Journal of Web Engineering, Vol. 4, No. 2 (2005) 144–164
© Rinton Press

TOWARDS ASSURING QUALITY ATTRIBUTES OF CLIENT DYNAMIC WEB APPLICATIONS: IDENTIFYING AND ADDRESSING THE CHALLENGES[a]

M. SH. AUN
Department of Information Engineering, Agusa Laboratory
Furo-cho, Chikusa-ku, Nagoya City, 464-8601, Japan
sharaf@agusa.nuie.nagoya-u.ac.jp

S. YUEN and K. AGUSA
Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya City, 464-8601, Japan
(yuen, agusa)@nuie.nagoya-u.ac.jp

Received July 11, 2004
Revised February 25, 2005

Web applications, nowadays, impose some entirely new challenges in the field of software quality. They differ from traditional software applications in several critical dimensions. This paper identifies the challenges involved in assuring quality attributes of Client Dynamic (CD) Web applications and then presents an approach proposed to address such challenges. Our approach, in addition to combining static and dynamic processing, involves feature engineering techniques. It allows for separating features out of the implementation artifacts and enables their debugging and conformance to quality attributes. The aim of this paper is to identify the challenges involved and to describe the set of components that incorporate the essential architecture design of an environment dedicated for addressing such challenges. We limit our attention to applications, where client scripted pages are considered as the building blocks. Scripted pages play an important role in making the web more interactive and dynamic. The effectiveness of our approach is illustrated (with the help of a prototype tool being implemented) by practical examples. The inherent advantages of our approach enables it to be helpful for assuring several other quality attributes such as maintainability and re-usability.

Keywords: Web engineering, features separation, Web analysis and debugging
Communicated by: Y Deshpande & S Murugesan

1 Introduction

To support the design phase of Web applications, several frameworks, architectures, and models, such as those described in [1] have already been designed and proposed. However, methodologies to support assuring quality attributes have not changed to cope with the design methodologies. Web applications, nowadays, impose some entirely new challenges in the field of software quality. A bug may cause a whole business to suffer and as Web-based systems grow more complex, a failure may propagate broad-based problems across many systems

[a]This research is partially supported by the Japanese Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (C)(2)#16500027. It is also partially supported by the 21st century COE program of Intelligent Multimedia Integration at Nagoya University.

[2],[3]. Moreover, developing such applications is usually accomplished in *ad-hoc* manner, which generally results in very poor quality products, with "spaghetti-like" [1] structures making them difficult to *debug, maintain and reuse.*

This work represents an on-going effort to develop a computer-aided environment that can support the construction of high quality CD Web applications. CD Web applications can be regarded as special case of document-based Web applications, where scripts or programs are allowed to be embedded within documents (e.g.; HTML), thus turning the formerly passive text into an application itself. Such applications by scripts and other enabling technologies are effective in view of the dynamicity and in making the web more interactive. They also fit well within the current shift [8] in the style of computation from process-oriented and object-oriented to document-oriented computation. This shift can be considered as a new paradigm as it will accelerate the way things are achieved. The possible uses [8],[9] of such applications range from controlling the content and the layout of a single document to support of coordination and collaboration techniques. In this paper, we identify the specific challenges involved in assuring quality attributes of CD Web applications over stand-alone applications and then present an approach to address such challenges. The paper also gives an idea about a tool being implemented within the context of the proposed approach. We limit our attention to applications, where client scripted pages are the building blocks. A subsequent paper will be devoted for the case, where server scripts are involved.

Assuring quality attributes of CD Web applications involves several extra challenges over assuring the quality attributes of stand-alone traditional applications. These challenges are due to the fact that Web applications differ from traditional software applications in several critical dimensions [5],[6]. Moreover issues like the coexistence of multiple technologies in an application, the immediate interpretation, the reflectiveness property of the languages used, the high volatility and the unpredictability of the Run-Time Environments (RTEs) used have lead to the fact that, trying to make a pure transposition of quality assurance techniques (like other techniques [7]) from software engineering to Web engineering both difficult and inadequate.

The aim of this work is to identify the challenges involved in assuring quality attributes of CD Web applications and to describe the set of components that incorporate the essential architecture design of an environment dedicated for addressing such challenges. Our approach involves four categories of techniques. These are: *separating features techniques, preprocessing techniques, dynamic processing techniques, and validating interface techniques.* In one of our previous works [11], we have shown how combining preprocessing and dynamic processing can help a great deal in the debugging process. In this work, we mainly focus on the first category giving the necessary description of others for the purpose of completeness. We enhance our previous work [12] by showing the detail techniques involved. Our approach starts by representing the source code with a graph that can enable a feature to be separated and then shows how useful (e.g.; help in assuring quality) features can be identified. Since quality attributes are diverse [4], we restrict our attention to supporting debugging, which represents a core activity for maintainability quality attributes.

The effectiveness of our approach is illustrated by practical examples. It is found that, separating features helps a great deal in the debugging process. It throws away features' irrelevant code, bringing functionality that reduces the amount of time spent on the debugging

process. The inherent advantages of the approach enable it to be helpful for assuring several other quality attributes such as understandability, maintainability and re-usability. Once a feature is separated, it can help in understanding, maintaining the overall system and it can be reused.

The rest of this paper is organized as follows: Section 2 presents the specific properties of CD Web applications. Section 3 identifies the challenges involved in assuring the quality attributes of such applications. In Section 4, our approach for addressing the challenges is described. Section 5 discusses the characteristics of our approach and Section 6 compares it with other related works. Finally the paper is concluded with the future directions.

