A Heuristics-Based Approach to Reverse Engineering of Electronic Services

C. Vassilakis[1], G. Lepouras[1], A. Katifori[2]
[1] Department of Computer Science and Technology, University of Peloponnese, Terma Karaiskaki 22100, Tripoli, Greece
{costas, gl}@uop.gr
[2] Department of Informatics and Telecommunications, University of Athens, Panepistimiopolis, 15784, Athens, Greece
vivi@di.uoa.gr

Abstract: Since the beginning of the electronic era, public administrations and enterprises have been developing services, through which citizens, businesses and customers can conduct their transactions with the offering entity. Each electronic service contains a substantial amount of knowledge in the form help texts, rules of use or legislation excerpts, examples, validation checks etc. This knowledge has been extracted from domain experts when the services were developed, especially in the phases of analysis and design and was subsequently translated into software. In the latter format though, knowledge cannot be readily used in organizational processes, such as knowledge sharing and development of new services. In this paper, we present an approach for reverse engineering electronic services, in order to create knowledge items of high levels of abstraction, which can be used in knowledge sharing environments as well as in service development platforms. The proposed approach has been implemented and configured to generate artifacts for the SmartGov service development platform. Finally, an evaluation of the proposed approach is presented to assess its efficiency regarding various aspects of the reverse engineering process.

Keywords: e-government; electronic services; reverse engineering; organizational knowledge

1 Introduction

In the past few years, governments are systematically working on realizing e-government policies and frameworks, which include the delivery of electronic services for enterprises and citizens. Citizens and enterprises expect that the provision of a rich spectrum of electronic services will result in a number of benefits, as reported in [1]. Efforts in the area of electronic services development have particularly focused on the implementation of the 20 public services that are listed in [2] as being "the first steps towards Electronic Government". Similar remarks hold for the e-commerce domain [3].

In this context, the development of an electronic service is usually treated as an isolated software project, thus information extracted from the involved domain experts in the requirements analysis phase is recorded as low level "user requirements", rather than as high-level organizational knowledge [4]. This practice leads however to suboptimal results for the organization since:

1. the "software specifications" format is inappropriate for knowledge sharing among the organization's employees, since it usually contains technical, low-level details and is mainly structured for the convenience of software

designers and implementers. Notable employee groups that could benefit from the knowledge offered within the analysis phase include other domain experts, who seek information on the specific subject and help desk workers, who could use this knowledge to provide information and guidance to users of the electronic service.

2. the knowledge, in its original form, could be used to tackle the "lack of expert assistance" usage barrier for electronic services identified in [5], according to which users refrain from using electronic services because no professional help is available. Since the knowledge, as offered by domain experts, includes a number of examples, explanations, descriptive texts, related legislation and so forth, it could well serve as assistance for service end-users.
3. the software specifications produced for an electronic service are usually considered as pertinent to the specific electronic service only and/or are not readily available for other projects. This leads to reduced opportunities for reusing the knowledge amassed within the analysis phase in the context of developing other services (e.g. re-using the personal details portion of a form, or the algorithms to check the validity of a VAT number).

In order to tackle these deficiencies, organizations are adopting knowledge management platforms, to promote recording of knowledge in explicit format facilitating thus searching, browsing and sharing, as well as electronic service development platforms, which can promote component reusability across services, at least to some extent. For services that have already been developed, however, the original knowledge has already been mapped to software specifications and software artifacts (HTML forms, JavaScript and/or Java code, database schemata etc), thus these services must either remain as "isolated islands" in the domain of the organization's electronic services or they must be remodeled in the chosen platform (knowledge management or electronic service development), increasing thus their overall development effort and cost.

In this paper we present a method for reverse engineering software components of already developed electronic services, and using the individual elements identified in the reverse engineering process to synthesize artifacts of higher levels of abstraction. These artifacts encompass aspects useful both for knowledge management and electronic service development, being thus suitable both for knowledge sharing and dissemination within the organization, as well as for developing new services, while at the same time facilitating the integration of these two organizational tasks. Once these artifacts have been formulated, they can be imported into knowledge management systems and/or service development platforms, after appropriate formatting has been applied. Bundling of knowledge management and service development aspects into a single artifact is considered important, since it allows for more efficient handling of issues spanning across these aspects. Two of these important issues are tackling the "lack of expert assistance" usage barrier identified above (since the knowledge required for offering assistance to service users must *both* be managed *and* included in the electronic service) and performing service maintenance after legislation updates (legislation is a special form of knowledge, and should be correlated with the software components that depend on it, in order to enable their direct tracing).

The method we present has been applied to produce artifacts suitable for importing into the SmartGov platform [6] [7]. The SmartGov platform is a knowledge-based development environment for public sector online services, which adopts a holistic view for knowledge management and electronic service development. The SmartGov

platform offers tools and services for authoring knowledge and electronic service components in a high level of abstraction, and establishing links between them to form a semantically rich network of information which can be subsequently exploited for knowledge retrieval and sharing. Finally, the SmartGov platform includes the Integrator component, which automatically compiles the high-level descriptions of electronic service components into executable service images.

The rest of the paper is organized as follows: section 2 presents related work in the areas of electronic service development platforms and reverse engineering of web applications. Section 3 outlines the basic architecture and benefits of the SmartGov platform, while section 4 elaborates on the reverse engineering process, presenting the methodology and techniques used to transform web pages code into SmartGov platform objects of high levels of abstraction. Section 5 presents an evaluation of the reverse engineering approach and, finally, section 6 concludes the paper and outlines future work.

2 Related work

Reverse engineering is a process of examination (as opposed to alteration) [8], directly supporting the essence of program understanding: identifying artifacts, discovering relationships, and generating abstractions. Reverse engineering methods and techniques are used for three canonical activities, namely data gathering, knowledge management and information exploration [9]. The activity of knowledge management in particular, refers to capturing, organizing, understanding, and extending past experiences, processes, and individual know-how. In this context, the reverse engineering process produces artifacts that, if properly managed, could be shared at development team, department or even organizational level, serving thus as an active repository of corporate knowledge [10]. In order to improve their effectiveness, knowledge repositories employ abstraction mechanisms as organizational axes for structuring the knowledge base. Abstraction mechanisms allow for focusing on high-level aspects of entities, screening unimportant (for structuring purposes) details. The most common abstraction mechanisms used in these contexts are classification, aggregation, and generalization [11]: classification recognizes generic objects (classes) and instances of them, aggregation groups individual objects into a single object of higher level of abstraction and generalization organizes classes into hierarchies under the “is-a” relationship. Regarding the application of software reverse engineering techniques on web applications, notable activities reported to date include [12], which aims at the construction of UML diagrams so as to support the maintenance and evolution of web applications (an extended version of this work is reported in [13]); the main focus of this work is the linking and interaction between web pages, not the artifacts present therein. Another work targeting page linking is [14], which describes a method for extracting task models from web pages, aiming to reconstruct the underlying logical interaction design. Finally, RetroWeb [15] aims at providing a description of the informative content of the site at various abstraction levels: physical, logical and conceptual. The naming process in this procedure is elaborated on in [16]. Retroweb targets informational web sites and does not address reverse engineering of forms and services.

Recording of knowledge in an explicit and reusable format is a topic traditionally addressed by knowledge management tools. A number of such tools have been already made commercially available (e.g. [17]; [18]; [19]; [20]; [21]) while research

efforts are also addressing various aspects of organizational knowledge management (e.g. [22]; [23]; [24]).
Finally, in terms of electronic service development platforms, traditional "isolated project" development approaches are still dominant. For example, the BEA WebLogic Workshop is targeted for IT staff only, facilitating building of web applications, web services, JSPs, Portals, EJBs, and Process Workflows that are optimized for a service-oriented architecture [25]; similarly Adobe's Government forms [26] and LiveCycle Designer [27] emphasize on form appearance, XML representation and other usability and back-end processing aspects, without any support for knowledge management in the development process.

