Program Transformations for Web Application Restructuring

Filippo Ricca and Paolo Tonella

ITC-irst, Centro per la Ricerca Scientifica e Tecnologica,

38050 Povo (Trento) – ITALY

Phone: +39.0461.314524

Fax: +39.0461.314591

tonella@itc.it

Filippo Ricca

ITC-irst, Centro per la Ricerca Scientifica e Tecnologica,

38050 Povo (Trento) – ITALY

Phone: +39.0461.314522

Fax: +39.0461.314591

ricca@itc.it

Program Transformations for Web Application Restructuring

This chapter aims at providing a presentation of the principles and techniques involved in (semi)-automatic transformation of Web applications, in several different restructuring contexts. The necessary background knowledge is provided to the reader in the sections about the syntax of the multiple languages involved in Web application development and about the role of restructuring in a highly dynamic and rapidly evolving development environment. Then, specific examples of Web restructuring are described in detail. In the presentation of the transformations required for restructuring, as well as in the description of the grammar for the involved languages, TXL (Cordy et al., 2002) and its programming language is adopted as a unifying element.

The chapter is organized into the following sections: in the section following the *Introduction*, the problems associated with the analysis of the multiple languages used with Web applications are discussed. Then, the process of Web application restructuring is considered. Three examples of Web restructuring are described in more detail in the next three sections (design restructuring, migration of a static Web site to a dynamic Web application, consistency among monolingual portions of a multilingual Web site). Related works and concluding remarks are at the end of the chapter.

Keywords: Web Applications, Program Transformation, Restructuring.

INTRODUCTION

All known problems that characterize the evolution of software systems are exacerbated in the case of Web applications, because their typical life cycle is much shorter and their development iterations are executed faster and more frequently. In fact, Web applications must accommodate continuing changing requirements, since the user needs evolve rapidly over time. Moreover, the infrastructure technology is itself subject to continuous updates. New technologies emerge at a high rate, replacing the existing ones. A strong pressure for change comes also from the market, where it is fundamental to be up to date with the new services and paradigms.

The consequences of a high evolution rate are often very negative for the internal quality of the Web applications. The initial architecture is subject to a drift, so that the original assignment of responsibilities to components is violated and the dependences among components tend to resemble a fully connected graph. Code fragments may be duplicated, thus making the related functionality delocalized, and consequently difficult to maintain and reuse. While in the beginning such negative effects may be invisible to the end user, after some time the application becomes unmanageable. New requirements are difficult to accommodate, nobody has a sufficient understanding of the architecture to allow for major changes, and defects tend to be inserted with the modifications, due to unexpected ripple effects.

The evolution scenario described above is more complicated with Web applications than with traditional software, because of the technologies underlying their development. In

fact, Web applications involve a basic client-server architecture, with the browser as the client and the Web server on the other side, upon which higher level functionalities are built. Since multiple users can connect concurrently, user sessions must be handled over time. However, this is not supported by the HTTP protocol in use, which is stateless and connectionless. When transactions are executed inside user sessions the problem is even harder. Security is another central concern. The support technologies used to solve these problems are based on multiple standards and programming languages. Consequently, the resulting Web application is composed of a heterogeneous set of components written in different languages and based on different standards. Portability across environments and browsers is another factor that complicates the internal organization of the Web applications.

When the evolution of a Web application makes its source code difficult to maintain, the typical action that is undertaken consists of rewriting the entire application from scratch in the new setting. In this chapter, a different approach is considered. It is based on the notion of continuous restructuring and it assumes that preventive interventions can be made to avoid the progressive degradation that accompanies software evolution. Restructuring is an expensive activity for which a budget is rarely available in a highly competitive market such as that of Web applications. However, the possibility to (partially) automate restructuring can make it cost-effective. Its adoption would involve periodic interventions of preventive maintenance, that precede the actual implementation of the required changes.

WEB APPLICATION LANGUAGES

Web applications are typically developed in multiple programming languages. Browsers are able to interpret HTML, so that this is the language that every Web application must include. However, any non trivial Web application involves some processing that occurs on the server side and produces a part of the displayed information. Moreover, the interactive facilities offered by purely HTML pages are often too poor and more advanced navigation modes are achieved by inserting code that runs on the client side. Thus, a typical Web application is written in at least three languages: HTML, a server side language (e.g., PHP) and a client side language (e.g., Javascript). Additional languages may be involved in the server side computation. For example, database access is typically achieved using SQL.

This section deals with the problems specific of each programming language used in Web development. The HTML language has features that are challenging from the point of view of its analysis and manipulation. Moreover, the coexistence of multiple languages in a single source file poses additional problems. A grammar of the HTML+PHP+Javascript languages is sketched in order to indicate possible solutions to the highlighted problems. The TXL language is used for grammar specification. In the next section, the most

important TXL constructs, used in the following sections, are summarized, for convenience of the readers who are not familiar with this transformation language.

TXL

TXL (Cordy et al., 2002) is a programming language designed to support the transformation of source code represented as a tree. A TXL program consists mainly of a context-free grammar and a set of transformation functions and rules. Given a grammar of the language under analysis, the parse tree of the source code is first, automatically, built, and TXL functions/rules are applied to it.