2 Properties of CD Web Applications

The specific properties of CD Web applications are the followings:

- *Coexistence of multiple technologies*: the source code of CD Web applications usually involves a variety of components which may be realized with different categories of languages. Applications may contain scripts, applets and other client enabling technologies. They are highly heterogeneous systems.

- *Immediate interpretation*: immediate interpretation is a well known property in the Web. Consequently, traditional quality assurance techniques (e.g.; faults identification) can not be used since they are mainly based on or integrated to the compilers.

- *Interactive and dynamic*: at the client, applications may contain scripts that define additional dynamic behavior and often interact with the browser, page content and additional controls such as Applets and plug-ins.

- *Event-driven applications*: unlike structural programs, Web applications are typically event-driven programs. They can respond to user interactions. This makes the portion of the program executed next to be specified by an event.

- *Reflective*: a key feature of web client applications is that they are reflective. A program can change itself during execution or generate a new object, program or script on-the-fly.

- *Manipulate environment objects*: another key features of client applications is that, most of the objects they manipulate are environment objects.

- *Code organization*: in addition to be heterogeneous, the source code of such applications can be scattered over and generated at several layers of the Web architecture.

All such properties, in addition to those related to the RTEs and the languages used, can contribute in a way or another in imposing new challenges and requirements.

3 Challenges

Not surprisingly, CD Web applications are of increasing complexity in terms of development, and assuring their quality attributes. Assuring quality attributes of such applications involves several challenges over assuring traditional stand-alone applications. These challenges may include the following:

- *The need for separating features*: the methodology has to be capable of separating useful features out of the implementation artifacts dealing with the complexity introduced by the coexistence of multiple technologies in one application. Separating features refers to the ability to identify and manipulate those parts of the software code which are relevant to a particular concept, task or goal. It has been under several studies in the context of non-web applications showing promise in several aspects [10]. Based on our experience [11]-[14], we believe that separating features to be one of the most suitable mechanisms for facilitating the assurance of several quality attributes. It can provide opportunities for better analysis that can facilitate testing, debugging and conformance to quality attributes.

- *The need for filling the analysis gap due to the lack of compilation*: the methodology has to be able to fill the analysis gap due to the lack of compilation and the type-less property of the languages used. Client enabling components are usually immediate interpreted. Immediate interpretation means, traditional compiling techniques are not available resulting in lack of type checking techniques, for example. This also means that bugs or other failures are only noticed during run-time. Moreover, Web scripting languages, like other scripting languages [15], are usually type-less. This implies that all variables look and behave the same so that they are interchangeable. Such feature has penalty: at time we think a variable or an expression has a certain type or data in it, when in truth, something entirely different in there.

- *The need for managing the reflective and dynamic properties*: the methodology has to be capable of managing the challenges introduced by the reflective and dynamic properties of the applications. Due to the first property, a program can change itself or generate new ones making the state of the program to be a combination of both the global variables and the program itself. Web RTEs also are much more volatile as web changes tremendously over the course of few milliseconds. More concurrent activities are involved on the web. The more dynamic the application is, the more challenges in assuring its quality.

- *The need for providing execution controlling mechanisms*: the methodology has to provide some mechanisms for controlling program execution so that the developer can investigate the state of the program at certain points during execution. Such mechanisms has to be platform-independent.

- *The need for coordinating and cooperating activities with the surrounding environments*: another specific and very important requirement is that, the methodology should be able to cooperate with its surrounding environments such as the browsers and be able to coordinate it activities with them. Scripting objects, as an example, are mainly environmental objects rather than language created objects. Thus, any failure due to manipulating them can not be treated far from the environment which creates such objects.

In addition, the methodology has to be capable of reorganizing (or collecting) the source code, which can be scattered over and generated at several layers of the Web architecture. For the purpose of this paper, we assume the availability of the source code.