3 The SmartGov platform

The SmartGov platform offers functionality for managing knowledge and validation rules, creating objects, designing forms and services and deploying them. The central concept in the SmartGov platform is that of Transaction Service Elements (TSEs). TSEs are in fact widgets, which can be used as building blocks for electronic services. Contrary to user interface widgets, TSEs are not limited to visual appearance only: they can contain metadata and domain knowledge. Metadata may include the object's type, range of allowable values, validation checks, labels, and on-line help, while domain knowledge includes information about the relation of the object to other elements, legislation information, documentation etc. Other concepts in the SmartGov platform are TSE groups (assemblies of individual TSEs which can be managed collectively), forms (canvases on which TSEs and TSE groups are placed) and transaction services (TSs – collections of forms offering a specific service). Similarly to TSEs, instances of these concepts can contain metadata and domain knowledge. Metadata elements in these concepts vary according to the concept type, e.g. metadata for a transaction service include the authentication method and whether modification of submitted documents is allowed, while metadata for a form may include actions to be taken upon form loading, specification of the visual layout of the form, validation checks involving multiple form fields and so forth. The SmartGov platform offers additionally functionalities for establishing links between instances of the modeling concepts (TSEs, groups, forms and transaction services), formulating thus a semantically rich network of elements, which can be browsed or queried by platform users. The overall architecture of the SmartGov platform is illustrated in Figure 1.

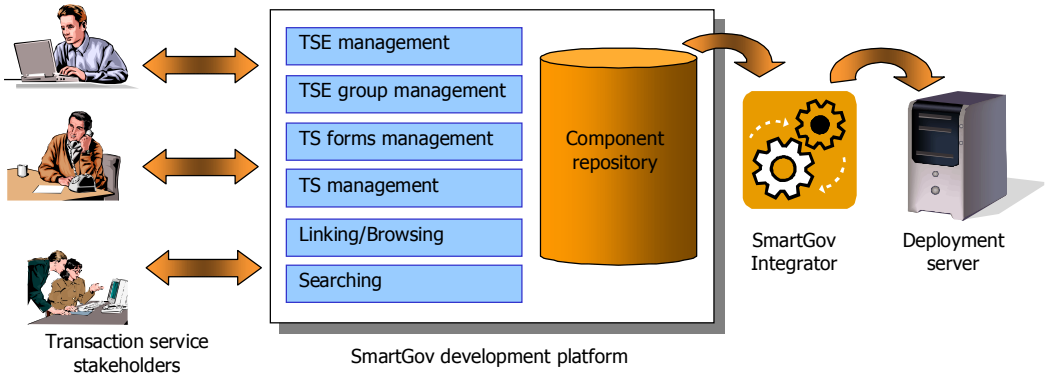

Figure 1 - SmartGov platform architecture

Once all elements for an electronic service have been defined, it is possible to create an executable image of the electronic service using the SmartGov Integrator module.

The SmartGov integrator accesses the component repository to extract pertinent definitions, and creates all necessary files for service deployment. In more detail, the Integrator creates forms for collecting user input, HTML pages for presenting help, examples or other documentation to service users, Java, and possibly JavaScript, code for enforcing validation rules (all validation rules are conducted at the back-end, while certain checks [e.g. date validation, number format checking etc] can be additionally performed at the front-end to promote interactivity). The Integrator also supplies default facilities for user authentication and document storage and retrieval. Finally, the integrator bundles all generated objects into a web archive (WAR file) and deploys it to a web server running a JSP container (the *deployment server*), through which the service is made available to the public.
For more information on the SmartGov platform, the reader is referred to [6], [7] and [28].

4 Reverse Engineering Electronic Services

In this section we present the rules employed for electronic artifact identification, composition and abstraction. The aim of this reverse engineering approach is to formulate semantically rich artifacts (TSEs, TSE groups, forms and TSs), with each one of them encompassing visual characteristics, knowledge (help texts, examples, etc), business rules (validation checks) and relations with other elements. The heuristics for combining individual HTML form elements into electronic service artifacts exploit the structure and nesting of HTML tags, naming conventions and element proximity, both in terms of code (e.g. tags are located nearby within the HTML file) and in terms of visual layout (e.g. horizontally or vertically adjacent cells within a table). In the rest of this section, we first present the common artifact patterns as they appear in transactional services; these patterns dictate the operation of the heuristics for artifact identification, composition and abstraction, which is discussed subsequently.

4.1 Artifact patterns in electronic services

When electronic service designers create the HTML pages that comprise a service, they attempt to arrange elements in ways that are meaningful and usable for prospective service users. In order to extract patterns for these arrangements, more than 50 online services from different countries were examined. The services used in this analysis were drawn from well-established government portals for online services, including the US portal (http://www.firstgov.gov/Citizen/Services.shtml), the UK online service directory (http://www.direct.gov.uk/Diol1/DoItOnline/) and the Singapore e-government services for citizens catalogue (http://www.ecitizen.gov.sg/); other sources were considered as well, including the Cyprus Ministry of Finance [http://www.mof.gov.cy] and the on-line taxation portal of Greece [http://www.taxisnet.gr]. The layout of printed forms has also been considered, since electronic services have been found to mimic the appearance of their paper-based counterparts, trying to provide their users with a familiar image and exploit the users' experience drawn from using the traditional service delivery channels. The printed forms were retrieved from the same sources as the online services, while samples and directives made available by the EU (e.g. VIES acquisition and INTRASTAT forms [http://www.revenue.ie/pdf/v-int.pdf]) were taken into account as well.
A common layout for service elements is the one depicted in Figure 2: In this layout we can identify the following areas:

- a *form header*, including the agency logo, links to the agency's home page and generic help, as well as a graphic acting as a separator
- the *form main body*, which includes short introduction of the form, the actual input elements (grouped here in four areas) and their explanations, and
- a *form footer*, including navigational controls (*Continue* button) and a service-specific help link.

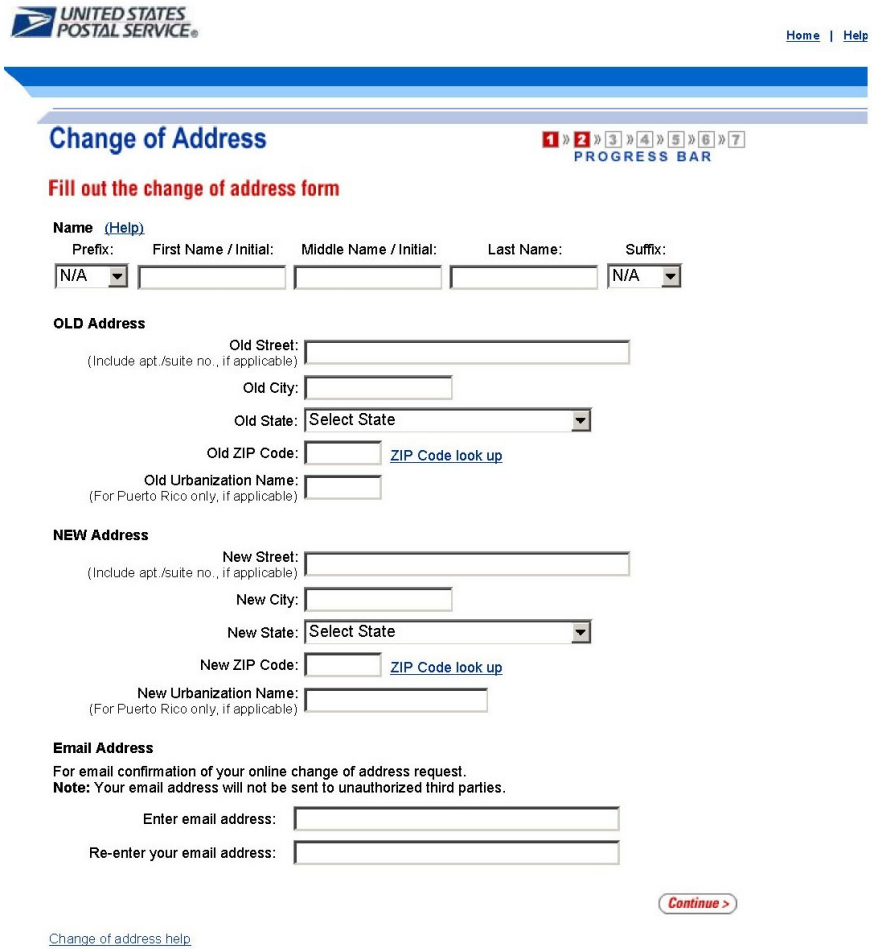

Figure 2 – A common layout for service elements

The input elements, in particular, have been organized into groups, with each group having a header (e.g. Name, OLD address). A help hyperlink (for the "Name" group) and/or some text for the group as a whole (e.g. in the "Email Address" group) may also be present. Within a single group, input elements may be laid out:

