Item 3	• Item 3
• ...	• ...
• Item N	• Item N

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). The line Allegro Mail was generated by a <%=subcategories ("vertical")%> instruction in the presentation template. Such a command establishes different categories of email products (in this case) to be visualized vertically by a selection list. 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. This way, an occurrence is found, as the title attribute of object EMC1 seems to contain such a string:

```
<eMail-Clients ID="EMC1" title="Allegro Mail">
...
<eMail-Clients ID="EMC2" title="Item 2">
...
<eMail-Clients ID="EMCN" title="Item N">
...
```

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:

```
<E-Mail ID="EM">
  <BelongsTo>
    <eMail-Clients ID="EMC1"/>
    <eMail-Clients ID="EMC2"/>
    ...
    <eMail-Clients ID="EMCN"/>
     ...
```

Analyzing this object and searching the domain model once again, the system finally finds the class Software, where the relation BelongsTo appears:

```
<Class name="Software">
  <Relation name="BelongsTo"...>
...
```

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."

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 2î/>
</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.

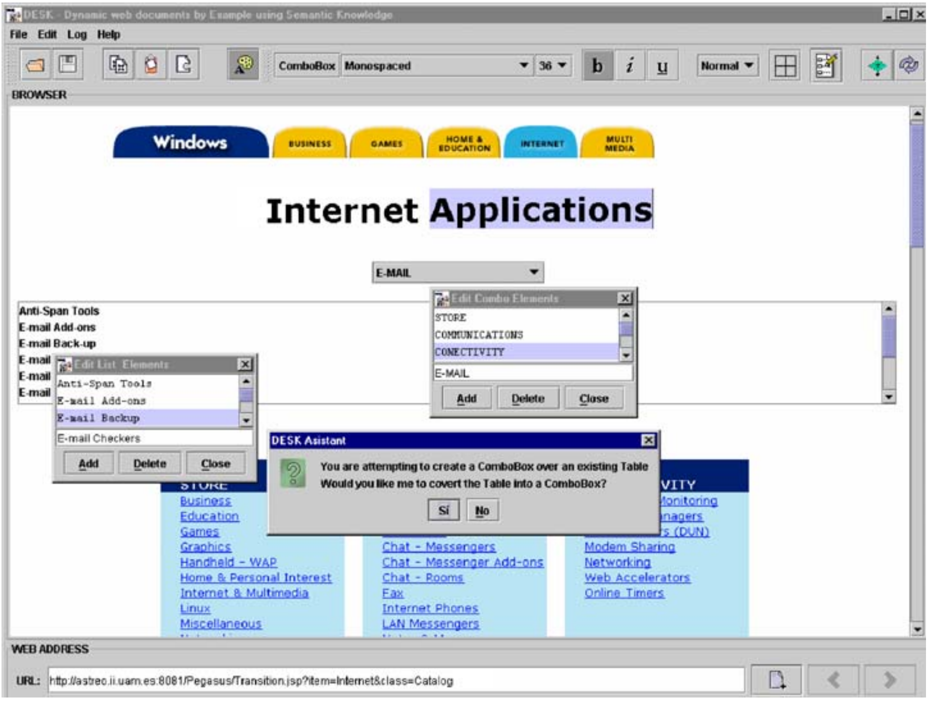
The process of running high-level heuristics enriches the monitoring model with information resulting from the characterization explained above—that is, semantic knowledge such as concrete domain-object names, attributes and semantic relationships. Finally, a specialized module for managing changes processes 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 the above example, the attribute title of object EMC1 is readily modified by such changes to the monitoring module.

Once performed, changes are only visible to the author who carried them out. This way, each user can see the presentation according to the changes s/he accomplished; presentation objects are rendered depending on the user profile. Instead of considering one presentation and domain-object-network per user, the system stores only the modified instances and identifies the corresponding changes in the user model when the page is generated again.

3.3 An example

At first sight, DESK client application looks like an HTML editor. It provides support 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 coded 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 6 shows the user interface of DESK client, where the web page depicted in Figure 1 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 when double-clicking on the widget. It is worth noting in Figure 6 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 7 shows the result of such a process, once DESK has changed the presentation model on the server-side (the internal mechanism of DESK client- and server-sides are detailed in later sections). As we can see, the inserted text "Applications" appears twice in the final



Anti-Span Tools
E-mail Add-ons
E-mail Back-up
E-mail
E-mail
E-mail

Edit List Elements: Anti-Span Tools / E-mail Add-ons / E-mail Backup; E-mail Checkers

Edit Combo Elements: STORE / COMMUNICATIONS / CONECTIVITY; E-MAIL

DESK Asistant

You are attempting to create a ComboBox over an existing Table
Would you like me to covert the Table into a ComboBox?

Sí No

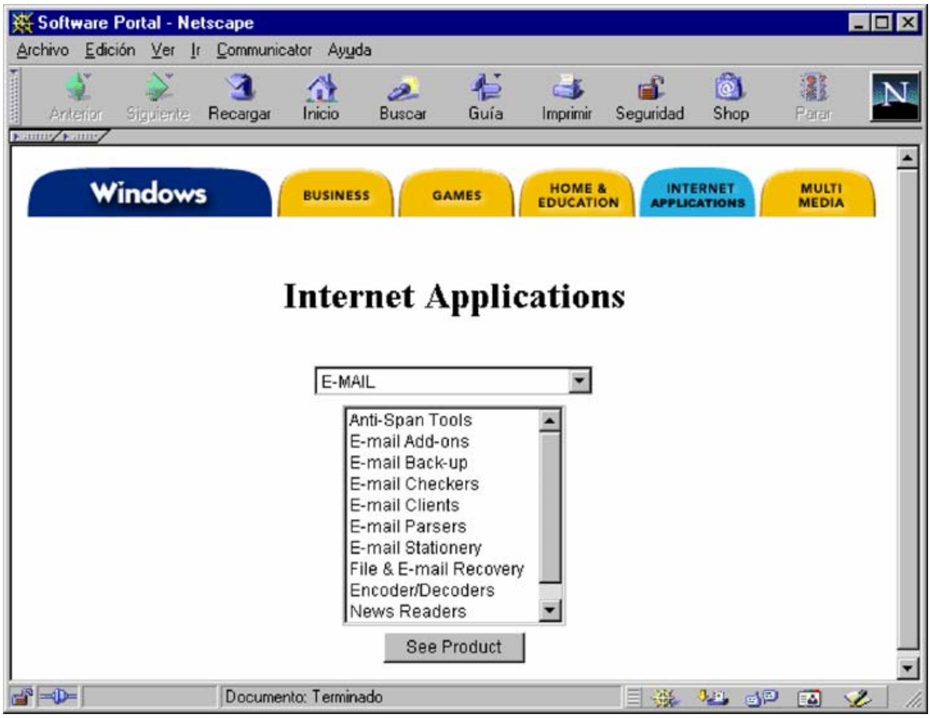
Business
Education
Games
Graphics
Handheld - WAP
Home & Personal Interest
Internet & Multimedia
Linux
Miscellaneous

Chat - Messengers
Chat - Messenger Add-ons
Chat - Rooms
Fax
Internet Phones
LAN Messengers

Modem Sharing
Networking
Web Accelerators
Online Timers

URL: http://astreo.ii.uam.es:8081/Pegasus/Transition.jsp?item=Internet&class=Catalog

Figure 6 DESK authoring tool in editing mode.