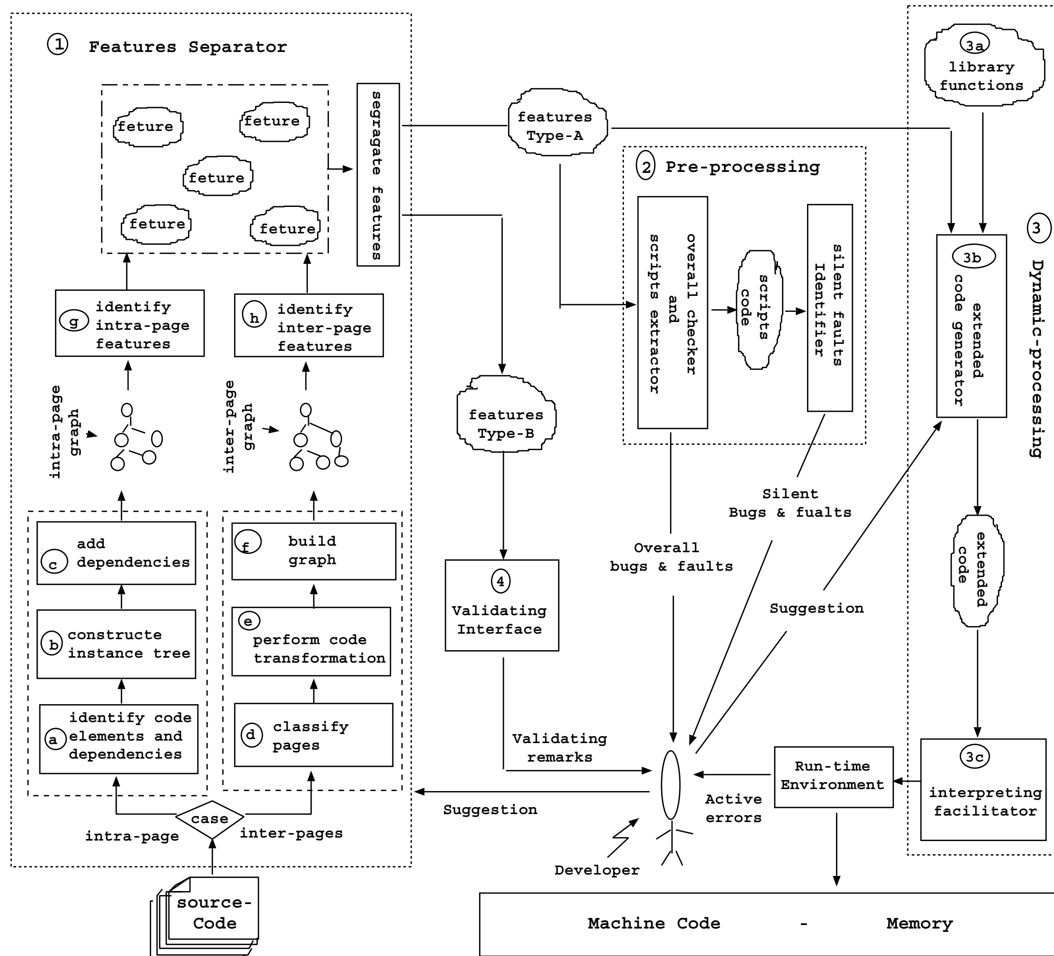

Fig. 1. Graphical view of the approach

4 The Approach

Having introduced the specific characteristics CD Web applications and the resulting challenges in assuring their quality, we now proceed to describe our approach to address such challenges. Fig.1 shows the graphical view of the approach. Our approach involves four main complementary and interacting components, and will operate with the guidance from the developer. The components are: *features separator component, preprocessing component, dynamic processing component, and validating interface component.* Each component has its own purpose and function as described next. In this work, we mainly focus on the first category giving the necessary description of the second and the third with the help of a prototype tool being implemented. As for the fourth, we give its outline for the purpose of completeness. More description related to preprocessing and dynamic processing can be found in one of our previous works [11].

4.1 Features Separator

This subsystem is responsible for identifying and separating useful features out of the implementation artifacts of an application. We propose that separating features to be one of the most suitable mechanisms for facilitating the assurance of several quality attributes. Once a feature is separated, it can be reused, debugged validated, etc. The input is the source code consisting of the page or the collection of pages that identify the functionality of the application. The output are two types. The first type (type-A) is those features which contain dynamic-enabling components such as scripts together with their related rendering components. The second type (type-B) of features are those with no scripts. This categorization is necessary as those features containing dynamic components must be treated differently. Details related to how features are identified and separated is presented next after defining and categorizing features.

4.1.1 Features in CD Web Applications

As a result of the specific properties of CD Web applications, there are two main categories of features. These are: (1) features tangled in a page and (2) features tangled across pages. We will refer to the first category as *intra-page* features and to the second category as *inter-page* features. Since feature engineering [10] is still in its infancy, there is not definition of what constitute a feature. In this paper, we adapt the following definition:

Definition*: A feature in a CD Web application can be defined as the subset of the code tangled in a page or across pages provided that it refers to a particular functionality.*

A feature, therefore, concerns with the computation of an embedded component within or across pages rather than the computation of a single variable or set of variables at some point of the program as in a program slice in non web applications. An example of a feature is a script that define a certain dynamic behavior together with its related rendering components.

4.1.2 Case-1: Identifying Intra-page Features

Identifying features tangled in a page involves two main steps. The first step (subcomponents *a, b, c* of Fig.1) is to represent the source code in a graph that exhibits the code elements and the relation among them. We call such graph *intra-page graph.* The second step is to identify intra-page features by processing the resulted graph.

Constructing Intra-page Graph

The first step toward constructing the graph is to determine the code elements of the code under consideration and the relations among such elements. We treat a page as a collection of tagged regions among which there are three kind of dependencies. These are *nest, data, and call* dependencies. A nest dependency holds between two tagged regions if the second is nested in the first. A data dependency holds between two tagged regions if the first defines a certain variable and the second region uses it. A call dependency holds between two tagged regions if the first call the second.

Definition: For a program $P = (V,E)$ with V as the set of tagged regions and E as the set of nest, data and call dependencies; we define an intra-page graph as follow: *Intra-page Graph* $= \sum(V_i, E_i)$ so that: $V_i \in V$, which is a tagged region and $E_i \in E$, which is a nest, data or call dependency as described above.

```
1 - <html>
2 - <head><title>Test Example</title></head>
3 - <body>
4 - <p> This example is written for the purpose of illustrating
        the main concepts behind the approach.</p>

     .....................
      some other code
     .....................

50- <form name="inform">
60- <input type=text name="inputText">
70- <input type=submit value=" click here " name="Subtbutton">
80- </form>
90- <script>
100- function openWin(m){
110- win1 = window.open("","win1","status,width=100,height=100");
120- var d=win1.document;
130- d.write(m);
140- d.close();
150- }
160- var mes=document.inform.inputText.value;
170- document.inform.Subtbutton.onclick=new Function("openWin(mes)");
180- </script>
190- <script>
     .................
     some other script code
     .................
390- </script>
400- </body>
410- </html>
```

Fig. 2. Case-1: source code

The graph is constructed so that each tagged region is an instance tree or a forest of trees, and each nested node tag is a child node. Then we add data and call dependencies. Thus, unlike other type of instance [24] tree and the Document Object Model (DOM) parsing tree, we involve data and call dependencies. Data and call dependencies play an important role in identifying features. They provide partial explanation for the inclusion of specific code element into a feature. The output, therefore, is a special type of dependency graph that exhibits tagged regions and the dependencies (i.e.; nest, data and call) among such regions.