- horizontally, with their descriptions being placed above (or below) them, as is the case with the "Name" group.
- vertically, with their descriptions being placed on the left hand side of the element (more rarely, on the right)

In both layouts, help texts and additional help or utility links for individual input elements may be present, which may be placed beside the field description or the input element. Examples of help texts and additional help links are illustrated in Figure 3, while an example of a utility link is the "Zip Code lookup", next to the zip code inputs in Figure 2. In some cases, a particular data item may be collected using more than one input element, as is the case of the date of birth (Figure 3) and the "SSN" data item (Figure 4). Typically, this technique is used for registration numbers (SSN, bank account numbers, license plates etc), as well as dates. In such cases, the

constituent input elements are usually juxtaposed on the layout, with the possible intervening of a separator (dash, slash, space and so forth). Notice that the overall form layouts in Figure 3 and Figure 4 follow the pattern identified for figure 1. An additional commonplace practice that can be identified in Figure 3 and Figure 4 is the use of the asterisk (*) to denote mandatory fields. The asterisk is most usually placed next to the input area or next to the label; positioning next to the label is usually preferred, since (a) service users will read the label, while they may not notice the input area surroundings and (b) radio button-type input widgets, have multiple input areas which are mutually exclusive, thus a "mandatory" flag besides every such input area would be confusing. The same holds for check boxes for which JavaScript code is used to make their use mutually exclusive.

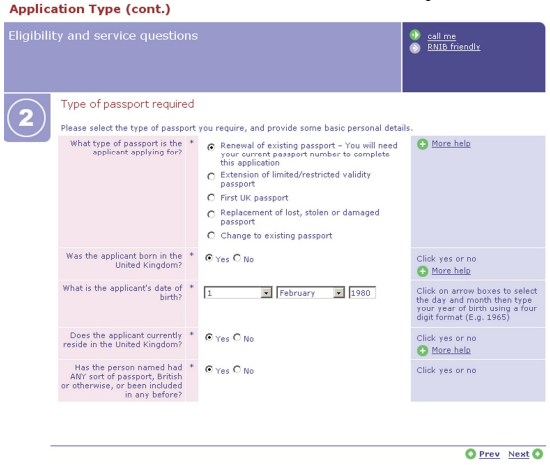

Figure 3 – Help texts and additional help links

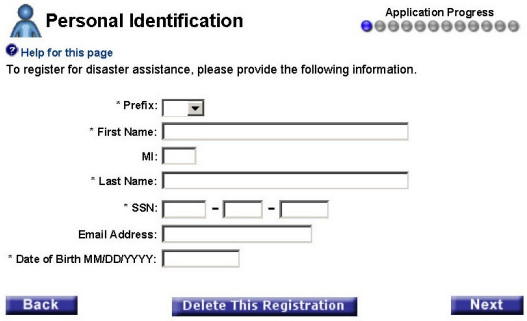

Figure 4 – A data item spanning across more than one input elements

A final case for form elements layout is that of repeating rows. This layout is used by service designers when users need to fill in tables, with each table row describing a specific item, such as a vehicle, a family member and so forth. An example of such a table, taken from an experimental version of the Greek VIES declaration service is depicted in Figure 5 (translated into English). In each row of this table, service users fill in details for their total transactions with an individual or enterprise, providing the country, the VAT registration number of their supplier, and the total value of supplies and triangular supplies. The designers of the form have added buttons allowing service users to add/remove rows to the table, and have included two summary fields giving the sums of the two last columns. The first row of the table hosts the column labels, each one followed by a question mark icon that effectively is a help hyperlink.

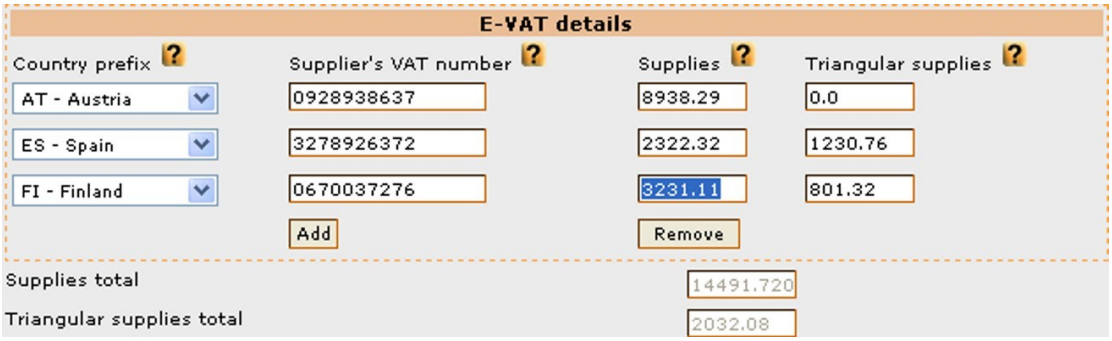

Figure 5 – Repeating row layout

A variant of the repeating row layout of Figure 5 where each row includes a check control and buttons are provided for performing an operation on all checked items is depicted in Figure 6. The check control may appear on either side of the listed elements and, besides this control, the group layout is identical to that of Figure 5. This arrangement has not been encountered in any of the analyzed electronic services but is commonplace in the domain of electronic commerce (e.g. shopping cart management) or webmail software, and has proven to be quite practical for users so it is anticipated that it will be included forthcoming electronic services.

Figure 6 – Repeating rows with control checkbox

4.2 Artifact identification and creation

The phase of artifact identification begins with the specification of the HTML pages that comprise the service. The pages may be read directly from the web server hosting the service, or from a local file system. The latter option is useful for simplifying cases with dynamic page generation and complex requests (e.g. passwords, cookies, secure socket layer etc). In such cases, the user may navigate to the service normally using the browser, save page content through the browser menu and use the saved pages as input to the reverse engineering procedure. Page storing to disk files may also be achieved using specialized tools, such as MetaProducts' Offline Explorer (http://www.metaproducts.com). Reading pages from disk files is a requirement for handling multilingual electronic services (described in section 4) and when complex layout structures should be removed to improve the efficiency of heuristics pertaining to groups (described in section 5). Table 1 summarizes the mode of operation and the required user intervention according to the properties of the reverse engineered service. No other user intervention (including code inspection, human judgment and guidance to the software) will be required during the operation of the reverse engineering software; after the reverse engineering procedure has completed, users may inspect the results through the interface of the SmartGov platform. When pages are read from the disk, a base URL is additionally provided for resolving relative hyperlinks, wherever necessary.