In a TXL grammar specification, each *'define'* statement contains the productions for each nonterminal. The vertical bar | is used to separate the alternatives. Nonterminals appearing in the body of a *define* statement are enclosed in square brackets, while symbols not enclosed in brackets are terminals. Conventionally, the nonterminal *program* is the goal symbol, i.e., the root of the parse tree being built.

Any nonterminal enclosed in square brackets may be modified by a nonterminal modifier. An example is the modifier *repeat.* If a nonterminal is preceded by *repeat,* zero or more repetitions of such a nonterminal are admitted. For example, the following *define* statement specifies that the nonterminal *program* derives into a repetition of zero or more *element* nonterminals:

define program

    [repeat element]

end define

Transformation of the source code is specified as a set of transformation functions and rules. Each function (rule) contains a *pattern* and a *replacement*. The former is the pattern that the function's (rule's) argument tree must match in order for the function (rule) to be applicable, while the latter is the result of the function (rule) application. A rule has the same syntax as a function (except for the keyword *function*, replaced by *rule*), but its semantics is quite different: a rule repeatedly searches the scope tree where it is applicable and replaces every such match, while a function is applied just once to its argument tree. The TXL syntax for the specification of a pattern and its replacement is the following:

replace [type]

    pattern

by

    replacement

where *type* is the subtree root type (nonterminal), and *pattern* must comply with such a type (i.e., it has to be derivable from the nonterminal). Of course, *replacement* must also be of the same type.

Rules and functions may contain constructors (keyword *construct*) and deconstructors (keyword *deconstruct*). Constructors allow building intermediate subtrees for later use, while deconstructors allow breaking tree variables into smaller pieces (subtrees), by means of a more refined pattern. When a deconstructor pattern does not match, the whole function/rule is considered to have not matched its pattern at all. The following examples deconstruct/construct a list into/out of head and tail:

deconstruct list

    head [element] tail [repeat element]

construct list [repeat element]

    head tail

Searching deconstructors (keyword *deconstruct* *) are a specialization of deconstructors. They allow searching and taking a subtree of the deconstructed tree. A searching deconstructor has the following form:

deconstruct * tree

    pattern

The semantics of this construct is: searching the tree bound to variable *tree* for the first subtree that matches the *pattern* and binding the pattern variables accordingly.

HTML

Parsing HTML may seem an easy task at first sight, since HTML is defined precisely by its DTD (Document Type Definition, see www.w3c.org). Unfortunately, only a few real world Web pages are compliant with it: HTML pages with missing end tags, overlapped tags, tags in the wrong context with respect to the DTD, etc., are quite common in existing Web sites and are not rejected by many of the available HTML browsers. For this reason our grammar (see Fig. 1) does not enforce the DTD, being more general. The parse tree that is built out of this grammar is pretty simple, consisting just of a sequence of elements: tags (nonterminal *Tag*), end tags (*endTag*) and texts (*htmlText*).

```
define program
    [repeat element]
end define
define element
    [htmlText] | [TagBlock] | [Tag] | [endTag]
end define
define htmlText
    [repeat idORspecial]
end define
define idORspecial
    [id] | [special]
end define
define TagBlock
    Block([id][opt attrs] [repeat element])[id] | [empty]
end define
define Tag
    <[id] [opt attrs]>
end define
define attrs
    [repeat attr]
end define
define attr
    [id] = [value] | [id]
end define
define endTag
    </[id]>
end define
```

Fig. 1: *HTML grammar specified in TXL.*

In Fig. 1, *special* includes all punctuation marks and special symbols; *value* represents strings (possibly double or single quoted); *id* indicates an identifier. The nonterminal *TagBlock* is never expanded, since its syntax is actually not part of HTML. Its usage is clarified below.

The HTML parser works into two steps. First, a flat sequence of *element* subtrees is appended to the parse tree, ignoring the nesting structure of the HTML tags. Then, nesting is taken into account and the initial parse tree is modified to properly represent it.

In the first step, the parser does not attempt to match start tags with corresponding end tags. This is done more easily after the parsing phase (second step) by a set of transformation rules (not shown here) that build *TagBlock* elements. A *TagBlock* is a composite element (see Fig. 1) consisting of a tag (first *id*), a matching end tag (second *id*) and a sequence of elements (*repeat element*) in between. Each time a pair of matching open and closed tags are found in the initial parse tree, a *TagBlock* subtree is constructed which replaces the original pair of tags and the enclosed *element* sequence.

The main advantage of this approach to HTML parsing is that documents that do not comply with the standard HTML DTD are not rejected. A parse tree can always be constructed for any HTML input, with start tags and matching end tags transformed into *TagBlocks* whenever possible. Such a transformation reproduces the structure of the HTML document in the parse tree when the HTML DTD is respected, while maintaining a flat sequence of elements when the HTML DTD is violated. Having *TagBlocks* in the

parse tree is essential in the process of restructuring by means of transformation rules, because typical transformation patterns include the specification of a nesting structure to be matched.

Multiple Programming Languages

Another important aspect to consider in the parsing phase is that a Web page may contain several different languages: HTML, scripting languages (e.g. Javascript), style sheet languages (e.g. CSS) and server side languages (e.g. PHP). In the following, HTML, Javascript and PHP are considered. A solution to this problem is working concurrently with multiple languages. TXL permits defining grammars for the different languages that may appear in a same source file and allows working with 'mixed trees' (see Fig. 2).