An example of intra-page graph is as shown in Fig.3, which is constructed based on the source code presented in Fig.2. The source code contains script components, rendering components and some text. The dot lines before the first script and within the second script are of any code. We removed the actual code for space and simplicity purpose. This example represents the condition where multiple features are encapsulated in one page. The first script and its related rendering components represent a feature. The existence of other features depends on the content where the dot lines appear. The first script is written in Java-Script and it is embedded in HTML document. It has one function which open a new window dynamically and it is supposed to write the parameter m to the dynamically created window. However there is a **bug** in the script. When it is executed, it opens the required dynamic new window but no output appears on that window. The feature containing the bug is of type-A. We need to separate such feature and debug it.

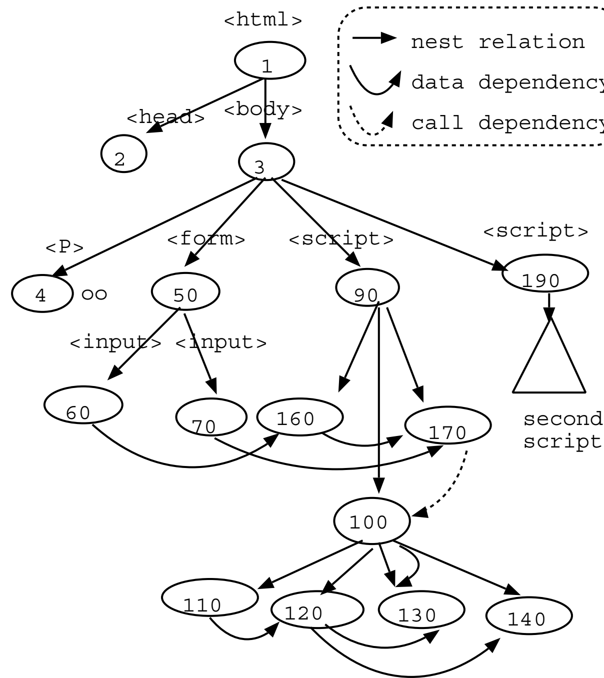

Fig. 3. Case-1: Intra-page graph

Identifying Intra-page Features

After constructing the intra-page graph, we use *level-based* technique to deal with scripting nodes and *heuristic* means to treat non-scripting nodes to build a feature. Level-based implies that: all nodes below a certain node, like ⟨*script*⟩, are grouped together to form a subset of the feature. Heuristic means that the node under consideration and all of its parents nodes found in the path to the root are also added. Finally, all of the remaining nodes having either data or call dependency to any of the nodes in the feature under construction are also added to form the required feature.

Applying this to our example, we start by searching for the node representing ⟨*script*⟩ tag. It is node with label 90. Then, all nodes below it (i.e.; nodes 160, 100 with its childes and 170) are grouped together to construct sub-part of the feature. Now we use *heuristic* means to treat non-scripting nodes. This comes out with nodes 90, 3, and 1. Finally, since there are dependency lines from nodes labeled ⟨*input*⟩ (i.e.; nodes 60 and 70) to the feature under construction, they are also added with their parent constructing the requires feature. This process adds nodes with labels 50,60 and 70. As for control nodes that indicate closing the tagged regions, such as nodes labeled 80, 390, etc they are added automatically based on the existence of their associated starting tags. This is why they don't have to appear in the graph.

The output of the process is as illustrated in Fig.4. It contains subset of the code that is related the the dynamic behavior defined by the script together with its related rendering components. It is a complete feature that comprised of code supporting the debugging process

of such applications.

```
1 - <html>
3 - <body>
50- <form name="inform">
60- <input type=text name="inputText">
70- <input type=submit value=" click here " name="Subtbutton">
80- </form>
90- <script>
100- function openWin(m){
110- win1 = window.open("","win1","status,width=100,height=100");
120- var d=win1.document;
130- d.write(m);
140- d.close();
150- }
160- var mes=document.inform.inputText.value;
170- document.inform.Subtbutton.onclick=new Function("openWin(mes)");
390- </script>
400- </body>
410- </html>
```

Fig. 4. A feature of type-A

By repeating the above process, we can find all features of type-A. Another feature of this type is the one involving the script starting at node labeled 90. As for type-B features, Fig.5 shows an example. It contains non-script nodes. It contains no script code. Such kind of features are mainly for validating purposes.

```
 1 - <html>
 2 - <head><title>Test Windows</title></head>
 3 - <body>
 4 - <p> This example is written for the purpose of illustrating
        the main concepts behind the approach. </p>
 50- <form name="inform">
 60- <input type=text name="inputText">
 70- <input type=submit value=" click here " name="Subtbutton">
 80- </form>
 90- <Script>
390- </script>
400- </body>
410- </html>
```

Fig. 5. A feature of type-B

The output of the process (i.e.; features type-A and type-B) are now easier to deal with in term of testing, debugging, reuse, etc. Debugging the script found in feature shown in Fig.4 can be accomplished as described in Section 4.4. Conformance to W3C recommendations of the HTML components (i.e.; feature type-B) can be achieved easily with a validator like the one available at [17]. Such tool is freely available to be used. Component “4”is to work as an interface to such tool. As for testing, testing sets can be organized around the feature they are intended to test rather than traditionally organized.

4.1.3 Case-2: Identifying Features Tangled Across Pages

For the purpose of simplicity, we will explain the underlying concepts with a running practical example. We will consider a practical Web application for a company that export used cars. The cars are of two categories and five types. The first category of cars are those manufactured within the last five years and the second category are manufactured before five years. The type of cars are Toyota, Mazda, Mitsubishi, Nissan and Honda.