Service properties	Pages are read	User intervention required
Simple or no authentication scheme, no complex layout, no multilinguality	Directly from server	None
Complex authentication scheme, no complex layout	Disk	User navigates to pages and saves them to disk, or uses tool with equivalent functionality
Multilinguality	Disk	User saves pages, creates configuration file
Complex layout	Disk	User saves pages, uses HTML editor to simplify layout

Table 1 – Service properties and mode of operation of the reverse engineering software

After the pages have been read, heuristics that attempt to recognize the patterns described in the previous section within the HTML pages are applied. For each pattern identified, a proper artifact is constructed, consisting of all information pertinent to it; if appropriate, links to other artifacts are also established. Since the patterns mainly pertain to the visual layout of components on the form, HTML tags that control positioning and appearance of elements are analyzed to determine the role of each form component as well as its relation to other components. Tag nesting, JavaScript code associated with HTML page elements and naming conventions are additional sources of information for the reverse engineering process.

Before the application of heuristics commences, the reverse engineering software constructs the *object model* of each page. The object model is a tree-structured representation of the HTML page components (tables, divisions, forms, fields etc), for which programming constructs facilitating traversal and attribute extraction are provided. The HTMLParser[1] package was chosen for this purpose, since it handles satisfactorily pages that do not strictly adhere to the HTML standard (e.g. start tags are not matched with a closing tag, attributes designated as mandatory are not provided and so forth), as is usually the case with "real world" HTML pages.

The heuristics for each type of transaction service component (transaction service element, transaction service element group, form, and transaction service) are presented in the following paragraphs.

4.2.1 Identifying and Creating Transaction Service Element Artifacts

A transaction service element (TSE) in the SmartGov platform is a compound object encompassing the input area and its properties (HTML input type, size, maximum length, initial value), the input area label (usually a description of the area), help texts (commonly provided as hyperlinks or as extended in-place text), the validation rules that apply to the values entered (data type, mandatory input, allowable ranges etc) and, finally, its relationships with other elements.

The first task towards identification of TSEs is locating the input widgets which service users will use to enter the data collected by the service. HTML provides four basic input widgets, namely *INPUT*, *SELECT, TEXTAREA* and *BUTTON* (button functionality can be also provided using the ONCLICK attribute in various element

[1] http://htmlparser.sourceforge.net/

types, including hyperlinks, images etc). For each such construct identified on a page, a respective TSE is created, except for the case of inputs of type *radio*, for which a *single TSE* is created for all *INPUT* instances with the same value for the *name* attribute (similarly, all *OPTION* elements of a *SELECT* construct are packed into a single TSE). The reverse engineering process subsequently locates information for the additional aspects of the TSE as follows:

1. Firstly, the TSE label is determined. The form is initially scanned for a *label* element the *for* tag of which matches the input element name (e.g. `<label for="fname">First Name</label>`), or for a *label* element enclosing the input area definition (e.g. `<label>First Name <input type="text" name="fname"></label>`). If such an element is found, the text specified in the *label* element is extracted and used as the TSE label. If no such label is found, the software attempts to determine the label by its positioning relative to the input area: the TSE label may be placed on the left hand side of the input area (Figure 3, Figure 4 and bottom half of Figure 2), or above the input area (upper half of Figure 2). Note that text layout is usually formatted using tables, thus "left hand side" does not necessarily refer to HTML code immediately preceding the input tag, but may be the text included in the table cell appearing on the left of the field under examination. The reverse engineering software takes into account the case that an extra column, indicating whether the field is mandatory or not, intervenes between the input area and the label field (Figure 3).
2. Afterwards, the help items for the field are located. The help items may be located at the right of the input area, either as directly following HTML code (Figure 2) or within an adjacent table cell (Figure 3). In some cases, only a hyperlink may be present which has to be clicked by the user in order to display the help content. In such cases, the reverse engineering software retrieves the content pointed to by the help anchor, and packs this content within the TSE artifact. The label text (determined in the previous step) is also scanned for presence of hyperlinks, since some electronic services provide access to help by including hyperlinks within the field labels. If such hyperlinks are found, the content pointed to by each hyperlink is extracted and packed with the TSE as a help item. This step may effectively produce multiple help items for a single TSE. Additional help items may be determined from code analysis (described below).
3. The next step is to extract an initial indication whether a TSE is considered mandatory or not. The presence of an asterisk either packed within the label (at its beginning or end – Figure 4) or as a separate table column (Figure 3) is used as such an initial indication. If a label is found to contain such an indication, it is trimmed to drop the asterisk. An additional check to determine whether some input element is mandatory or not is performed in the code analysis phase (described below).
4. Subsequently, the default value for the input area is determined by examining the settings of the HTML attributes associated with the input area. The HTML attributes differ according to the input area type- e.g. the "value" attribute is used for text boxes and buttons, the "checked" attribute applies to check boxes, the "selected" attribute applies to drop down lists, whereas for text areas, the default value is determined by the text included between the starting and ending tags (*<textarea>*, *</textarea>*). The values of the "maxlength",

"size", "rows" and "cols" attributes, whenever present, are also extracted and bundled as properties of the TSE under construction.

5. For input elements with a closed set of values (such as select widgets and radio buttons), the set of values is examined to determine the data type of the input element. If all the values within the set are of the same type (integers, floats, dates, etc), the data type of the TSE under construction is set accordingly; otherwise, the data type is set to "string". Data type inference for input elements with an open set of values (free user type-in) is handled through code analysis (described below).

The TSE properties listed above can be directly determined form attributes values of the input elements or from the placement of text relevant to the input element. However, some important aspects of TSEs, namely their data type, whether a TSE is read-only or not[2], as well as validation checks (minimum and/or maximum value, relationship with other elements and so forth) can not be directly modeled as attribute values; instead, e-service developers use JavaScript to provide these features. In order to determine these features the reverse engineering software analyzes the JavaScript code associated with the input element events. This analysis may also reveal additional help items and supplementary indications whether the TSE is considered mandatory or not. JavaScript code analysis is mainly based on heuristics, rather than rigorous semantic analysis which was considered too exaggerate an approach for addressing the issues at hand, taking also into account that the results of the reverse engineering process will be reviewed by humans before they are used for code generation. The heuristics employed are as follows:

1. if the "onfocus" and "onselect" event handlers of the input element are present and contain code that moves the focus away from the field (typically this is performed using the "this.blur()" method or by moving the focus to another field through the "anotherfield.focus()" method), then the TSE under construction is characterized as "read-only". Note that this is complementary to checking for existence of the "readonly" and "disabled" input element attributes described above, i.e. if either of the checks succeeds, the transaction service element is characterized as "read-only".
2. if the JavaScript code within the page contains instructions that compare the value of the element with the empty string (*element.value = ""* or *element.value.length == 0*) and emit a message if the condition is true (by invoking the *alert* function), then the TSE is considered mandatory. Code patterns that trim the spaces from the element value and compare the result with the empty string (e.g. *trim(element.value) == ""*, *ltrim(rtrim(element.value)) == ""* or the equivalents checking the length of the result) are also taken into account in this check.
3. if the "onfocus", "onselect" and "onmouseover" event handlers of the input element are present and contain code that displays text on the browser status bar (e.g. *onfocus="javascript:window.status = 'Enter the number of children'"*). Many service implementers employ the area occupied by some specific page element to display help items (e.g. *onfocus= "javascript:document.getElementById('helpArea').innerHTML='Enter the*

[2] Some browsers support the "readonly" or "disabled" attributes for specifying read-only attributes; however, since this is not a cross-browser feature, many e-service designers resort to JavaScript code to implement this functionality. The reverse engineering software marks input elements for which these attributes are specified as "read-only".

number of children'"), but these cases have proven extremely tricky to recognize and handle correctly since, for instance, there exist multiple ways to refer to some specific element in a page (e.g. use getElementsByName instead of getElementById, some services employed read-only form elements to display help items thus form element addressing patterns like *document.forms['theform'].elements['helpArea']*.value should be searched for etc). Consequently, it was decided that the reverse engineering software would not handle the latter case of help items.