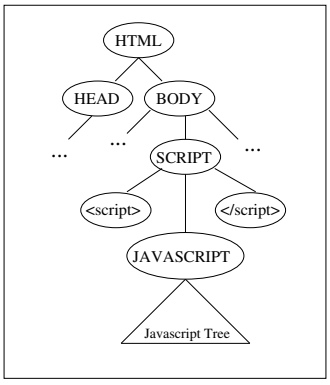

Fig. 2: *Mixed parse tree, containing a Javascript subtree.*

In order to admit multiple subtree types inside the resulting parse tree, the HTML grammar given in Fig. 1 has to be extended, by redefining the nonterminal *element* in the following way:

redefine element

… | php_code | javascript_code

end redefine

In addition to the previous productions (indicated by the ellipsis), *element* can now derive into a PHP or a Javascript subtree. The following new productions for the PHP and Javascript subtrees are added:

define php_code

[start_php] [sequence_php] [end_php]

end define

define javascript_code

[start_javascript] [sequence_javascript] [end_javascript]

end define

The nonterminals *start_php* and *end_php* are the alternative tags that can be used to insert portions of PHP inside HTML code (for example <? and ?>), while *sequence_php* is a

repetition of PHP statements, PHP declarations and PHP expressions, i.e., the actual body of the PHP code fragment. The same holds for Javascript code portions.

WEB APPLICATION RESTRUCTURING

The life cycle that characterizes Web application development is extremely short and the user requirements are very volatile, so that the evolution of a Web application is particularly hard. Radical changes in the adopted technologies, user interface, data access and format, and presentation modes, are quite common, and may result in a progressive degradation of the Web application architecture, if not addressed properly. Continuous restructuring, as prescribed in the Extreme Programming development process (Jeffries et al., 2000), is the key to manage the complexity resulting from a high evolution rate.

Process

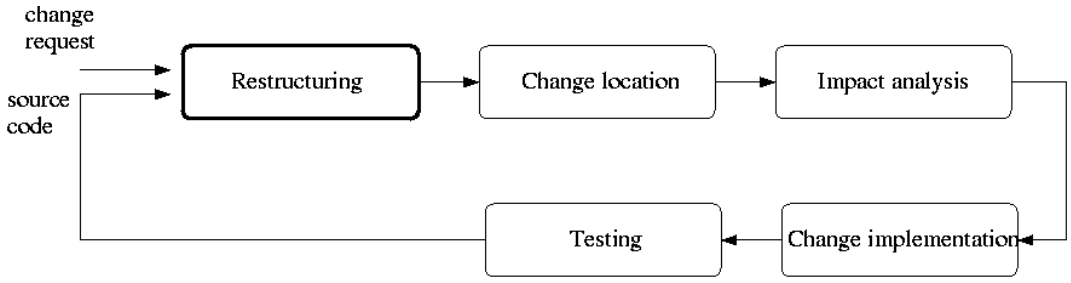

Fig. 3: *Role of restructuring in the process of Web site evolution.*

In Fig. 3, the role of restructuring in the whole Web engineering process is highlighted. During Web site evolution, a modification of the source code is triggered by a change request. The origin of the change request may vary, ranging from a new user requirement, to the availability of new technologies, but it is always reflected into a source code change.

When no restructuring practice is in use, the process of source code change involves four main steps. First, the code portion to be changed is located. Such a task is mainly a program comprehension task, which might be quite difficult if the Web application has a large size and is not well documented. Then, the direct and indirect effects associated with the located change are assessed (Impact analysis), to determine the actual amount of code to be updated. The next step is the implementation of the change, followed by a testing phase where the functionalities expected by the application are verified. In the context of Web applications, which are subject to a tremendous pressure for change, the cycle depicted in Fig. 3 is repeated several times, with very close deadlines for its completion.

The consequence of a fast and repeated execution of the four rightmost steps in Fig. 3 is typically a degradation of the initial architecture of the Web application and an increasing difficulty in managing the complexity of the software and its evolution. These negative effects can be contrasted by introducing a fifth step, which precedes the other four ones, and is focused on a preventive reorganization of the Web application, aimed at improving

its internal structure, thus simplifying and anticipating future changes. If supported by automated tools, the introduction of a restructuring step may be cost-effective even in a rapidly evolving environment with a very short life cycle, such as that of Web engineering.

Preventive restructuring ensures that location of the change will be easier, since functionalities are properly assigned to software components. The impact of the change will be much more limited, thanks to a lower coupling among components. Implementation of the change will be straightforward, and testing will also be limited to the test cases that exercise the (small) code portion that was changed.

Restructuring Categories

The main kinds of restructuring interventions that are specific to Web applications are the following:

- Syntax update
- Internal page improvement
- Design restructuring
- Separation of style/presentation from content
- Dynamic page construction

Syntax update

The W3C consortium defines the standard version of the HTML language. However, popular browsers are often tolerant to syntax errors and non compliances with the standard, and they succeed rendering a given HTML page even if it is not syntactically correct. Moreover, the tools that support the creation of Web pages often introduce syntax errors in the HTML code they automatically produce. As a consequence, several existing HTML pages do not follow the standard.