The Application

Fig.6 shows the simplified application. In this application, a customer can select one of the two main categories of cars. After selecting a category from the main page, a list of the available cars of such category is shown. The customer then can select the type of the car. By clicking the type, a full description of his choice is shown including the price, the model, etc. Here the user can either go ahead and order a car or return back to select another option. In case he pressed order, an order form is opened where he can insert his name, address, e-mail, etc. By choosing submit in the order form, a full description of the car selected and the data entered is generated for the purpose of confirmation. Such form is then transfered to the sales department of the company for further processing.

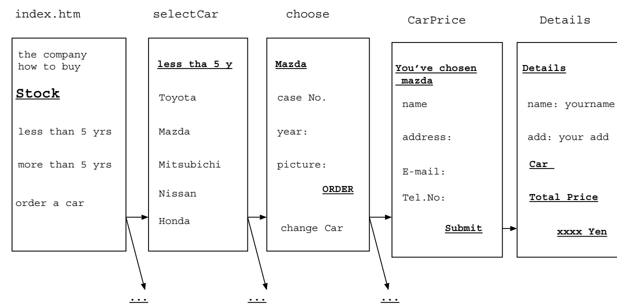

Fig. 6. Case-2: The simplified application

This case represents the condition where features are tangled across several pages. An example of such features is the code related to the following task: *"select and order a Mazda car that was manufactured within the last five years"*. This features spans over many pages that have to be generated and/or presented to the user. The next two sub-sections describe how such feature can be separated.

Constructing Inter-page Graph

Constructing inter-page graph involves thee steps, subcomponents *d, e and f* shown in Fig.1. The first step is to classify the pages composing the application. One way to solve this problem is is to classify pages according to various criteria. Several approach in this direction have been proposed, but mainly they serve textual properties of web document. However, Web pages under consideration are different from textual documents in that they contain URL, HTML tags and embedded code within pages.

For the purpose of separating features, we found that a proper classification should be based first on application architecture (e.g.; pages and structural links, see below) and then on page structure (e.g.; HTML formalities). For the purpose of maintenance, a related work [21] has also classified web pages into two categories: unique and non-unique. Unique is in the sense that no other page has the same structure.

Classifying the pages, based on the application architecture, in view of a tree, is to be accomplished first. In our case, the index page is the root, the lists of available cars (less than and more than than 5 years) are the intermediate pages. Pages that contain information for each available car represent the leaf pages. The root page has no parent. The intermediate

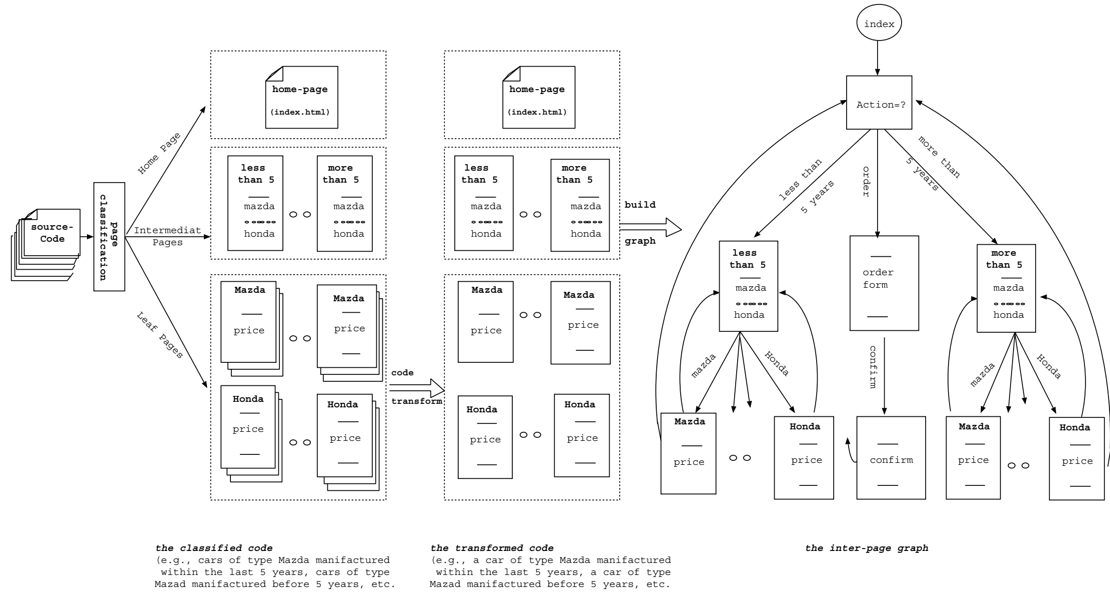

Fig. 7. Case-2: Inter-page Graph

page has both parents and childes. The leaf page has no childes. The second step is to classify the leaf pages based on their structure. This step segregates pages of each type (e.g.; Toyota, Mazda, Honda, etc.). The output of page classification is as shown in Fig.7 (left).

Then, we transform the code by representing each equivalence subclass of pages with its representative. Code transformation is necessary so that, instead of dealing with individual pages composing the application, whose number can be very big, we need only to deal with less number of pages. Equivalence subclasses of pages is likely to occur within each identical leaf pages. In our case, pages which represent Mazda cars are identical and same is for Toyota and so on. The result of this task can be expressed as shown in Fig.7 (middle part).

Definition: For an application $P = (V,E)$ with V as the set of transformed pages and E as the set structural links, we define an inter-page graph as follow: *Inter-page Graph* $= \sum(V_i, E_i)$ so that: $V_i \in V$, which is a page and $E_i \in E$, which is a structural link.