4. if the "onchange" event handler is present, then the code associated with it is scanned for JavaScript function invocations whose argument list includes only the specific field and possibly constant values (effectively, the argument list shound *not* include references to other form fields). The name of each invoked function is examined to determine whether it is a compound word, whose first component is one of the words "check", "is", "valid", "validate", "verify", while the second component being a data type name or a synonym for it (number, date, integer, float, numeric and so forth) –e.g. `<input name="price" type="text" onchange="checkNumber(this, 'Price should be a number');">`. If a match is found, the data type for the TSE under construction is set accordingly. The whole JavaScript of the page being processed is also scanned for conditions of the form `if (checkNumber(price)...)`, to cater for cases that data type validation of user input is deferred until the submittal of the form, rather than being performed synchronously with data typing; if such code is detected, the TSE data type is again set accordingly.
5. code associated with the "onchange" event handler and that (a) does not reference other fields (b) does not meet the naming criteria of item (4), is recorded as a *validation check* for the TSE. This code may include any custom validation check e.g. value range checks, data format checks and so forth. Conditions of *if* statements anywhere within the JavaScript code of the page (i.e. not necessarily within the field's "onchange" event handler) that reference only the specific TSE (and possibly constant values) are added -together with the associated code block- to the list of validation checks associated with the TSE.

At this stage, all data regarding the TSE have been collected, and the transaction service element is finalized.

4.2.2 Identifying and Creating Transaction Service Element Group Artifacts

The HTML standard provides a construct for specifying groups of fields, namely the *fieldset* tag, which may surround a number of input area specifications. Browsers supporting this feature draw a border surrounding the input areas (Figure 7) to provide a visual clue that these elements are logically associated. The field set may be assigned a label (*Company Info* in the example), using the *legend* tag nested inside the *fieldset* tag. The reverse engineering software identifies such constructs and for each one of them creates a TSE group artifact, which is automatically linked with the individual TSEs it contains. The description of the TSE group is derived from the contents of the *label* element nested within the *fieldset* construct, while extra text occurring within the *fieldset* construct and not directly associated with a specific TSE (e.g. the *Please enter*... phrase in Figure 7) is considered as a detailed description for the TSE group. Hyperlinks occurring within such extra text are considered as help

items for the TSE group as a whole. The TSE group under construction is finalized by adding to it the pertinent validation checks. These are identified as follows:

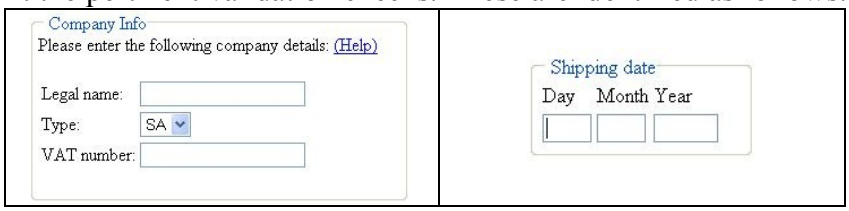

Figure 7 – Rendering of *fieldset* constructs

1. the *onchange* event handler of the TSE elements belonging to the group are scanned for code that involves two or more elements of the TSE group (e.g. `onchange="check_date(day, month, year)"`) but not referencing any field outside the group.
2. the page's JavaScript code is examined for conditions of *if* statements involving two or more members of the TSE group (and possibly constant values), but not referencing any field outside the group. These conditions, together with the associated action blocks, are added to the list of validation checks associated with the TSE group.

At this stage, all data regarding the TSE group have been collected, and the transaction service element is finalized.

The *fieldset* tag is not however the predominant approach for implementing field groups. Most often, implementers employ tables to organize groups of fields since (a) not all browsers support the *fieldset* tag and (b) tables provide more flexibility for laying out titles, borders, fields etc (note that *all* examples in figures 2-4 employ tables for laying out fields). The reverse engineering software deduces field groups using the following procedure:

1. first, the *table segments* are identified. A table segment comprises of a *header* row containing only text and, commonly, spanning across all columns
2. each table containing fields is initially considered to be a single field group. Commonly, the first row of each such table does not contain input fields but the field group description, and may span across all columns (see Figure 3 and Figure 5). The text from such rows is extracted and used to set the value of the relevant TSE group attribute. If the first row does contain an input field, then the text directly preceding the table is used as the field group description (cf. Figure 4).
3. within a single table, if some row contains only text (no input fields) and is followed by rows containing labels and input fields, (cf. Figure 2, rows "OLD Address", "NEW Address", "Email address"), then the text of the first row is used as the group description and the TSEs created from the subsequent rows are the TSE group members.

After deciding whether the members of a TSE group are laid out using tables or consecutive fields, help items and validation checks proceed as described for the case of TSE groups defined using the *fieldset* tag.

The repeating row layout (cf. Figure 5) is another type of TSE group that the reverse engineering software caters for. To detect such groups, the algorithm scans the page for tables (or table segments) with the following properties:

1. the first row contains text labels (*header row*)
2. the header row is followed by two or more rows (*data rows*) containing input elements. The types (input, select, etc) and attributes (e.g. size, maxlength), of the input elements appearing in the same column, must be the same; only the

name attribute is allowed to have a different value, since it typically identifies uniquely each element within a form.

The header row may be (optionally) preceded by a *name row*, which contains only text (the name of the group) and typically contains a single cell, spanning across all columns. Alternatively, the name of the group may be placed right before the table as "normal" text.

When such a table (or table segment) is detected, the reverse engineering software constructs a new TSE group. The name of the group is derived from the *name row*, or the text immediately preceding the table. For each table column, only a single TSE is constructed, whose name is extracted from the header row, while additional properties are determined as described in section 4.2.1. These TSEs constitute the members of the newly created TSE group. A special flag is set for this TSE group to indicate that its elements can be repeated; the number of data rows detected on the form is copied into the *initial rows* property of the TSE group. An additional check made here is whether the first or the last element of a group row is a check control (typically a check box – c.f. Figure 6); in these cases additional information is coupled with the TSE group in question to designate that (a) the repeating group allows its rows to be selected on the basis of the specific check control (b) action buttons "check all", "clear all", "toggle selection" and "remove selected" should be added.

Insofar, no appropriate heuristic has been found to detect the presence of the "Add" and "Remove" buttons (see Figure 5) which insert or remove data rows or the checked group-level action buttons (e.g. "Delete selected" in Figure 6); the reverse engineering software sets appropriate indications in the TSE group that row addition and deletion are allowed, thus when the service is re-generated using the Integrator, such buttons will appear. If, however, some buttons are undesirable (e.g. the number of rows is fixed thus no "Add/Delete" buttons should appear at all, or no "Toggle selection" functionality is desired because it is considered confusing), service designers can simply clear the respective indications using the SmartGov development environment. Similarly, no heuristic has been formulated to detect and appropriately create the summary fields (e.g. "Supplies total" and "Triangular supplies total" in Figure 5); no attempt is made to import such fields, but guidelines are given to service designers on how to create such summary fields within the SmartGov development environment.

4.2.3 Creating Form Artifacts

For each file processed, the reverse engineering software produces one form artifact. The form artifact is linked to the TSE and TSE group artifacts it contains, while the *form header* and *form footer* areas (i.e. HTML code before the first TSE/TSE group and HTML code after the last TSE/TSE group respectively) are used to populate the respective elements of the form artifact. Hyperlinks within the form header and footer are exploited to create help items for the form, as previously described for TSEs. Validation checks involving multiple fields not belonging to the same TSE group are finally added to the form artifact; these are validation checks pertaining to the form as a whole. The same holds for Javascript code associated with the *onsubmit* form event handler.

4.2.4 Creating the Transaction Service Artifact

Finally, for each invocation, the reverse engineering software constructs a single transaction service (TS) artifact. This contains links to the form artifacts comprising the service and each link is complemented with the order that the form appears in the service. Once the transaction service artifact has been created, the created artifacts are

imported into the SmartGov platform, being thus made available to be used for developing other services. After the import procedure has been completed, the reverse engineered service may be re-generated, by invoking the Integrator module of the SmartGov platform.