To further complicate the matter, the standard itself which defines the syntax of HTML changes over time. Old documents should therefore be updated to the new versions of the standard. However, browsers are often tolerant also in this respect, accepting documents that follow the old standard, as well as those that follow the new one.

The main disadvantage of a Web site that contains pages with syntax errors or pages not compliant with the current standard is that its correct visualization depends on the ability of the browsers to recover from syntax errors in a reasonable way. Such a feature may vary from browser to browser, so that there is no warranty of portability across browsers. Moreover, new versions of a same browser may handle syntax errors differently. For these reasons, it is advisable to update the syntax of the HTML pages in a Web site, making them compliant with the standard. One of the tools available for such a purpose is *Tidy* (see the W3C Web site, www.w3c.org).

Internal page improvement

Restructuring interventions, in the *internal page improvement* category, aim at supporting easier-to-follow and shorter navigation paths. Moreover, they operate on the mechanisms used to inter-connect documents and objects.

For example, the transformation of absolute URLs into relative URLs (by means of the tag *BASE*) is an internal improvement of a navigation mechanism, that makes the references to other resources independent from the base directory containing them and from the server hosting the site. The addition of a *NOFRAMES* section in the HTML code allows a correct navigation also in browsers which do not handle frames. The addition of a description for non loaded objects (e.g., applets, movies, etc.) makes the Web site useful also for users who disable such object loading.

Design restructuring

Restructuring of the design refers to the overall architecture of the Web application. It affects the decomposition of the software into components and the mechanisms used to make components communicate with each other. The quality of the architecture is one of the main factors that influence the maintainability and evolvability of the Web application. Thus, these restructuring interventions are essential to preserve and increase the internal quality of the source code.

Examples of design restructuring are the reorganization into frames of pages containing hard-coded menus, the migration of part of the content to a database (see also below), the

separation of different concerns (such as security, style, etc.) and their assignment to different components, the factorization of replicated functionalities into shared library components.

Separation of style/presentation from content

Separation of style/presentation from content can be achieved by resorting to style sheets, such as CSS, to control layout and presentation, so that formatting tags can be removed from the pages. Moreover, Content Management Systems (CMS) can be employed to reach an even higher degree of separation between the content, not necessarily written in HTML, and its HTML version, that is published on the Web.

Dynamic page construction

Dynamic page construction is useful each time the actual HTML page transmitted to the browser depends on some user-specific parameters. If all possible combinations of parameters are accounted for by a static set of HTML pages, the number of such pages tend to explode, making the site impossible to manage. A parametric dynamic generation of these pages may solve the problem.

An example where dynamic generation might be convenient is the presence of several pages with a same structure (template), filled in with variable parts. Instead of replicating the structure for all possible variants, a same template is dynamically filled-in with information extracted from a database. Another example is a Web site that provides the same information in multiple languages (e.g., English and Italian). Instead of replicating

the pages for all supported languages, a shared page structure can be filled-in with a monolingual content selected dynamically.

In the following sections, three of the restructuring examples sketched above are considered in more detail: design restructuring, migration to dynamic sites, where data and presentation are stored separately, and support to the automatic consistency among Web site portions written in different languages.

DESIGN RESTRUCTURING

An example of design restructuring is the *automatic reorganization into frames*. This transformation allows separating menu and content in a Web application where the menu is inserted and replicated in all pages.

When a menu is embedded in Web pages, usually a portion of HTML code is devoted to it. For example, the HTML construct *TABLE* may be exploited to format the list of URL references in the menu. Such a construct is replicated in all pages that share the same menu. A limited number of programming patterns, similar to the usage of the *TABLE* tag described above, is typically used to insert a menu into a Web page. Thus, matching such patterns on the parse tree of the pages under analysis can be an effective way to discover the presence of a menu in a page.

A reorganization of the pages into frames simplifies the navigation and improves the maintainability of the Web site. In fact, if the menu is displayed in a separate frame, it remains always accessible during the navigation, and it does not have to be replicated in all pages that share it. When some menu item changes, a single HTML file has to be updated if frames are used, while all pages sharing the same menus need an update otherwise. An example of such a transformation is depicted in Fig. 4.

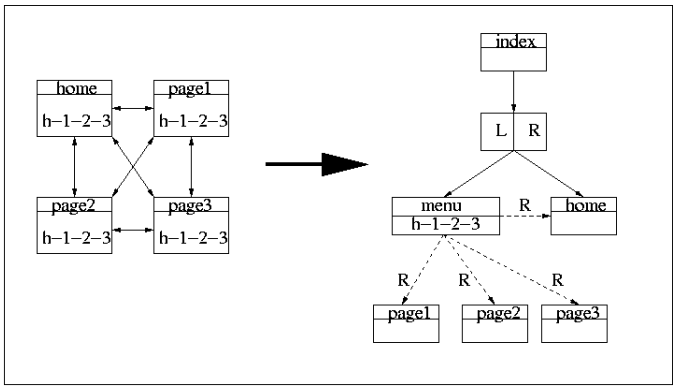

Fig. 4: *Reorganization of menus into a frame.*

In the initial Web site, all four pages *home, page1, page2, page3*, are linked with each other through the menu represented as *h-1-2-3* (see Fig. 4, left). After the transformation a new main page (*index*) is introduced. This page consists of two frames, with identifiers *L* and *R* (see Fig. 4, right). Page *menu* is loaded into frame *L,* while page *home* is the initial page of the frame *R*. The dashed edges labeled *R* indicate that the pages reachable

through the hyperlinks in page *menu* are loaded into the frame *R*, i.e., the page *menu* is used as a menu to force the loading of pages into the frame *R*.