Structural links [25], unlike free links, can express that a page is a child, a sibling or the parent of another page. Free links, instead, are simply references to other documents. They are usually associated with icons or labels that indicates the semantic to the end user.

Finally, the graph is constructed. The graph is created by the selection of vertices (pages) and edges (structural links) from the transformed code. The home-page represent the root, and the leaf pages represent the leaf. Thus, unlike the above case, in this case each node of the graph represents a page rather than a tagged region and each linked page by structural link in forward manner is a child node. The result of this task can be expressed as shown in Fig.7 (right part). In such figure, many links and details are omitted for the purpose of simplicity.

Identifying the Features

Starting with the inter-page graph, separating features is accomplished in two steps: overall features are identified with the help of the graph. Every path starting at the root node and ending at the leaf node is likely to represent a feature. Identifying further features is then accomplished by decomposing the code represented by the sub-graph into code fragments as explained in case-1 presented above.

Accordingly, identifying the above mentioned feature (i.e.; *"select and order a Mazda car that was manufactured within the last five years"*) becomes as expressed in Fig.8. As we can see, this feature is still an application after throwing away portions of the code that is irrelevant. It contains the generated order for together with a representation of the Mazda car category. For the benefit gained from separating such feature, see the explanation provided in Section 5.1, while explaining the feasibility of the approach.

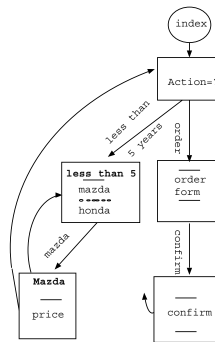

Fig. 8. A feature for selecting and ordering a Mazda car

4.2 Preprocessing

Preprocessing (unlike dynamic processing) refers to the process of analyzing the source code without being executed. In spite of the dynamic nature of client applications, we found that substantial part of the conventional static analysis techniques are necessary. Preprocessing is mainly necessary for filling the analysis gap due to the lack of compilation. It identifies both overall and static (or silent) faults [11] that can be found in type-A features. Overall faults refer to those faults found outside the script tagged regions. They may exist in a form or other tagged areas. Silent bugs refer to those bugs which causes the RTE not to run the feature. They make the browser neither to run the script found nor to produce any helpful error message. An example of such kind of bugs is the case when using a reserved word like *case* as a function name in a JavaScript program.

This subsystem involves two subcomponents:

- *Overall Checker and Scripts Extractor*: Features that contain dynamically enabling components represent the input to this subsystem where they are scanned for overall bugs and the dynamic-enabling components are extracted for further processing. The source code of the whole feature is read, overall checked, and the embodied dynamic enabling components (scripts) are extracted. The output is the script components in

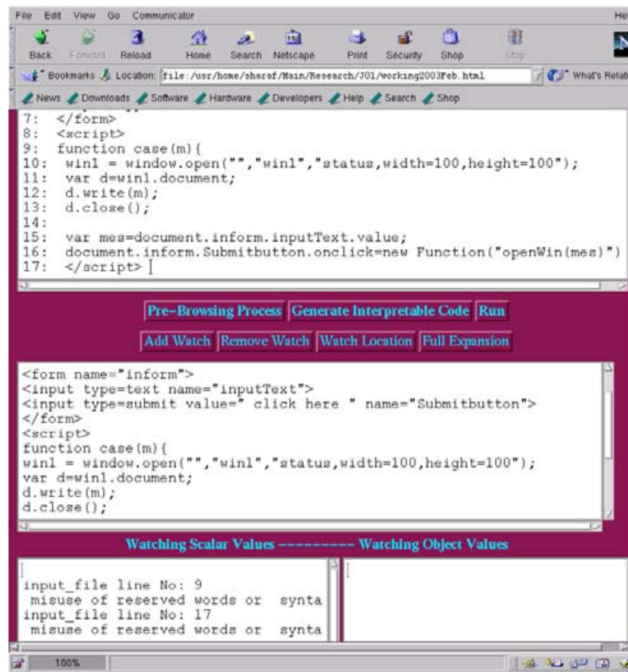

Fig. 9. Example of identifying silent bugs

addition to overall bugs identified in a feature. Compared to available checking tools such as weblint [16], this subsystem have also to extract the embodied component found for further processing.

- *Silent Faults Identifier*: This component concentrates on eliminating silent faults that cause the RTE neither to run the application nor to produce any helpful error messages. It involves a scanner that fills scripts analysis gap due to the lack of compilation, bringing functionalities for identifying the root cause of the failures that make a script not to run. It involves a lexer and a parser for the language used for developing scripts. The lexer generates tokens and the parser check them according to the language grammar. The input of this component is the scripts extracted by the previous component and the output is the silent bugs identification.

Based on our experimental tool implementations, Fig.9 gives us an idea about how silent bugs can be identified. In this figure, the input file shown in the middle window of the figure has two silent bugs each can cause every thing to freeze while trying to execute script directly. The first bug is the use of the reserved word *case* as a function name. The second bug is due to the unclosed block found in the function. There is a block start indicator (i.e. { in line No. 9) but such opening block is not closed any where within the script.

Both existing bugs are captured as the lower window of the figure indicated. The line numbers, where the bugs exist are also indicated. This makes the developer task easier in correcting the source code.

4.3 Dynamic Processing

Dynamic processing (unlike preprocessing) refers to the process of analyzing the source code while being executed. Dynamic processing serves several purposes. It identifies and eliminates active or behavioral bugs, provides the necessary mechanisms for controlling program execution while debugging, and helps in overcoming the challenges due to the dynamic property of the applications. Behavioral bugs refer to those bugs, which cause the execution of a script to stop before completion or the execution is completed but unexpected or undesirable results are produced. This component involves three subcomponents. These are: library debugging and testing functions, extended code generator, and interpretation facilitator.