Software Portal - Netscape
Archivo Edición Ver Ir Communicator Ayuda
Anterior Siguiente Recargar Inicio Buscar Guía Imprimir Seguridad Shop Parar
Windows
BUSINESS
GAMES
HOME & EDUCATION
INTERNET APPLICATIONS
MULTI MEDIA
See Product
Documento: Terminado

Internet Applications

E-MAIL

Anti-Span Tools
E-mail Add-ons
E-mail Back-up
E-mail Checkers
E-mail Clients
E-mail Parsers
E-mail Stationery
File & E-mail Recovery
Encoder/Decoders
News Readers

Figure 7 Resulting web page once the changes have been processed.

presentation (page title and tabbed pane at the top). This is because the change concerns an attribute, called title, included in the corresponding domain ontology's instances, 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.

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 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 will be explained in the following sections, semantic information combined with the user's syntactic actions offers interesting possibilities for user assistance, enabling to handle many transformations automatically.

3.3.1 Assistance in authoring tasks

As 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 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 intelligent 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 for lighter conditions and detects *iteration patterns* [21]. This will be detailed in Section 3.4. Only one agent is needed in order to check the monitoring model and detect different types of actions. The 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. This task must be carried out by experts. 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) and (*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 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 elaborated 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 semantics (*3*) reflect 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 changes by configuring the agent’s behavior in order to carry through meaningful transformation by using the monitoring model. Although DESK-A will be detailed in Section 3.4, interested readers can refer to [17] in order to find further cases of transformations that have been omitted in this paper for the sake of brevity.

3.3.2 Exploiting semantics

Once the monitoring model has been sent to the server-side 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 DESK server-side 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.

3.4 Iteration patterns

Iteration patterns can be though of as a generalization of common user actions that can appear more than once, so that they can be used to apply similar behavior on future interaction. In a practical way, iteration patterns provide automatic mechanisms to assist the user in achieving cumbersome tasks.

In order to deal with iteration patterns, the system exploits the monitoring model to extract meaningful information to reason about. As already explained, the monitoring model can be regarded as a low-level task model where all the actions that the user achieves on the web interface are stored and enriched with information about the interface itself. This way, one of the advantages of using a monitoring model is that a semantic history of user actions can be built in real time. In doing so, the system features DESK-A, a specialized agent included in DESK that analyses and manages the monitoring model to find iteration patterns.

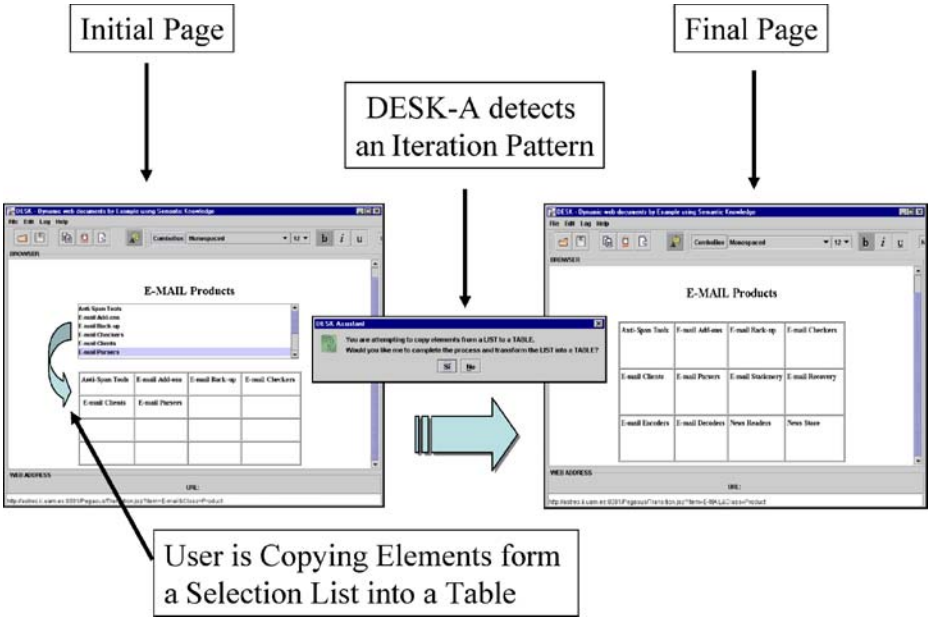
3.4.1 Detecting iteration patterns

Detecting iteration patterns consists of analyzing the history of user actions to find out meaningful information about the user's high-level tasks. To carry out this challenge, the system implements a set of heuristics for identifying relationships among the user's actions and the interface's presentation elements. More precisely, the system detects linear relationships in the geometric structure of each widget to identify two different types of interaction patterns: *regular patterns* and *non-regular patterns*.

Regular patterns can be considered as linear iteration sequences that can be detected by means of specialized algorithms. Such algorithms attempt to detect linear relationships on widget attributes (e.g., columns and rows in a table, a list of numbered items in a selection list, and so forth). By contrasts, non-regular patterns are meant to be iteration sequences where no a-priori linear relationships can be found by analyzing widget attributes. Consequently, they have to be tackled apart.

Regular patterns Regular patterns are detected and processed by means of specialized heuristics called Iteration Pattern Algorithms (hereafter, IP Algorithms). IP Algorithms are a set of algorithms specialized in studying widgets and extracting specific properties from them. Such properties will help find suitable iteration masks for moving elements automatically from one widget into another, keeping the same domain model values and mappings.