The reorganization into frames requires the TXL functions/rules depicted in Fig. 5, as well as some (limited) intervention of the user, who has to choose which menu to migrate into the frame *L*. The rest of the process is totally automatic.

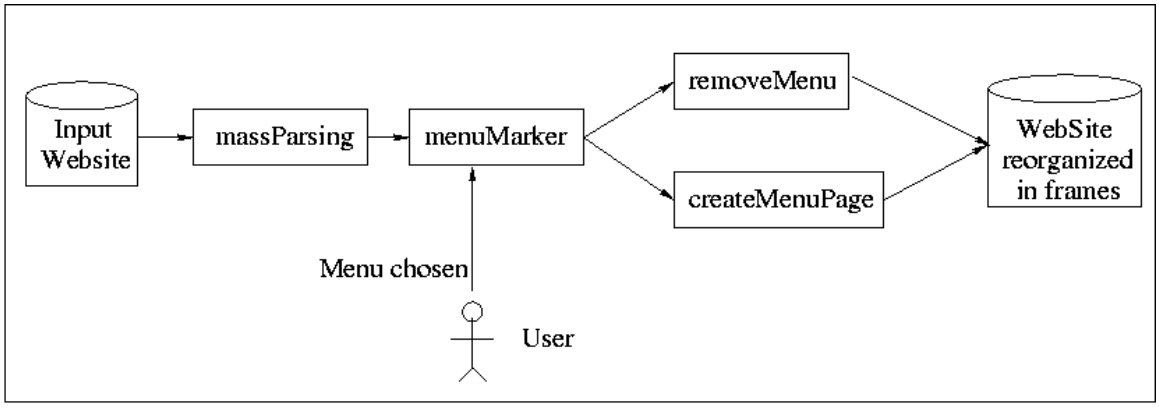

Fig. 5: *Main steps of the reorganization into frames.*

The TXL function *massParsing* parses the list of Web pages in the input Web site and builds a set of Parse Trees (PTs), one for each page. The function *menuMarker* applies the rule *markWithTagMenu* to each PT. For each page, this rule marks the identified menus by inserting them between <*MENU*> and </*MENU*> tags. A semi-automatic procedure devoted to the identification of potential menus inside a Web site was proposed in (Ricca et al., 2002). The user chooses one of such menus for restructuring.

The selected menu is then removed from each page containing it, being replaced by a single menu page to be loaded into the frame *L*.

The following rule, named *removeMenu*, is applied to each page of the initial Web site to remove the selected menu.

```
rule removeMenu
    replace [element]
      el [element]
    deconstruct el
      Block('MENU attrs [opt attrs] inBlock [repeat element] )'MENU
    by
      _
end rule
```

This rule searches an element that matches the pattern "*Block('MENU attrs [opt attrs] inBlock [repeat element])'MENU"*. If this element is in the Web page under analysis, it is deleted from the page. The replacement in the rule above is the underscore, indicating a subtree built according to the default constructor (in this case, an empty *element*).

The function *createMenuPage* (not shown here) searches the Web page *home* for the *TagBlock MENU*. The list of elements contained in the *TagBlock MENU* is filtered, and only tags of type anchor (*A*-tags) are considered. Anchors are refined by adding the target

frame (*target='R'*), to form the page *menu*. A fixed page *index*, loading *menu* into the left frame, and *home* into the right frame, is added to complete the transformation.

MIGRATION TO DYNAMIC SITES

Web sites of the first generation consist typically of a set of static HTML pages. Content and presentation are mixed and a same page structure is replicated every time a similar organization of the information is needed (for example, this happens for Web pages that are product-cards or personal pages). Such a practice poses several problems to the evolution of these sites: it is not easy to update the content, which is intermixed with formatting, and a change in the structure has to be propagated to all pages replicated in the Web site.

Fig. 6 shows a (semi) automatic restructuring process aimed at transforming a portion of a static Web site into a dynamic one.