- *Library Functions*: Dynamic enabling components (or scripts) mainly focus on manipulating pages or documents they work within. The documents are treated and manipulated as objects, possessing both data and behavior. The collaboration of objects represents the structure of the document. So based on the Document Object Model (DOM) used and the required object hierarchy required while debugging, this part may contain a variety of library functions to be used as required by the developer. The functions must include specialized code for inspecting both objects and scalar variables.
- *Extended Code Generator*: The purpose of this subcomponent is to perform the source code instrumentation [26]. Instrumentation is the process of inserting additional statements into a program for the purpose of gathering information about its dynamic behavior. based on the developer suggestions, instrumentation is accomplished by expanding the code under consideration automatically. The inputs to this part are the source-code of the features containing scripts, the required debugging functions and the developer options (suggestions). The source code is to be expanded and the debugging functions to be augmented. Developer options are necessary for providing more flexible debugging options. The output of this part is the extended code which is the original source code plus the required functionalities that facilitate the debugging. The RTE, therefore, is controlled from within the application itself rather than by external components.
- *Interpreting Facilitator:* This part is necessary for facilitating the execution of the applications directly from the context of the environment. This is the case as it can make the execution of the code under the control, which is one of the requirements to give the developer the ability to investigate the state of the program under consideration. It works as an interface between the extended code and the interpreter.

An example for showing the effectiveness of process in eliminating active or behavioral bugs is shown Fig.10. The input, shown in the middle part of the figure, is the feature presented Fig.4. This feature contains one script written in JavaScript and it is embedded in HTML document. The script has one function which opens a new window dynamically and it is supposed to write the parameter m to the created window. However, there is a bug in the script. When it is executed, it opens the required dynamic new window but no output appears on the opened window.

While debugging such feature, we can notice the following: the extended (processed) code is in the upper-most window and the results of the debugging process are shown at the lower part of the same figure. The library function has been inserted as script before the source

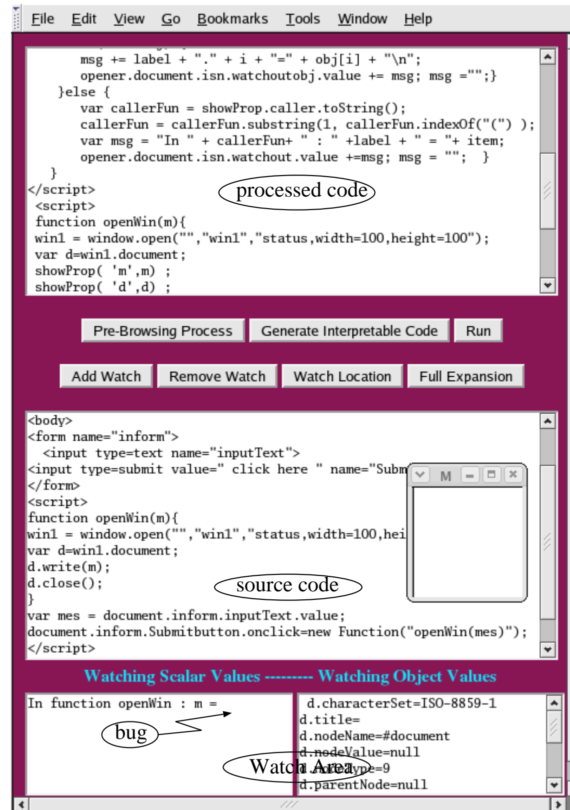

Fig. 10. Results sample while debugging a feature that contains a dynamic-enabling component

code of the feature and then such library function is called as necessary. The value of the variable m (shown at the bottom-left window) is null indicating the bug.

Properties of the object d are also written to the bottom-right window. In another word, the values of scalar type variables and the properties of object type items, each appeared in a different window. The distinction between scalar type values and object type items is an important feature as it helps a great deal in overcoming the difficulties introduced by the type-less property of the scripting languages used used for developing the applications.

4.4 Validating Interface

Details related to validating functionalities is outside the scope of this paper. In summary, this component passes the source code of features (type-B) through a special program that compares the code against standards-based measures. It checks syntactic accuracy, structural integrity and conformance to standard requirements. With separating of features, many exiting and emerging tools by W3C and others can be utilized for assuring conformance to standards. Nowadays, for example, there are several available tools like[17] by W3C to check HTML documents, others like[18] to make them tidy and so on.

Challenges ↓	Comp 1	Comp 2	Comp - 3			Comp 4
			3a	3b	3c	
the need for separating features	O					
the need for filling the analysis gap		O				O
the need for managing the reflective and dynamic properties			O	O	O	
the need for providing execution controlling mechnisms			O	O		
the need for coordinating and cooperating activty with others environments			O	O	O	

Components (column headings)

Comp. 1: features separator; comp. 2: preprocessing component;
compont 3: (a) library functions, (b) extended code genrator
(c) interpreting facilitator; comp 4: validating interface component

Fig. 11. Addressing the challenges

5 Discussions

5.1 Addressing the Challenges

The new proposed approach involves four main components as illustrated in Fig.1. Each of the components has its own purpose and function. By looking closely at these components, we can find that they, collectively, fulfill (Fig.11) the specific requirements and address the challenges presented in Section 3.

The first component (or features separator) is responsible for dealing with challenge-1. It enables a feature to be separated overcoming the difficulties introduced by the fact that on the Web multiple technologies have to coexist in one application. It can play an important role in overcoming the challenges represented by the "spaghetti-like" [1] structures of available web applications. Separating features out of the implementation artifacts helps a great deal the debugging process. It throws away irrelevant code, bringing functionality that reduces the amount of time spent on the debugging process. Details related the debugging process itself can be found in our previous work published elsewhere [11].