Figure 8 shows some snapshots of DESK where an automatic transformation of widgets takes place. The figure depicts how the user is attempting to copy elements from a selection


Initial Page
DESK-A detects an Iteration Pattern
Final Page
E-MAIL Products
DESK Assistant
Si
No
E-MAIL Products
User is Copying Elements form a Selection List into a Table

You are attempting to copy elements from a LIST to a TABLE.
Would you like me to complete the process and transform the LIST into a TABLE?

Figure 8 The scenario depicts how DESK-A has detected an automatic transformation of a selection list into a table.

list into a table previously created. After a couple of intents, DESK-A asks the user for confirmation in order to transform the selection list into a table. The tool accomplishes the transformation automatically once the user has accepted the suggestion. Finally, the selection list has been replaced with a table that has the same number of items and internal domain-model mappings.

IP Algorithm is a key component in achieving automatic transformations. There are several IP Algorithms that can are applied depending on the type of widget the system deals with. A sample code of one of these algorithms (based on the scenario depicted in Figure 8) for addressing transformation of tables and selection lists is as follows:

```
IP_Algorithm (Widget W1, W2, Set TG) {
  ColumnSequence       = A.getColumnSequence(W2);
  RowSequence          = A.getRowSequence(W2);
  ElemIndexSequence    = A.getElementIndexSequence(W1);
  ColJumpSet           = ColSequence.getColJumpSet();
  RowJumpSet           = RowSequence.getRowJumpSet();
  ColShiftSet          = BuildColShiftSet(ColumnSequence,
                         ColJumpSet,RowJumpSet);
  RowShiftSet          = BuildRowShiftSet(RowSequence,
                         ColJumpSet,RowJumpSet);
  Iterator             = BuildIterator(W2.getBounds(),
                         TG, ColShiftSet, RowShiftSet,
                         ElemIndexSequence);
  ...
  While (Iterator.hasNext()) {
     i = Iterator.getNexti(i);
     j = Iterator.getNextj(j);
     k = Iterator.getNextk(k);
     W2.setElementAt(i,j,W1.getElementAt(k));
  }
}
```

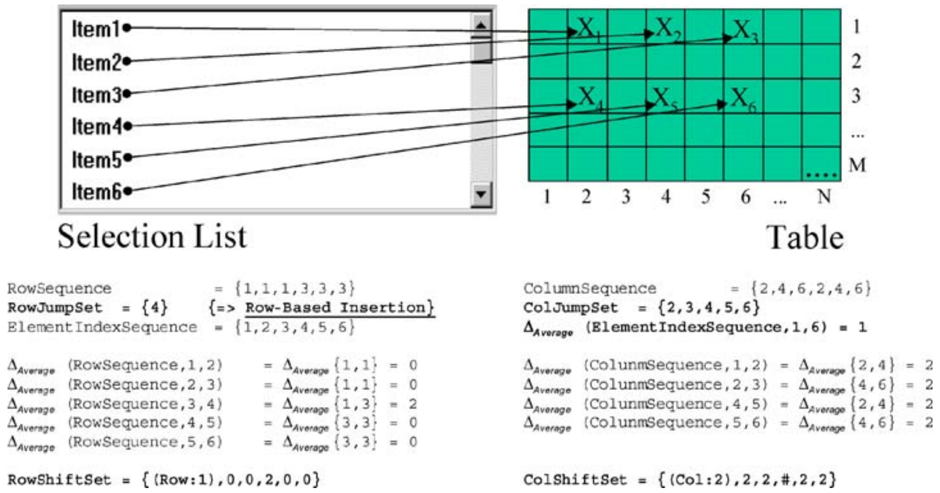
W1 represents the source widget (a selection list) and W2 is the destination one (a table). TG contains information about the widget's properties (number of fixed columns and rows). A is a set that stores information about actions that concern the process of copying elements from one widget into another. This set is very useful in order to obtain common properties about the widget's manipulation sequence (for example, the column insertion sequence of elements into a table), as well as to obtain an abstract model about the widgets that are being manipulated by the user throughout the interaction. Properties stored in A can be accessed by means of specialized methods:

- A.getSize(Widget)
- A.getElementIndexSequence (Widget)
- A.getColumnSequence(Widget)
- A.getRowSequence (Widget)
- A.getElementAt(Widget,i[,j])

- A.getID(Widget)
- A.getClassName(Widget)
- A.getObjectName(Widget)
- A.getExistsRelation(Widget1,Widget2)

The main goal of the above operators is to provide the inference engine with information about the widget (and its properties), such as the size of a given widget, the insertion sequence of elements (index, column and row), the class and the object's names as they appear in the domain model, and the existing relationships between the source widget and the destination one. Therefore, the engine builds an iteration mask (Iterator) that provides an efficient mechanism for automatically copying elements from the source widget into the destination one, adapting the properties of the destination widgets as the original one appears in the underlying models of the interface.

Figure 9 depicts an example (based on Figure 8) as the result of executing the above algorithm for copying elements from a selection list into a table. Before transforming the selection list intro a table, the system generates specific sets that store information concerning the rows and columns involved as well as the jump sequence's sets. Finally, a couple of iteration masks are calculated for both column and row, those intended to create an automatic iteration process for carrying out the transformation among widgets. As shown in Figure 9, ColumnSequence and RowSequence sets contain, respectively, the column and the row insertion sequences of elements copied into the table at each user step. On the other hand, ElemIndexSequence contain the sequence of items selected for being copied from the selection list. Furthermore, the IP Algorithm calculates the column (ColJumpSet) and the row (RowJumpSet) jump sets by processing A. The algorithm also detects whether the insertion is carried out either on rows or columns by comparing both jump sets. This way, if RowJumpSet is greater (in size) than ColJumpSet, then the insertion is achieved by iterating through rows, if not the insertion is achieved by iterating through columns. Otherwise, if both sets have the same size, then special considerations has to be taken (there is a straight linear relationship between row and column on the insertion



RowSequence = {1,1,1,3,3,3}
RowJumpSet = {4} **{=> Row-Based Insertion}**
ElementIndexSequence = {1,2,3,4,5,6}

$\Delta_{Average}$ (RowSequence,1,2) = $\Delta_{Average}$ {1,1} = 0
$\Delta_{Average}$ (RowSequence,2,3) = $\Delta_{Average}$ {1,1} = 0
$\Delta_{Average}$ (RowSequence,3,4) = $\Delta_{Average}$ {1,3} = 2
$\Delta_{Average}$ (RowSequence,4,5) = $\Delta_{Average}$ {3,3} = 0
$\Delta_{Average}$ (RowSequence,5,6) = $\Delta_{Average}$ {3,3} = 0