4.3 Handling multilingual electronic services

Many electronic services nowadays are deployed with multilingual capabilities, so as to address larger portions of their target groups in multi-national or multi-cultural environments. A multilingual electronic service normally has the same layout, input elements and validation checks in all languages, but the language-dependent resources (texts, labels, messages, help items and sometimes images) are naturally different.

The SmartGov platform fully supports development and deployment of multilingual applications, allowing its users to define multilingual resources in all languages that are needed. Multilingual resources can be defined in all artifacts of the SmartGov component repository, i.e. TSEs, TSE groups, forms, transaction services and knowledge units. The executable version of the service produced by the Integrator module automatically detects the user's preferred language, as declared in the browser configuration, and adapts the content to this preference.

The reverse engineering software includes provisions for retrieving the multilingual resources of multilingual applications and importing them into the SmartGov platform. In order to reverse-engineer a multilingual service, the user must first browse the service pages in all languages and save them to disk files. Afterwards, a configuration file must be created, indicating the page number and the language that each disk file corresponds to. Figure 8 depicts an excerpt of such a configuration file.

```
[Page 1]
en=pg1_english.html
fr=pg1_french.html
gr=pg1_greek.html

[Page 2]
en=pg2_english.html
...
```

Figure 8 – Configuration file excerpt for reverse engineering a multilingual service

Once invoked for reverse engineering a multilingual service, the reverse engineering software reads the pertinent configuration file and processes the pages specified therein sequentially. For each page, the reverse engineering software extracts the (language, filename) pairs, and parses the designated files to create the corresponding object models. The object model of the first file is analysed to create TSEs, TSE groups and forms, as described in sections 4.2.1-4.2.3. When a text, message, label or help text is extracted from the page, its location in the object model of the first file is determined, and the object models of the subsequent files (corresponding to the remaining languages) are accordingly traversed to locate the matching resource, as expressed in each one of the other languages.

To make this procedure more clear, let us consider an example. Figure 8 presents a portion of a form from a "Password request service", where the form name is placed at the first cell of the first row of the outermost table, while the second cell of the same row contains help items, pertaining to the form as a whole. The form contains a single TSE group, namely "Type of passport required", whose elements are laid out using a nested table. The first row of this table includes the name and the description

of the TSE group, while the subsequent rows contain the TSEs themselves. Each one of the TSE rows contains the label, the mandatory field indication, the options and the help item.
The reverse engineering software analyses the form and determines the location of each resource; for instance, the form name can be found at the location Table1/Row1/Cell1, using the page object model traversal path. The same path is followed in other file corresponding to a translation of the same page in some different language to retrieve the pertinent resource in that language. Similarly, the label of the first TSE ("What type of passport…") can be found at the location Table1/Row2/Cell2/Table2/Row2/Cell1 and so forth.

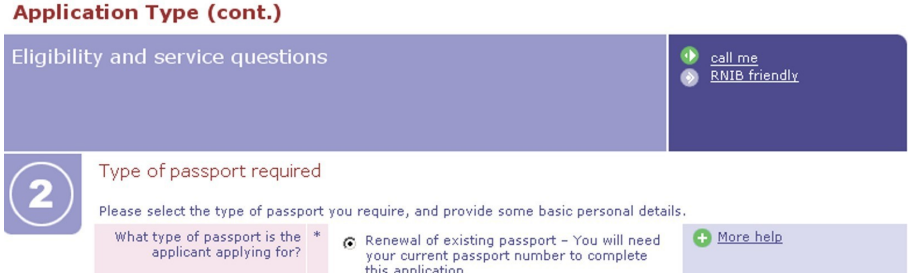

Figure 9 – Locating elements in the object model

A clear limitation of this approach is that it cannot handle cases where the linguistic properties of some of the languages that the service is offered in require a different layout. For instance, if some service is offered in English and Saudi-Arabian, the method will not work since text flow in Saudi-Arabian is right-to-left, requiring thus a different layout (labels would be placed for at the *last* table cell, not the first). Similar issues with Asian languages exist as well. Insofar, the reverse engineering software will not handle correctly these cases and the service developers should manually import the required resources into the SmartGov platform.

5 Evaluation

In order to evaluate the correctness of the reverse engineering heuristics presented in the previous section, an experiment was set up, in which a number of services were processed by the reverse engineering software and the results were reviewed for completeness (all artifacts on the page were recognized) and accuracy (artifacts and their elements were recognized correctly). The services varied in the number of forms employed, form size (number of fields) and form complexity (type and nesting of HTML constructs to achieve the desired visual layout as well as complexity of JavaScript code). A total of 12 services (3 of which multilingual) comprising of 63 forms were analyzed in this experiment. The services used in the evaluation were selected according to the following criteria:

1. the 50 services used in the artifact pattern identification phase for determining artifact types and patterns were excluded from the evaluation stage. This decision was made because the HTML code patterns used in these services were already known to the reverse engineering software, and thus results higher success ratios, not representative of typical software operation, were bound to be measured if the software operated on the analyzed services.
2. eight out of the twelve services that were used in the evaluation stages were drawn from the portals that the 50 analyzed services were drawn from, and had been deployed by a ministry or agency that had deployed at least one of the analyzed services (Category 1). For instance, the Greek Ministry of

Finance had deployed multiple services regarding electronic taxation document submission, including VAT statements, generic tax return forms (submitted by all individuals and enterprises), and three specialized tax return forms that accompany the generic one and provide specific income details pertinent to professional classes (self-employed workers, sailors and real-estate owners). Out of these services, four were used in the artifact pattern identification phase and one in the evaluation phase.

3. the four remaining services were drawn from the same portals, but no services deployed by the same ministry or agency were included in the 50 analyzed services (Category 2).

Services from categories 1 and 2 above were initially considered separately, because it was anticipated that the reverse engineering software would have greater success rates for category 1. This stemmed from the hypothesis that services developed and deployed by the same ministry or agency would have been crafted using similar methods and techniques, and would thus have similar patterns appearing in the HTML code, and since the reverse engineering software was already aware of these patterns, it would be easier for it to identify the artifacts. Statistical analysis of the obtained results, however, did not support this hypothesis; the differences in success rates among the two categories were found to be less than 1.5% in all cases, therefore in the rest of this document both categories are presented collectively. Finally, medium to high complexity services were selected for the evaluation, excluding trivial single-form services with 8 to 15 fields, for which success rates typically exceeded 95% (as determined in the software development and testing phase). All services used in the evaluation included field groups, while four of them included the repeating row layout (Figure 5); we also note that the artifact layouts in the selected services covered all layout cases and options presented in section 4 (label and help item placement, validation checks, group arrangements etc).

The results of the evaluation are presented in Table 2 and discussed in the following paragraphs.

Correctness metric	Complete & accurate	Inaccurate	Incomplete
Input element identification	96%	4%	0%
Input element-label coupling	81%	19%	0%
Input element data type determination[3]	64%	11%	25%
Input element validation check correctness	75%	3%	22%
Input element correlation with help items	78%	12%	10%
Group identification[4]	63%	11%	26%
Group name extraction	61%	17%	22%
Group membership correctness	65%	4%	31%
Group correlation with help items	62%	11%	27%
Form name extraction	73%	19%	8%
Form correlation with help items	69%	25%	6%
Multilingual label correlation	83%	3%	14%
Multilingual help item correlation	72%	4%	24%

Table 2 – Reverse engineering correctness analysis results

Input element identification, i.e. the task of locating the input elements on the page, was quite successful, achieving a correctness of 96%. This was anticipated, since this task merely traverses the pages' object model and checks whether each element is an HTML input field. The 4% of errors that were detected is due page elements which were intended by designers to display help texts and messages and were incorrectly interpreted by the reverse engineering software as read-only input elements.