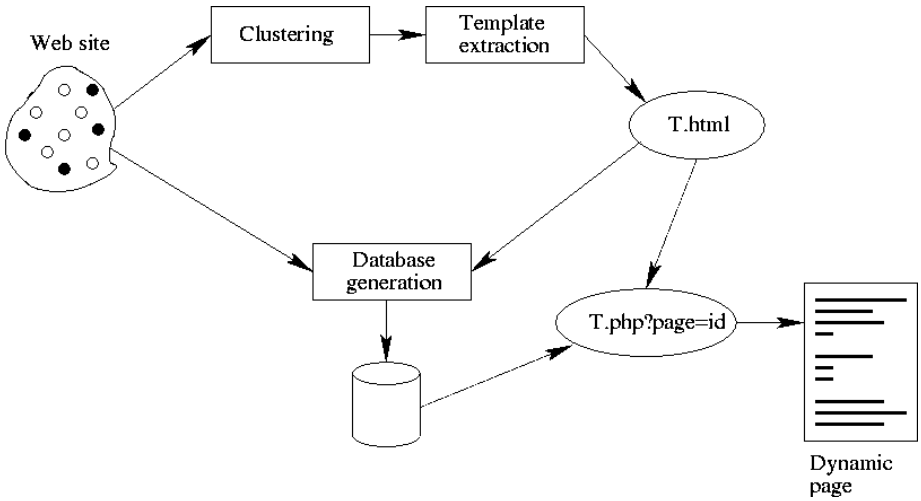

Fig. 6: *Transformation of static pages into a dynamic Web application.*

The process depicted in Fig. 6 replaces a set of static pages having the same structure with a unique server script (*T.php*) that generates dynamic pages. The dynamic pages generated at run time have a fixed part (the template, *T.html*) and a variable part that is built using the information stored in a database.

Clustering is used to recognize Web pages with a common structure (Ricca and Tonella, 2003). Moreover, the output of clustering is used in the template extraction phase that provides the candidate template, *T.html*. In fact, the template is the Longest Common Subsequence (LCS) of HTML *elements* that can be found in the Web pages grouped inside a same cluster. A comparison between original pages and template provides the records to be inserted into the database (database generation phase). Variable parts in the original pages that are preceded by fixed parts shared with the template represent the candidate records. In the end, a script generates the migrated pages dynamically from the

template and the database. Manual interventions (not shown in Fig. 6) are limited to the refinement (when necessary) of the database and of the template.

The subject of the remaining of this section is the database generation phase. This process assumes that the template *T.html* has already been computed (by means of the LCS algorithm). Specifically, variable parts are replaced by a *DB* tag during LCS computation, so that the actual template has the form of the skeleton of an HTML page, with variable parts marked by *DB* tags. Below is an example of such a template:

```
<HTML>
 <BODY>
    <TABLE>
       <TR>
          <TH> Name </TH>
           <TD> <DB field='name'> </TD>
       </TR>
       <TR>
          <TH> Surname </TH>
           <TD> <DB field='surname'> </TD>
       </TR>
    </TABLE>
 </BODY>
</HTML>
```

Then, the TXL rule *createTableOfFields* (not shown), applied to the template, generates a table of fields. This rule just extracts the first *htmlText element*, which precedes each tag <*DB*>. Such an element becomes the name of a (potential) database field. In the example above, *Name* and *Surname* are the candidate fields of the database being constructed.

At this point, for each field in the table of fields and for each page in the Web site, the TXL function *buildFieldPlusValue* is called. This function, shown below, generates the couples field-value for each page. In the end, the sequence of values in these couples form the record to be inserted into the database.

```
function buildFieldPlusValue page [repeat element] field [htmlText]
    replace [couple]
        _
    deconstruct * page
        field restWithValue [repeat element]
    deconstruct * restWithValue
        value [htmlText] rest [repeat element]
    construct c [couple]
        field : value
    by
        c
end function
```

The two searching deconstructors applied in sequence inside *buildFieldPlusValue* respectively check the presence of an *element* (of type *htmlText*) equal to the parameter *field,* and recover the next following *element* of type *htmlText,* which is used as candidate value for the given field. The couple *c* produced by this function pairs *field* and *value* (using the colon as a separator).

MULTILINGUAL SITES

Multilingual Web sites are expected to provide the same information formatted in the same way and with the same interactive facilities in more than one language (e.g., Italian and English). For such sites, design, quality assurance and evolution are more complicated than for monolingual sites. Multilingual contents have to be kept consistent with each other. The information provided should be the same for every language of the site: a change in a language has to be propagated to all other languages. Moreover, presentation and navigation should be language independent, still meaning that every change has to be properly propagated to all replications in different languages.

In (Tonella, Ricca, Pianta & Girardi, 2002) some techniques and algorithms have been proposed to support restructuring of multilingual Web sites so as to align the information provided in different languages and make it consistent across languages. The target of the restructuring process is an extension of XHTML (the new version of HTML supported by the W3C consortium) called MLHTML. Web pages written in MLHTML are ensured to

provide multilingual information consistently. In fact, the page structure is shared, instead of being replicated for every supported language. Thus, any change in the structure (presentation, navigation, etc.) is automatically reflected in all languages. Moreover, the multilingual content is managed in a centralized way, so that alignment can be enforced easily.

MLHTML adds just one new tag to XHTML, the tag <*ML lang="L"*>, where L is a language identifier. All language dependent page portions are embedded inside such a tag, while the remaining of the page is shared across languages. Thus, a set of Web pages written in different languages are migrated into a single MLHTML page, where language dependent parts are easily identified by the *ML* tag. The physical proximity of the multilingual information enclosed within *ML* tags helps keeping the content aligned over time. The HTML Web pages actually published by the Web site can be extracted from a single MLHTML page offline, once for all, or they can be generated dynamically by a server script on demand.