RowShiftSet = {(Row:1),0,0,2,0,0}

ColumnSequence = {2,4,6,2,4,6}
ColJumpSet = {2,3,4,5,6}
$\Delta_{Average}$ (ElementIndexSequence,1,6) = 1

$\Delta_{Average}$ (ColunmSequence,1,2) = $\Delta_{Average}$ {2,4} = 2
$\Delta_{Average}$ (ColunmSequence,2,3) = $\Delta_{Average}$ {4,6} = 2
$\Delta_{Average}$ (ColunmSequence,4,5) = $\Delta_{Average}$ {2,4} = 2
$\Delta_{Average}$ (ColunmSequence,5,6) = $\Delta_{Average}$ {4,6} = 2

ColShiftSet = {(Col:2),2,2,#,2,2}

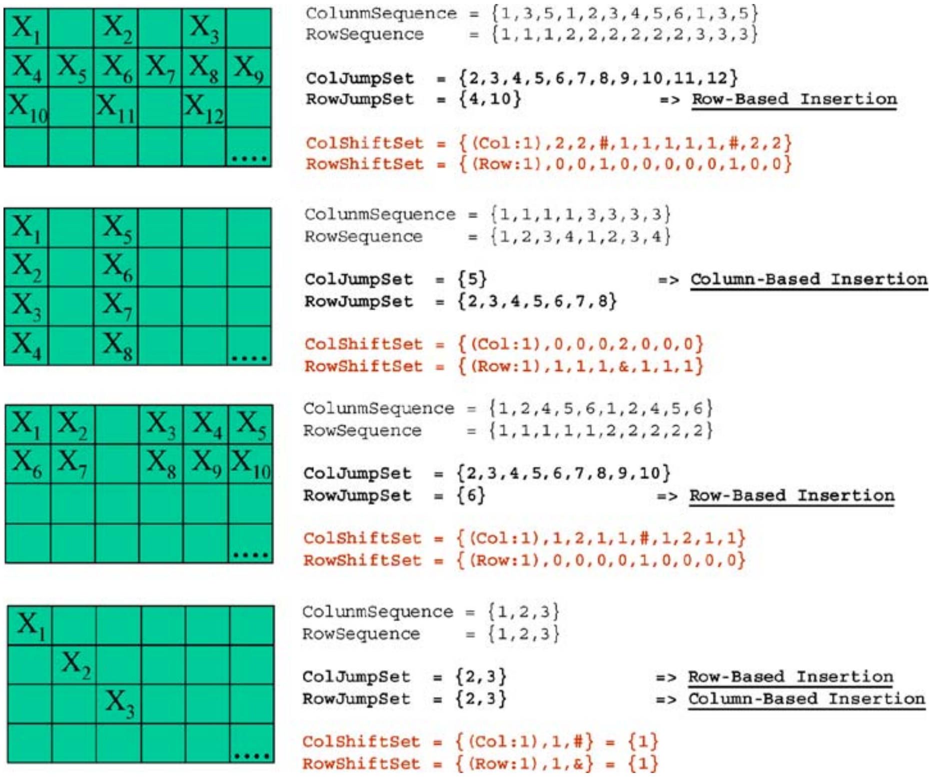
Figure 9 Execution of an IP Algorithm when copying elements from a list into a table.

sequence). Next, an increment mask is calculated for columns (ColShiftSet) and rows (RowShiftSet) by using an operator, namely $\Delta_{Average}$ defined in Equation (1).

$$\Delta_{\text{Average}}(x_1, x_2, x_3, \ldots, x_n) = \begin{cases} \frac{(x_2-x_1)+(x_3-x_2)+\ldots+(x_n-x_{n-1})}{n-1}, & n > 1 \\ 0, & n \leq 1 \end{cases} = \begin{cases} \frac{x_n-x_1}{n-1} & n > 1 \\ 0, & n \leq 1 \end{cases} \quad (1)$$

Equation (1) represents an operator that calculates the average sequence of jumps. The operator is applied to obtain a couple of masks (ColShiftSet and RowShiftSet sets) that include the increments used in the loop for column and row jumps. Initial positions are also considered at the beginning of the loop (Col:2 and Row:1), resulting in this case as follows: increasing 2 columns for the first time, jumping then two more rows (# in RowShifSet and 2 in ColShiftSet), next jumping 2 columns, and finally repeating the sequence all over again. All these sets are finally used to create the iteration index intended to iterate though the widgets and easily complete the iteration sequence previously calculated.

Figure 10 shows several examples of similar transformation processes, where different cases of tables with different types of insertion sequences are depicted. Those result in different values for each set depending on each widget's geometry. As shown, the algorithm can face correctly a great deal of cases where cut-in columns and rows are detected as part of the iteration mask, using & symbol for row-based jumps and # for column-based jumps.



X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8 X_9 X_{10} X_{11} X_{12} ....
X_1 X_5 X_2 X_6 X_3 X_7 X_4 X_8 ....
X_1 X_2 X_3 X_4 X_5 X_6 X_7 X_8 X_9 X_{10} ....
X_1 X_2 X_3 ....

```
ColunmSequence = {1,3,5,1,2,3,4,5,6,1,3,5}
RowSequence    = {1,1,1,2,2,2,2,2,2,3,3,3}

ColJumpSet  = {2,3,4,5,6,7,8,9,10,11,12}
RowJumpSet  = {4,10}                  => Row-Based Insertion

ColShiftSet = {(Col:1),2,2,#,1,1,1,1,1,#,2,2}
RowShiftSet = {(Row:1),0,0,1,0,0,0,0,0,1,0,0}
```

```
ColunmSequence = {1,1,1,1,3,3,3,3}
RowSequence    = {1,2,3,4,1,2,3,4}

ColJumpSet  = {5}                     => Column-Based Insertion
RowJumpSet  = {2,3,4,5,6,7,8}

ColShiftSet = {(Col:1),0,0,0,2,0,0,0}
RowShiftSet = {(Row:1),1,1,1,&,1,1,1}
```

```
ColunmSequence = {1,2,4,5,6,1,2,4,5,6}
RowSequence    = {1,1,1,1,1,2,2,2,2,2}

ColJumpSet  = {2,3,4,5,6,7,8,9,10}
RowJumpSet  = {6}                     => Row-Based Insertion

ColShiftSet = {(Col:1),1,2,1,1,#,1,2,1,1}
RowShiftSet = {(Row:1),0,0,0,0,1,0,0,0,0}
```

```
ColunmSequence = {1,2,3}
RowSequence    = {1,2,3}

ColJumpSet  = {2,3}                   => Row-Based Insertion
RowJumpSet  = {2,3}                   => Column-Based Insertion

ColShiftSet = {(Col:1),1,#} = {1}
RowShiftSet = {(Row:1),1,&} = {1}
```

Figure 10 Some examples of iteration patterns detected by IP algorithms.