Coupling of input elements with their labels also achieved a satisfactory performance, rising up to 81%. All cases employing the *label for* construct were correctly recognized, but these constructs were only used in two services. In the remaining services label correlation with input elements was determined through placement (either juxtaposition or table cell adjacency). The heuristics applied in these cases resulted to approximately 20% of errors, mainly for two reasons: the first was that in some cases labels appeared on the *right* of the related input elements, while the second was that in certain forms certain items intervened between the labels and the input elements. For instance, in the Greek taxation documents input elements were preceded by a *field identifier*, copied from the paper-based documents (see Figure 10; these codes were used in the paper-based documents for convenience in the data-entry procedure). In these cases the reverse engineering software incorrectly perceived the field code as a label for the input elements.

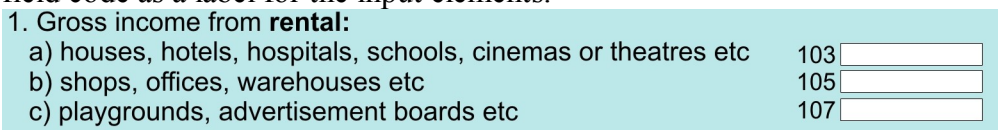

Figure 10 – Intervening field codes in a Greek taxation document (translated into English)

[3] "Inaccurate" refers to the cases that validation checks did exist but the reverse engineering software failed to determine the correct data type; "incomplete" refers to cases where no validation checks existed.

[4] "Inaccurate" refers to cases that a non-existent group was identified; "incomplete" refers to cases that an existing group was not identified.

The average success of data type determination was 64%, the percentage however varies significantly among different services. More specifically, there existed services (three in number) which did not include any Javascript code to check user input in free type-in fields (user input is checked solely by the back-end code, which is not considered in the reverse engineering process), thus the reverse engineering software had no means to determine the data type of these fields. In these services, the reverse engineering software could only determine correctly the types of input elements with closed sets of values (typically select widgets, check boxes and radio buttons) and the correctness score for these services is a mere 24%. In the other services, the correctness score was a respectable 72%.

Validation checks for input elements (including mandatory input designations) were correctly identified in 75% of the cases. The trivial case of recognizing elements with the *readonly* and *disabled* attributes was always successful; searching for invocations to "this.blur()" and "anotherfield.focus()" in the "onfocus" and "onselect" event handlers was 100% accurate but contributed only to recognizing an 8% of the overall read-only fields. The asterisk heuristic for identifying mandatory fields was successful in 85% of the cases, with the remaining cases being asterisks placed in locations not covered by the heuristic or asterisks denoting that explanatory texts followed (and misinterpreted as mandatory indications). The heuristics comparing the element value to the null string or its length to zero and directly invoking the "alert" function if the condition held were successful only in 53% of the cases, because in many cases more code intervened between the comparison and the display of the message. Additionally in many cases implementers used custom functions to detect whether some field was mandatory, and these functions were invoked using the field id (and possibly some message) as a parameter, either upon field change or upon form submittal. These cases were not detected by the reverse engineering software. In overall, 3% of the results were inaccurate, having mostly recognized non-mandatory elements as mandatory (the asterisk placed besides the input field in these cases actually referred the user to some explanatory text below), while the remaining 22% of errors were cases that not all validation checks were recognized and packed within the TSE.

Input element correlation with help items was found to be satisfactorily successful (78%). This was due to the fact that help items almost always appear alongside the input elements, or the element labels themselves serve as hyperlinks to the help texts. A 14% portion of the failures in this class was due to the fact that help items were not placed as expected on the forms including the cases that labels were placed on the right of input elements, in which labels were mistook as help items and the cases that additional items intervened between the input field and the help text/anchor. An additional 6% was owing to the fact that the reverse engineering software could not successfully retrieve hyperlink targets (authentication and/or cookies were required to perform the retrieval).

Metrics related to input element groups (identification, name extraction, membership correctness and correlation with help items) appear noticeably lower than input element-related metrics, since success rates range between 61% and 65%. This lag has been traced to the nesting and complexity of HTML table structures which are used to lay out group elements on the form. More specifically, in many cases nested tables are used to create aesthetically pleasing borders around a group, empty columns spanning along all rows are employed to create indentation effects and so forth; tables are also extensively used to create navigation areas (e.g. a column on the left with menu items). As an example consider the service of Online Registration Renewal service of

the State of California depicted in Figure 11[5], where an "outer" table is used for creating the "menu" (darker column, on the left) and "application data" areas, while additional nested tables are employed to lay out individual group elements.

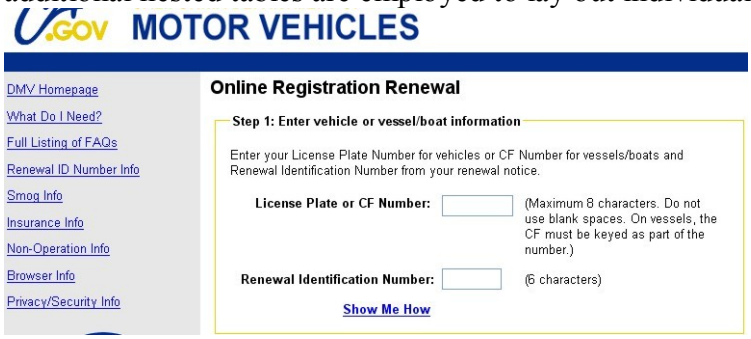

Figure 11 – Using nested tables laying out menus and borders

Although these arrangements are easy for humans to perceive, it has proven cumbersome to detect and handle correctly within the reverse engineering software, leading to a considerable amount of errors. Because of these layout complexities, the "group name extraction" metric produced a 17% of inaccurate results (textual elements of the form incorrectly were used as group names) and 22% of incomplete results (the group heading text could not be determined because HTML DOM elements intervened between the heading and the beginning of the group); intervening elements lead also to a 31% incompleteness errors for field group membership since the software deduced that the elements were separated and thus not parts of the same group. Interestingly, group help items were found in many cases to be mistakenly considered as form-level help items: this happened because the help hyperlinks were found too far apart from the group label, and thus were not accounted as group help; being though recognizable help items, they were bundled with the containing object, which was the form.

Instead of trying to identify and correctly analyze all possible layouts within the reverse engineering software, an alternative approach was adopted for tackling this problem: the pages are saved on the disk and are *preprocessed* by humans, using visual HTML editors, before they are fed into the reverse engineering software. In such a visual environment (Figure 12), it is easy for the human user to select the areas containing the groups and unnest them and/or remove the menu areas, producing thus an HTML file that can be processed by the reverse engineering software with significantly better success rates.Table 3 presents the correctness metrics related to group elements after applying HTML pre-processing. Note that HTML pre-processing can be used to tackle additional issues leading heuristics to failure, such as the case of labels appearing on the right of input elements, where a simple table column transposition will allow the reverse engineering software to correctly recognize the texts as being labels for the fields and not help items.