Existing Web sites can be migrated to MLHTML according to a process consisting of two steps:

- Page alignment
- Page merging

The first operation in this process, explained in detail in (Tonella, Ricca, Pianta & Girardi, 2002), aims at aligning the pages in the different languages. The outcome is a Web site where corresponding pages in different languages have exactly the same structure, with the same content translated into all supported languages. The second operation, i.e., the merging of multiple, aligned, HTML pages in different languages into a single MLHTML page, is the subject of this section.

The TXL transformation employed for page merging unifies a list of aligned pages into a single MLHTML page. The multilingual content is unified by embedding it into <*ML*> tags, while the XHTML code for the shared page structure is just kept. This transformation is obtained by means of the application, in sequence, of the following TXL functions/rules: *MLHTMLtransform, arrangeDoubleElements, arrangeMLblock* and *arrangeReferences*. For space reasons, the transformation will be described with reference to a bilingual site. However, its generalization to more than two languages is straightforward.

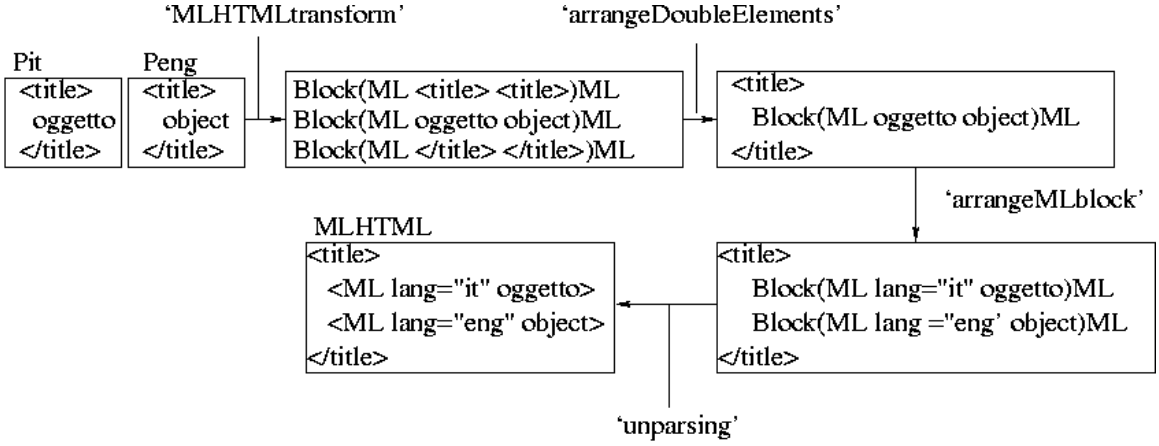

Fig. 7: *Two HTML pages (Pit and Peng) are merged into a single MLHTML page.*

An example of transformation, involving the pages *Pit* (Italian) and *Peng* (English) is shown in Fig. 7. The function *MLHTMLtransform* takes two pages (without *TagBlocks*) in input and produces a new page where each element is of type *TagBlock ML*. The *i*-th element of the resulting page is *Block(ML elP1_i elP2_i)ML* where *elP1_i* is the *i*-th element of the first input page and *elP2_i* is the *i*-th element of the second input page.

The rule *arrangeDoubleElements,* shown below, is applied to the result of *MLHTMLtransform* and transforms a *Block(ML elP1_i elP2_1)ML* into *elP1_i* only when *elP1_i* is equal to *elP2_i*. Since the page structure is shared across pages and only the content can change (when provided in different languages), the transformation applies only when *elP1_i* and *elP2_i* are either a same tag or a same text not translated in different languages (e.g., a proper name).

rule arrangeDoubleElements

replace [element]

Block('ML eTag [element] eTag)'ML

by

eTag

end rule

The rule *arrangeMLblock* splits each *TagBlock ML* into two *TagBlocks,* and adds the attribute *lang="L"*. The effect of this rule (with *e1* and *e2* two generic elements) is the following:

Block('ML e1 e2)'ML → Block('ML lang="language1" e1)'ML
Block('ML lang="language2" e2)'ML

The rule *arrangeReferences* (not shown here) transforms the URL references to objects (such as HTML pages, images, etc.). URLs are replicated for all supported languages, and are embedded inside *ML* tags, with a suffix *'?lang=L'*, where *L* is each supported language. This last transformation is used only in case the HTML pages to publish are generated dynamically by a server script. In this case, the selected language becomes a parameter to be handled dynamically. It is propagated from page to page as an extra URL parameter.

An example of resulting URLs is the following:

<ML lang="it"> <A HREF="program.ml?lang="it"> Programma </A> </ML>
<ML lang="eng"> <A HREF="program.ml?lang="eng"> Program </A> </ML>

When the selected language is English ("*eng*"), only the second anchor is included in the dynamically generated page by the server script. Such a hyperlink has a parameter *lang="eng",* which is used to filter the requested page (*program.ml*), so that only the *ML*

embedded portions with *lang="eng"* are extracted. The other page portions, not embedded within *ML* tags, are transmitted to the Web browser unfiltered.

RELATED WORKS

Quality assurance is considered a central concern by the design methods that have been proposed to support the development of Web applications (Conallen, 2000; Isakowitz, Kamis & Koufar, 1997). However, only a few works have so far considered the problems related to the analysis and restructuring of Web applications.