Figure 10 also shows a case where the iteration pattern is defined as an identity function (the same number of row jumps than column ones), finely detected by DESK-A as well.

Non-regular patterns Unfortunately, it is not always possible to create an iteration pattern that best fits a sequence started by the user. Actually, when the system is not able to find out linear relationships in iterative sequences on the widget's geometry then had-hoc or specific-purpose iteration patterns have to be considered.

The system faces the challenge of non-regular patterns by enabling the hand-coded creation of a pool of pre-defined iteration patterns. Those can be defined by experts. Therefore, it is able to customize the design and tell the system how to resolve the iteration in order to accomplish the transformation successfully. The pool of non-regular patterns can be included in the engine configuration, specifying the behavior for how DESK-A has to deal with each type of widget.

Figure 11 shows an example of two iteration patterns that can be defined in the non-regular part of the DESK-A configuration file. This example reflects non-regular patterns where linear relationships are hard to find out, since there is not a straight relationship among the widget's attributes (column and row insertion sequences), so that IP Algorithms cannot be applied directly. In any case, non-regular patterns cannot be considered commonplace. Actually, they are rather difficult to find in common practice, so that a customized pool of predefined patterns is used in order for the system to tackle such a kind of patterns.

3.4.2 DESK-A

As shown in Figure 12, DESK-A (DESK-Agent) is a specialized inference assistant for finding out high-level tasks related to user actions.

While DESK-A is based on Information Agents [1] and Wrappers [14, 27], by contrast it searches the monitoring model, which has an explicit semantic representation of the user's actions, rather than searching the HTML code directly. Therefore DESK-A is able to activate more complex heuristics in order to find out transformation of presentation widgets, such as transforming a combo box into a table or transforming a table into a selection list. DESK-A can also infer more complex intentions automatically, such as sorting a selection list and copying attributes from one table cell into another.

DESK-Agent detects and manages both regular and non-regular patterns by monitoring the user input. Basically, DESK-Agent comprises three main states:

- **Pre-activation**: where the agent checks the monitoring model for high level tasks. This depends on the configuration set.
- **Activation**: where the agent searches for specific widget values in the monitoring model once it is pre-activated. Here, DESK-A analyzes in-depth the history of user actions and creates different models for each widget involved in the interaction.
- **Execution**: where the agent executes the transformations taking into account the values found at the activation step.

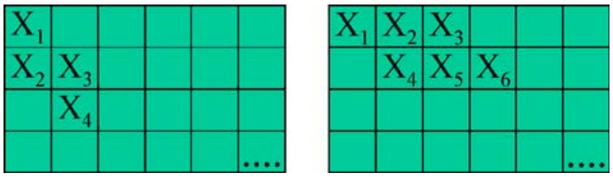


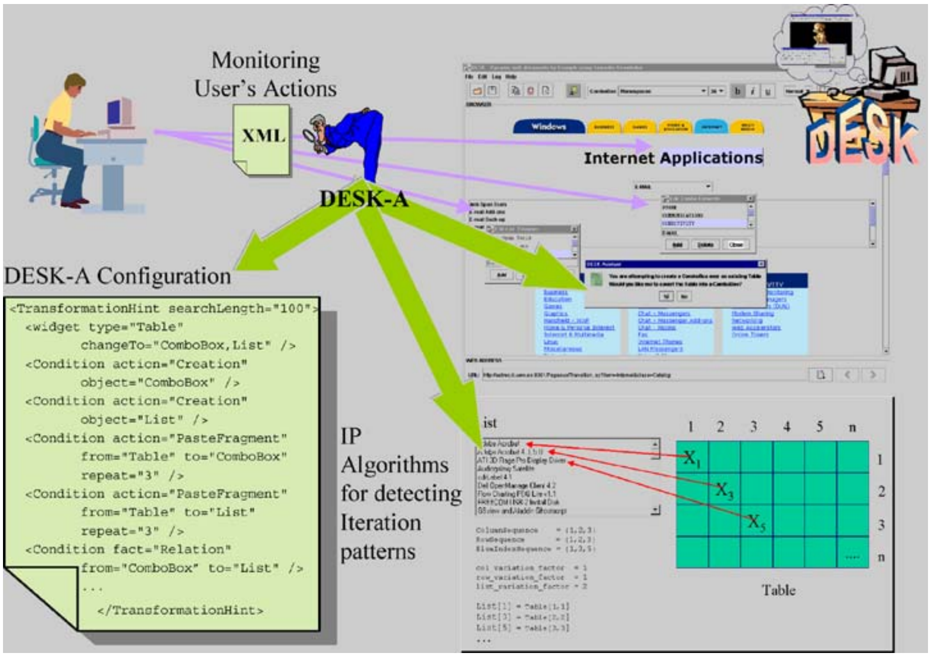
Figure 11 Two examples of non-regular iteration patterns detected while copying elements from a selection list into a table.

DESK-Agent searches the monitoring model for primitives that better fit the requirements defined at its configuration. The agent can be set-up by defining a configuration file on the client-side. That configuration reflects the agent's behavior:

```
<TransformationHint>
  ...
  <widget type="List" changeTo="Table">
    <Condition action="Creation"
               widget="Table"  />
    <Condition action="PasteFragment"
               from="Table" to="List" />
    <Non_Regular_Pattern_Pool>
      <Pattern  col_sequence="1,1,2,2"
                row_sequence="1,2,2,3"
                elem_sequence="1,2,3,4">
        <Resolve i="from 1 to List.getSize(); i++1"
                  next_col_sequence="col[i],col[i]"
                next_row_sequence="row[i],row[i+1]  "
                next_elm_sequence="elm[i]" />
      </Pattern>
      <Pattern col_sequence="1,2,3,2,3,4"
               row_sequence="1,1,1,2,2,2"
                 elm_sequence="1,2,3,4,5,6">
        <Resolve
                next_col_sequence="3,4,5,4,5,6,..."
                next_row_sequence="3,3,3,4,4,4,..."
                next_elm_sequence="7,8,9,10,11,..." />
      </Pattern>
      ...
    </Non_Regular_Pattern_Pool>
  </widget>
  ...
  </TransformationHint>
```

The above code is a fragment of the DESK-A configuration, where <TransformationHint> elements are pre-activation directives that the agent will check for arranging transformations between both widgets (<widget>), in this case a selection list (type="List") and a table (changeTo="Table"). Furthermore, DESK-A checks the creation status (action="Creation") of the table, as reflected in <Condition> elements, and analyses the copy sequence of elements (action="PasteFragment") from the table into the selection list, identifying dependences between the two widgets. When all these prerequisites are satisfied, the agent executes transformation heuristics for detecting iteration patterns (IP Algorithms) by selecting meaningful information from the monitoring model. Finally, the process results in transforming the widgets and keeping the same structure that holds the source widget by firstly asking the user for confirmation.