[5] The service is accessible at https://vrir.dmv.ca.gov/vrir2/rinPlatePage.do

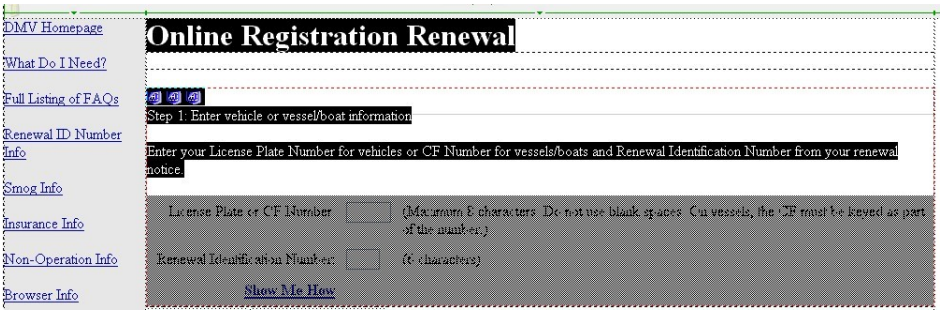

Figure 12 – Editing a form to remove complex layouts

Correctness metric	Complete & accurate	Incomplete	Inaccurate
Group identification	87%	4%	9%
Group name extraction	84%	6%	10%
Group membership correctness	91%	3%	6%
Group correlation with help items	85%	6%	9%

Table 3 – Correctness metrics related to groups after applying HTML pre-processing

Finally, the assessment for reverse-engineering multilingual resources has shown that multilingual labels are correlated with the input elements quite satisfactorily, at a success rate of 83%. Failures to correctly recognise labels have been attributed to two reasons: the first was that in some cases, HTML pages had been manually edited for each distinct language, so as to adapt to the specific needs of each language (e.g. a label in some language may be very short due to, say, use of an abbreviation, while in some other language it may be wordy, necessitating thus more space). The second reason for failures in this task was the fact that HTML pre-processing (wherever employed) had to be consistently applied to all files corresponding to the same service page (one for each language); however, some pages were inconsistently edited (e.g. a *div* element appeared in some files and did not appear in other ones), thus correlation was unsuccessful. Association of help items with the input elements shows a lower success rate than help items; this is due to the fact that a number of help items were provided by means of hyperlinks, and the retrieval of the correct content failed, because authentication, cookies and/or browser data (mainly the *preferred languages*) were needed to either allow the retrieval altogether or to return the content in the correct language. The incompleteness errors for multilingual help correlation are significantly more than the respective error rate for "normal" help items, because multilingual services have been found to employ hyperlinks for help texts at a higher percentage than single-language services, thus errors due to authentication/cookies while retrieving multilingual content are more frequent in multilingual services.
The introduction of a more elaborate content retrieval mechanism that would provide the appropriate HTTP headers within the request is expected to increase the success rate for this metric.

6 Conclusions – Future work

In this paper we have presented a heuristics-based method for reverse engineering electronic services to create constructs of high levels of abstraction. These constructs can be managed more effectively and used for knowledge sharing within an organization and/or reused for the creation of new services. The proposed approach

has been implemented and evaluated on a number of services to validate its correctness. The evaluation procedure has shown that the heuristics perform satisfactorily in most cases, while manual pre-processing of HTML pages can be introduced to further improve the success rates of the reverse engineering process, especially in cases of complex layouts.

Future work will be mainly targeted to extend the reverse engineering procedure so as to take into account the back-end code which accepts, validates and stores into repositories the data entered by the users. Progress in this direction is however slow, because this code is not made available by the organizations. Improvement of the page retrieval mechanism, in order to correctly retrieve pages requiring authentication, cookies and browser data is also under consideration. Finally, the introduction of abstraction mechanisms to elucidate *templates* from "similar" elements (e.g. templates for social security numbers, car registration plates etc) will be examined, since such templates can, similarly to object types, group characteristics of commonly used constructs and thus simplify their maintenance.

Acknowledgement: The authors would like to thank Mr. George Galyfos for his work on the implementation of the reverse engineering software.

7 References

[1] Top of the Web. User Satisfaction and Usage Survey of eGovernment services, December 2004. http://admin.topoftheweb.net/resultfiles/2004/report.pdf [Accessed March 2008]

[2] e-Europe. Common list of basic public services, 2000. Accessible at http://europa.eu.int/information_society/eeurope/2002/action_plan/pdf/basicpublicservices.pdf [Accessed March 2008]

[3] United Nations, E-Commerce and Development Report 2004, Available from http://www.unctad.org/en/docs/ecdr2004_en.pdf

[4] Vassilakis, C., Laskaridis, G., Lepouras, G., Rouvas, S., Georgiadis, P. A framework for managing the lifecycle of transactional e-government services. Telematics and Informatics vol. 20, 2003, pp. 315–329

[5] Vassilakis, C., Lepouras, G., Fraser, J., Haston, S., Georgiadis, P. Barriers to Electronic Service Development, e-Service Journal, vol. 4(1), 2005.

[6] Georgiadis P. Lepouras G., Vassilakis C. et al. A Governmental Knowledge-based Platform for Public Sector Online Services. Proceedings of the 1st International Conference on Electronic Government-EGOV 2002, pp. 362-369

[7] SmartGov Consortium, 2002. SmartGov Project Deliverable D41: User Requirements, Services and Platform Specifications. Available from http://www.smartgov-project.org

[8] Chikofsky, E. and Cross, J. Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software vol. 7, no. 1, 1990, pp. 13–17.

[9] Tilley, S. The canonical activities of reverse engineering. Annals of Software Engineering, vol. 9, 2000, pp. 249-271.

[10] Kazman, R., S. Woods, and J. Carrière. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. Proceedings of the 5th Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 154–163.

[11] Sowa, J..Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, Reading, Massachusetts, 1988.

[12] Di Lucca, G. A., Casazza, G., Di Penta, M., Antoniol, G. An Approach for Reverse Engineering of Web-Based Applications. Eighth Working Conference on Reverse Engineering (WCRE'01), 2001, pp. 231-240.

[13] Di Lucca,G. A., Fasolino, A. R., Tramontana, P. Reverse engineering web applications: the WARE approach. Journal of Software Maintenance and Evolution: Research and Practice, Volume 16, Issue 1-2, January-April 2004, 2004, pp. 71–101.

[14] L. Paganelli, F. Paternò, A Tool for Creating Design Models from Web Site Code. Technical Report 2003-TR-56, Available at http://dienst.isti.cnr.it/Dienst/UI/2.0/Describe/ercim.cnr.isti/2003-TR-56
[15] Essanaa S. B., Lammari, N. RetroWeb: A Web Site Reverse Engineering Approach. Proceedings of Web Engineering: 4th International Conference, ICWE 2004, Munich, Germany, July 26-30, 2004, pp. 306–310.
[16] Essanaa S. B., Lammari, N. Improving the Naming Process for Web Site Reverse Engineering. Proceedings of NLDB 2004, LNCS 3136, 2004, pp. 362–367.
[17] Cognos. Cognos Business Intelligence Overview. 2005. http://www.cognos.com/products/cognos8businessintelligence/index.html
[18] Computer Sciences Corporation. CSC Knowledge Management. 2005 http://www.csc.com/solutions/knowledgemanagement/
[19] Captaris. Business Information Delivery Solutions. 2005. http://www.captaris.com/products_and_solutions/business_information_delivery/index.html
[20] TheBrain Technologies Corporation. TheBrain Enterprise Knowledge Platform. 2005. http://www.thebrain.com/
[21] Empolis. e:Corporate Knowledge Suite. 2005. http://www.empolis.com/en/D70A9AD4D55C470E95D449E85ACEBE9D.php
[22] Abecker, A. et al. Workflow-Embedded Organizational Memory Access: The DECOR Project. In: IJCAI'2001 Workshop on Knowledge Management and Organizational Memories, August 2001, Seattle, Washington, USA.
[23] Desouza, K. Facilitating Tacit Knowledge Exchange. COMMUNICATIONS OF THE ACM June 2003/Vol. 46, No. 6, pp. 85-88.
[24] Gasson, S. The Management of Distributed Organizational Knowledge. Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), Track 8, Volume 8, 2004
[25] BEA Systems Inc. Bea Logic Workshop. 2005. http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic/workshop
[26] Adobe Systems Incorporated. Adobe Government Forms. 2005. http://www.adobe.com/government/pdfs/95003217_gov_forms_sol_ue.pdf
[27] Adobe Systems Incorporated. Adobe LiveCycle Designer. 2005. http://www.adobe.com/products/server/adobedesigner/main.html
[28] SmartGov Consortium, 2004. SmartGov Project Deliverable D13: Final Project Report. Available from http://www.smartgov-project.org