One of the first systematic studies on Web site analysis is (Warren, Boldyreff & Munro, 1999), where the evolution of the entities in Web sites is characterized by means of a set of metrics traced over time. In (Ricca & Tonella, 2001), several structural analyses, derived from those used with software systems and based on a Web application model, are proposed and employed for restructuring. An experience of Web site reengineering is described in (Antoniol, Canfora, Casazza & De Lucia, 2000), where the target representation of the reverse engineering phase is based on RMM (Isakowitz, Kamis & Koufar, 1997). The recovering process and the following re-design activity were conducted almost completely manually.

A tool for interactively creating and updating HTML documents, preserving the DTD compliance, is described in (Bonhomme & Roisin, 1996). This tool, called *Tamaya*, is based on a transformation language tailored for HTML documents (presented in the paper), and can be used to restructure existing HTML pages which are non DTD-compliant.

In (Graunke, Findler, Krishnamurthi & Felleisen, 2001) an automated transformation converts traditional interactive programs into Web applications based on CGI programs. The authors extend a programming language with primitives that support saving and restoring the interaction state across successive interactions.

The document by Chisholm and Kasday (2001), taken from the W3C Web site (www.w3c.org), provides information about evaluation, repair and transformation tools that make the Web more accessible (see the *Web Content Accessibility Guidelines* at the W3C Web site). Tools in this document are classified into three different categories: *evaluation tools* - performing a static analysis of pages or sites, focused on their accessibility, and returning a report or a rating; *repair tools* - once the accessibility concerns of a Web page or site have been identified, these tools assist the author in making the pages more accessible; *filter and transformation tools* - these tools assist Web users rather than authors to either modify a page or handle a support technology.

CONCLUSIONS

Development and evolution of Web applications is one of the most challenging tasks in software engineering. The fast change-rate of requirements makes Web applications subject to continuous maintenance interventions. If this is not accompanied by periodic restructuring, the quality degrades and in the end the software becomes unmanageable. However, restructuring is not affordable if not supported by automated tools.

In this chapter, the possibility to automate restructuring of Web applications has been investigated. The general framework - that of program transformations - requires the ability to parse the source code of a Web application, and this is a non trivial task, given the multiplicity of standards and languages involved. An approach to solve this problem was described, with examples given in the TXL program transformation language. Based upon the parse tree produced by the syntactic analysis of the Web application, several restructuring transformations can be applied. The instances described in this chapter include an example of *design restructuring* (namely, the automatic reorganization into frames), an example of *dynamic page construction* (based on the usage of a template and on the migration of variable information to a database), and an example of *content management* (in case of multilingual content). Program transformations have the potential to (partially) automate the complex process of Web application restructuring.

REFERENCES

Antoniol, G., & Canfora, G., & Casazza, G., & De Lucia, A. (2000). *Web Site Reengineering using RMM*. In proceedings of the Int. Workshop on Web Site Evolution, pages 9-16, Zurich, Switzerland, March.

Bonhomme, S., & Roisin, C. (1996). *Interactively Restructuring HTML Documents*. In proceedings of the 5th International World Wide Web Conference (WWW5), Paris, France, May.

Chisholm, W., & Kasday, L. (2001). *Evaluation, Repair and Transformation Tools for Web Content Accessibility.* W3C Report (www.w3c.org), 2001.

Conallen, J. (2000). *Building Web Applications with UML.* Addison-Wesley Publishing Company, Reading, MA, USA.

Cordy, J. R., & Dean, T. R., & Malton, A. J., & Schneider, K. A. (2002). Source Transformation in Software Engineering using the TXL Transformation System. *Information and Software Technology*, 44(13), 827-837.

Graunke, P., & Findler, R. B., & Krishnamurthi, S., & Felleisen, M. (2001). *Automatically Restructuring Programs for the Web.* In proceedings of the 16th International Conference on Automated Software Engineering, Paris, France, November.

Isakowitz, T., & Kamis, A., & Koufar, M. (1997). *Extending RMM: Russian Dolls and Hypertext.* In proceedings of HICSS-30.

Jeffries, R., & Anderson, A., & Hendrickson, C. (2000). *Extreme Programming Installed*. Addison-Wesley Publishing Company, Reading, MA, USA.

Ricca, F., & Tonella, P. (2001). *Understanding and Restructuring Web Sites with ReWeb.* IEEE MultiMedia, 8(2), 40-51.

Ricca, F., & Tonella, P., & Baxter, I. (2002). Web Application Transformations based on Rewrite Rules. *Information and Software Technology*, 44(13), 811-825.

Ricca, F., & Tonella, P. (2003). *Using Clustering to Support the Migration from Static to Dynamic Web Pages.* In proceedings of the International Workshop on Program Comprehension, (207-216), Portland, Oregon, USA, May.

Tonella, P., & Ricca, F., & Pianta, E., & Girardi, C. (2002). *Restructuring Multilingual Web Sites.* In proceedings of the International Conference on Software Maintenance, (290-299), Montreal, Canada, October.

Warren, P., & Boldyreff, C., & Munro, M. (1999). The Evolution of Websites. In proceedings of the International Workshop on Program Comprehension, (178-185), Pittsburgh, PA, USA, May.