DESK-Agent also deals with non-regular patterns by supporting the creation of a pool of pre-defined iteration patterns (<Pattern> element inside <Non_Regular_Pattern_Pool>, in



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

```
ColumnSequence    = (1,2,3)
RowSequence       = (1,2,3)
ElemIndexSequence = (1,3,5)

col_variation_factor  = 1
row_variation_factor  = 1
list_variation_factor = 2

List[1] = Table[1,1]
List[3] = Table[2,2]
List[5] = Table[3,3]
...
```

Figure 12 DESK-A overview.

the agent configuration code). This way, DESK-A completes and resolves (<Resolve> element) the iteration sequence in order to accomplish the transformation successfully. Non-regular patterns are represented by using an indexed-construction, defining a for-like loop to iterate through columns, rows and selection list items (<Resolve i="from 1 to List. getSize(); i++1"). Furthermore, DESK-A supports a numerical representation of iteration sets (<Resolve next_col_sequence = "3,4,5,4,5,6,… ") for column, row and item indexes. This kind of specification becomes far more natural and easy-to-understand for non-expert users.

In short, DESK-A provides an intelligent assistant to help end-users carry out different, sometimes difficult to achieve, kind of actions while editing web pages. The configuration (file) of the assistant can be customized by experts, in order for DESK-A to act according to the end-user's needs. This mechanism can be extended for increasing productivity in user interaction by means of providing non-expert users with continuous assistance in their daily solving activities with computer applications as well as generating programming code without the necessity of learning programming or specification languages. This challenge can be carried out by exploiting the monitoring and semantic detection strategies. The main goal is to assist the user in a great deal of different scenarios, such as classical interface builders and toolkits, authoring tools for generating model-based user interfaces and, in general terms, programming environments. To this purpose, the abstract mechanism of pattern detection can be extended and new IP Algorithms can be created, in order for other kind of user intentions to be detected by the system regardless of the domain and the interface used. To corroborate this hypothesis, an experiment with real users has been carried out. That experiment helped to evaluate the accuracy of the intelligent heuristics implemented and the system's response to different user modifications.

4 Evaluation and discussion

The main goal of the work presented is to provide easy-to-use mechanisms for personalizing dynamic web pages. To achieve this, a methodological 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, this approach protects the user from having to use programming languages when authoring dynamic web pages. To carry out such a challenge, the system features an intelligent reverse-engineering mechanism that helps end-users carry out modifications in a WYSIWYG environment. Therefore the system accomplishes the changes by automatically modifying the underlying models on the server, thus providing end-users with a new web page with minimal effort.

An initial experiment has been carried out in order to evaluate and assess the quality of the approach presented here. Next sections reports on an experiment carried out with real users that has helped evaluate the authoring mechanisms supported by DESK. Additionally, a discussion will be provided in order to analyze DESK's functionality.

4.1 User experiment

For the study, we recruited 12 participants with 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 skills in web programming, but to have a basic ability to handle web navigation. Post-study interviews revealed that only 40% of 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, which were the only skills required to carry out our experiment.

In general, the main objectives of this experiment were 1) measure DESK's hit rate in inferring user intentions from their actions monitored throughout the experiment and 2) observe each user's perceived predictability and the ease of use in web page authoring with DESK. In order to fulfill those objectives, a study was designed, consisting of asking users to use DESK for authoring a dynamic web page containing a great variety of content and presentation elements. The changes suggested to users consisted of replacing text, transforming widgets (such as a bullet list into a table), adding new elements to the table created, modifying text attributes (color, justification and so on), inserting new text, removing existing text and moving HTML objects. The task was then to modify the given page (i.e., carrying out changes to page elements independently of the order) to obtain a personalized final version with all the changes applied. The lack of a specified order in which the modifications can 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. Consequently, different users could carry out the modifications by following different steps and thereby we expect the system to respond in different ways. The main objective of this was to get the maximum information about the operation of the system's inference mechanisms. The variety of modifications proposed helped us observe different aspects of the system's behavior and inferences made, such as:

– How the system identifies 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 automatically presentation elements (e.g., a bullet list into a table), 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 with 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 transformation, identifying presentation structures and allowing 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.

This experience revealed interesting aspects from both DESK and the user's 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 intentions.).

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

Table 1 shows the numerical values obtained by the experiment. 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 *maximum* and *minimum* 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 observed depends on the step 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' intentions. This implies that DESK achieves most changes successfully when carrying out the reverse-path analysis. Any errors were mainly due to ambiguities when inferring user intentions [17] 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,

Table 1 Total number of primitives, hit rate and time measured.

	No. of primitives	Hit rate	Time consumed
Maximum	281	98%	7 min
Minimum	155	90%	4 min, 11 s
Mean	200	95%	5 min, 39 s
Standard Deviation	34	2.9	0.8

since users perceived that DESK is similar to other static web tools but includes powerful mechanisms to modify dynamic web pages automatically.

Additionally, two different questionnaires were used to evaluate human reactions to the interaction with DESK, covering the topics of satisfaction, ease of use and user expectations. Users were asked to fill out a questionnaire based on User Interface Satisfaction [7] and another based on Perceived Usefulness and Ease of Use [11]. 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.

The evaluation of the questionnaires also revealed interesting conclusions concerning ease of use and predictability of the authoring tool. The predictability is a value ranged between 0 (minimum) and 5 (maximum) 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. The predictability's result obtained through the experiment was: $Maximum=5$, $Minimum=3$, $Mean=4.2$ and $Deviation=0.6$, which 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.

With respect to the ease of use, results obtained support the initial hypothesis. All users (100%) 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.

4.2 Related work

This research aims at providing a set of PBE (Programming by Example) techniques intended for authoring domain-independent web-based user interfaces and dealing with high-level user tasks. Different domains have been considered in order to validate the tool. From this point of view, DESK is comparable to other approaches such as *Predictive Interfaces* [10] and *Learning Information Agents* [1], where the system observes and monitors the user interaction with the software environment. These approaches help the user by predicting and suggesting some commands to carry out tasks automatically.

One of the main limitations of early PBE systems that monitor user's actions [9] is that they are too literal. Some of these systems replay a sequence of actions at the keystroke and mouse-click level, without taking any account of context or attempting any kind of generalization. By contrast, later works are based on recording user actions at higher level of abstraction and making explicit attempts to generalize them. However, such systems have been demonstrated only in special, non-standard, often tailor-made software environments [15].

Eager (described in [9]) is one of the most famous PBE attempts to bring together PBE and Predictive Interfaces. Eager is a Macintosh-based assistant that detects consecutive occurrences of a repetitive task and proposes the user complete the loop automatically. The loop is inferred by observing the user's actions. Eager needs the user to complete two consecutive tasks. This becomes a limitation since occurrences do not have to appear consecutively. Familiar [31] overcomes some Eager's limitations but it does not address the previous mentioned problem either. Other works, like APE and SMARTEdit (both described in [15]) attempt to solve this difficulty by using *machine-learning* mechanisms in order to learn efficiently and rapidly when to make a suggestion and which sequence of actions to suggest to the user. By contrast, DESK analyses the monitoring model regardless of the number and the sequence of user actions, and it finds meaningful high-level information about the user's intent. DESK operates in real time and does not need to learn about the user's behavior, by using a rule-based approach rather than *machine-learning* algorithms. As well as Familiar, DESK is a domain-independent approach, but in DESK the domain information is used in order to enhance the inference process.

Mondrian, a Lieberman's earlier work described in [9], was based on AppleScript to monitor the user and control applications. However, Mondrian does not provide domain independence and high-level application knowledge either. Similarly, in TELS [25] the system takes into account the user's actions, inferring iteration patterns for addressing loops and conditions. TELS enables the end-user to meet the inference process, by asking for her/him opinion. In DESK, the system avoids the user from having to make assumptions about the inference mechanisms and so the PBE-based inference process becomes totally transparent. However, in order to solve only ambiguous situations, the system asks the user for help.

The use of data models was already present in PBE systems like Peridot [27] and HandsOn [6]. In a very simple form, Peridot enables the user to create a list of sample data to construct lists of user interface widgets. The data model in Peridot consists of lists of primitive data types. 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 at run-time. Our view in this regard is that it is interesting to lift these restrictions and support richer information structures. To this end, DESK uses ontology-based domain information for further user-intent characterization.

Concerning EUD (End-User Development) related work, there have been interesting contributions during the last years. WebRevenge [29] can track the reverse path of a web page. WebRevenge generates a CCTT task model [30] by analyzing the interaction and the interface's elements. WebRevenge works together with TERESA [26], an abstract 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 to web-based migration of applications to different platforms. By contrast, DESK is intended to help the user during the interaction with the system rather than when using it as a multi-modal generation system. DESK also takes into account both user interaction and an ontological data model is used during the interaction to improve the inference process. DESK uses a low-level task model rather than a CCTT-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.

Another interesting work also closely related to EUD paradigm is LAPIS [24]. LAPIS is a web scraper that allows rendering high-level conceptual information 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-user understand web information easily. As well as LAPIS, 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, analyzing also user's actions as part of the characterization process for inferring user intentions.

As for commercial web development tools, probably Microsoft FrontPage and Macromedia Dreamweaver are the most popular ones. These tools offer a high functionality (i.e., a great expressivity and variety of different functions) 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 typical Web users, rather they are intended for Web designers. 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 [33] 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) did not lack functionality but rather ease-of-use. In general, the cognitive load in carrying out editing tasks by using such environments is very high, because these commercial tools are mostly intended for professional designers rather than end-users. Although providing with the highest capability 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 the ease of use, something that is barely visible in existing commercial authoring tools today. In DESK, the goal is to provide easy mechanisms for authoring dynamic adaptive web pages, relieving the user from having to deal with programmatic representations. DESK includes less functionally than commercial tools in favor of increasing the ease of use. The tools presented here 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.

5 Conclusions

Most of tools and technologies targeted at authoring the 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 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 development of web applications.

Many informal user studies revealed that the web development tool that users envision is typically "Word for Web Apps", expressing a preference for a desktop-based tool that embraces the WIMP, drag-and-drop, and copy-and-paste metaphors, offers wizards, examples and template solutions [34]. The research presented 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 adaptive ontology-driven web applications.

Our main contribution is focused on DESK, which provides the designer with an intuitive authoring environment capable of addressing complex web page designs. The authoring tool presented 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. While DESK is focused on making changes to presentation objects, PEGASUS's domain ontologies are fixed and can only be modified (i.e., insert, add or remove new classes, attributes, relations and objects) by specific ontology tools such as PERSEUS. However, our contribution is also focused on supplying support to change some static content on the generated presentations. This way, we provided DESK with the ability to change content. This principally means to change the value of some attributes and data fields in the domain objects of the application's domain model. This kind of modification does not affect the ontology classes at all, but only the domain objects created for a concrete presentation. On the other hand, this kind of atomic-content changes could be useful for the end-users, as s/he might desire to change some text (corrections, amendments), translate small pieces of information (into different languages) or simply add some content to customize the presentation accordingly.

Additionally, DESK features a specialized assistant. Namely DESK-Agent detects the user's high-level tasks during the interaction and executes heuristics to achieve transformations on presentation elements with the aim of automating iterative tasks. DESK-A checks up on pre-activation condition and searches the history of user actions for obtaining meaningful information about widget characteristics. This automates a great deal of transformation processes and provides the user with assistance to complete iterative tasks on him or her behalf.

To test assumptions about our approach's ease of use, we have carried out an initial 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. The outcome of the experiment revealed a high satisfaction rate of users with respect to the tool. This was due to the similarity that the users perceive with respect to ordinary web editing and browsing tools, although by contrast the proposed system includes some add-on mechanisms that allow for editing dynamic web pages and assisting the user in accomplishing cumbersome tasks.

All in all, the main goal of the work proposed was 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, this work makes minimal assumptions about user skills in web-based languages, supplying with a EUD solution that involves an automatic process of reverse engineering intended to extract user intent and reduce interaction efforts.

The presented work is based on well-known disciplines such as Programming By Example and Model-Based User Interfaces paradigms. In this sense, PBE and MBUI techniques can be combined together in order to relieve the user from having to deal with web-based languages and complex non-end-user-intended development environments. Certainly, this implies to reduce sometimes the expressiveness of MBUI approach, since users do not need to manipulate declarative specifications of the interface, but to dedicate all their effort to easily carry out their expectation in software customization [22]. In general terms, the user should not be aware of the interface's internal specification processes. This led 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.

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

References

1. Bauer, M., Dengler, D., Paul, G.: Instructible information agents for web mining. In: Proceedings of the International Conference on Intelligent User Interfaces, New Orleans, USA, pp. 21–28 (2000)
2. Boehm, B.W., Clark, B., Horowitz, E., Westland, C., Madachy, R., Selby, R.: Cost models for future software life cycle processes: COCOMO 2.0. In: Arthur, J.D., Henry, S.M. (eds.) Annals of Software Engineering Special Issue on Software Process and Product Measurement. Baltzer AG Science Publishers, Amsterdam, The Netherlands (1995)
3. Brusilovsky, P., Eklund, J., Schwarz, E.: Web-based education for all: a tool for the development of adaptive courseware. Comput. Netw. ISDN Syst. **30**, 1–7 (1998)
4. Castells, P., Macías, J.A.: An adaptive hypermedia presentation modeling system for custom knowledge representations. Proceedings of WebNet-World Conference on the WWW and Internet. Orlando, Florida, pp. 148–153 (2001)
5. Castells, P., Macías, J.A.: Context-sensitive user interface support for ontology-based web applications. Poster Session of the 1st. International Semantic Web Conference, Sardinia, Italia (2002)
6. Castells, P., Szekely, P.: Presentation models by example. In: Duke, D.J., Puerta, A. (eds.) Design, Specification and Verification of Interactive Systems, pp. 100–116. Springer-Verlag, New York (1999)
7. Chin, J.P., Diehl, V.A., Norman, K.L.: 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 (1988)
8. Communications of the ACM. Special Issue on End-User Development. September, Volume 47, Number 9, 2004
9. Cypher, A.: In: Watch What I Do: Programming by Demonstration. The MIT Press, USA (1993)
10. Darragh, J.J., Written, I.H.: Adaptive predictive text generation and the reactive keyboard. Interact. Comput. **1**, 27–50 (1991)
11. Davis, F.D.: Perceived usefulness, perceived ease of use, and user acceptance of information technology. Manage. Inf. Syst. Q. **3**, 319–340 (1989)
12. Dean, M., Connolly, D., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL web ontology language 1.0 reference" W3C Working Draft 29 July. Available at http://www.w3.org/TR/owl-ref (2002)
13. EUD-NET: Network of excellence on end-user development. http://giove.cnuce.cnr.it/EUD-NET
14. Klann, M.: End-user development roadmap. In: Proceedings of the End User Development Workshop at CHI Conference, Florida, USA (2003)
15. Lieberman, H.: In: Your Wish is My Command. Programming by Example. Morgan Kaufmann Publishers, USA (2001)
16. Lieberman, H., Paternò, F., Wulf, V.: In: End-user Development. Human-Computer Interaction Series. Springer Verlag, New York (2006)
17. Macías, J.A. (ed.): 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 (2003)

18. Macías, J.A., Castells, P.: 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 (2003)
19. Macias, J.A., Castells, P.: Using domain models for data characterization in PBE. In: Proceedings of the End User Development Workshop at CHI Conference, Ft. Lauderdale, Florida, USA (2003)
20. Macías, J.A., Castells, P.: An EUD approach for making MBUI practical. In: Trætteberg, H., Molina, P.J., Nunes, N.J. (eds.) Proceedings of the First International Workshop on Making model-based user interface design practical: usable and open methods and tools. Funchal, Madeira, Portugal (2004)
21. Macías, J.A., Castells, P.: Finding iteraction patterns in dynamic web page authoring. Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction Hamburg, Germany, pp 164–178 (2005)
22. Macías, J.A., Paternò, F.: Customization of web applications through an intelligent environment exploiting logical interface descriptions. Interacting with Computers—The Interdisciplinary Journal of Human–Computer Interaction **20**, (1), 29–47 (2008)
23. McLean, A., Carter, K., Lövstrand, L., Moran, T.: User-tailorable systems: pressing issues with buttons. ACM Proceedings of CHI, pp. 175–182 (1990)
24. Miller, R.C.: End user programming for web users. In: Proceedings of the End User Development Workshop at CHI Conference, Ft. Lauderdale, Florida, USA (2003)
25. Mo, D.H., Witten, I.H.: Learning text editing tasks from examples: a procedural approach. Behav. Inf. Technol. **1**, 32–45 (1992)
26. Mori, G., Paternò, F., Santoro, C.: CTTE: support for developing and analysing task models for interactive system design. IEEE Trans. Softw. Eng. **8**, 797–813 (2002)
27. Myers, B.A.: In: Creating User Interfaces by Demonstration. Academic Press, San Diego (1998)
28. Murray, T.: Authoring knowledge based tutors: tools for content, instructional strategy, student model, and interface design. J. Learn. Sci. **7**, (1), 5–64 (1998)
29. Paganelli, L., Paternò, F.: Automatic reconstruction of the underlying interaction design of web applications. Proceedings of the SEKE Conference, pp. 439–445 (2002)
30. Paternò, F.: In: Model-Based Design and Evaluation of Interactive Applications. Springer Verlag, New York (2001)
31. Paynter, G.W., Witten, I.H.: Automating iteration with programming by demonstration: learning the user's task. Proccedings of the IJCAIWorkshop on Learning about Users. Stockholm, Sweden (1999)
32. Puerta, A.R., Eisenstein, J.: Towards a general computational framework for model-based development systems. Proceedings of the International Conference on Intelligent User Interfaces (IUI). ACM Press, New York (1999)
33. Rode, J., Rosson, M.B.: 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 (2003)
34. Rode, J., Rosson, M.B., Pérez, M.A.: End-user development of web applications. In: Lieberman, H., Paternò, H.F., Wulf, V. (eds.) End-User Development. Human Computer Interaction Series. Springer Verlag, New York (2006)
35. Sahuguet, A., Azavant, F.: Building Intelligent Web Applications Using Lightweight Wrappers. Data and Knowledge Engineering (2000)
36. Shneiderman, B.: Leonardo's Laptop. The MIT Press, USA (2003)