The second component (or the preprocessing) mainly responsible for making overall check of the features containing scripts and eliminates overall and silent faults that causes the RTE not to run the feature. Together with the validation component, it can fill the analysis gap due to the lack of compilation.

The third component (dynamic processing) which consists of three sub-components serves several purposes. It provides the necessary mechanism for controlling program execution, facilitates the execution of the applications from within the context of the new environment, and helps in managing the difficulties, which arise due to the reflectiveness and dynamic

properties of the applications. Finally, the fourth component works as an interface with validating tools. Validation of features type-B can be performed from different point of views such as conformance to W3C standards recommendations.

The approach, therefore, combines static (preprocessing) and dynamic processing together with features engineering techniques. In spite of the dynamic nature of client applications, we found that substantial part of the conventional static analysis techniques are necessary. They fulfill analysis gap due to the lack of compilation, bringing functionalities for identifying the root cause of the failures that make a script not to run.

Combining both static and dynamic capabilities can point out a new direction for building new tools for assuring quality applications as it has several advantages. Dynamicity, which is mainly caused by scripts components is managed as the subsystems can cooperate their activities with the RTE. The approach can augment the required functionalities for assuring the quality attributes to the components themselves. This allows them to be uploaded by the interpreting facilitator without the need for extra mechanisms or techniques.

Involving separating features techniques makes the approach capable to deal with a wide range of applications. Involving separating features techniques have also several inherentadvantages, See Section 5.2.

The approach is feasible. It was practically applied successfully to the case study, presented in Section 4.1.3, where we were able to isolate a bug that makes every car order to get lost and never arrive the sale department of the car exporting company. The reason for that, was as follow: the *mailto* tag in the order form was altered somehow and make maintaining such system problematic. The company lost several customers since no orders arrived any more. With separating the feature above, the bug was easily located and debugged. By fixing such bug, orders related to all type of cars were correctly received afterward saving the company a big amount of money and time.

5.2 InherentAdvantages

Despite of the primary benefit from involving separating features techniques for facilitating the debugging process of client applications, we found that the approach helps a great deal in enabling assuring several quality attributes. These include understandability and maintainability. Once a feature is separated, it can be reused or it can help in understanding and maintaining the overall system.

Moreover, with separating features, many exiting and emerging tools can be utilized for assuring high quality systems. Nowadays, for example, there are several available tools by W3C, like [17], and others like the one described in [18] to check HTML components. Using available tools for checking HTML components give us the opportunity to concentrate on others. Also emerging and future tools can also be utilized.

Furthermore, the approach can be of a great help in supporting both forward and reverse engineering [10]. Supporting reverse engineering is an important issue as Web applications are tomorrow's legacy systems [27]. This is the case as the approach can provides opportunities for better analysis.

Finally we can say that, our approach fits well in the current shift in the style of computation from process-oriented and object-oriented to document-oriented computation. As indicated above, such computation has a variety [8],[9] of uses.

6 Related Work

There does not seem to be any work in the academic literature on quality assuring techniques targeting the dynamic behavior of the client applications. The software engineering literature includes some works on assuring quality, but it is generally aimed for non web or tradition software systems rather than for web applications which differ from traditional software systems in several dimensions. In the web engineering literatures, there is no paper yet investigating assuring quality of CD Web applications. In fact a research has shown that "no automated support exists yet for anything more than basic HTML editing and barely any CASE tool supports advance Web-based applications development" [22].

However, the followings works can be considered as related: A paper by the ⟨*Bigwig*⟩ project team (Claus Brabrand et al.) [19], addresses the static validation of dynamic generated HTML in the context of the ⟨*Bigwig*⟩ language. While the aim of their work was at validating HTML components in the context of specific language, our work aim is at addressing the challenges that are involved in assuring the quality attributes of CD Web applications focusing on scripts as they are the main reason behind making the web more dynamic.

Another paper (tailored to supporting maintenance) by P. Atzeni et al. [21], has proposed a model that can support maintenance as well as the design phase of Web applications. Another paper (tailored to architecture recovery) by A. E. Hassan et al. [27], have proposed an approach for recovering the architecture of Web applications on the windows platform. Jonas Boustedt [29] has also made some efforts in trying to find a suitable criteria for classifying equivalence pages or nodes for the server responses in Web services with the help of spider. These efforts, however, in addition to be tailored to specific tasks, have considered the client as a pure presentation layer. Our work differs from such efforts in presenting a new approach (through separating of features) for facilitating better analysis that can overcome many challenges including the complexity introduced when the client works beyond the presentation layer and it fits well in assuring several quality attributes.

Recently, some browser vendors claim the provision of tools for debugging scripts. An example for that, is the JavaScript debugger [30]. However, such tool works only with navigator, limited to JavaScript, causes frequent RTE freezing and makes the RTE unavailable for any purpose except debugging in the debugging mode. Another tool is by Microsoft; Script debugger[31]. This tool, however, works only with Internet Explorer, and it is limited to the windows platform.

Our approach differs from such tools in several aspects. First, It involves features separation techniques in addition to combining both static and dynamic capabilities. Separating features has many inherentadvantages. Moreover, our approach scales well to debugging server-side scripts. Most of the techniques provided by the approach can be adapted and applied to the later case. An example for that is source code instrumentation technique. It can be extended because it is accomplished by source code transformation rather than by binary code, for example. We foresee to achieve more scalability and extendability of our approach. While the prototype tool implementation is specific to Javascript, that is not necessary the case. The language can be any other scripting language. Furthermore, our approach is portable as it is not tied to any language or environment. Finally, we can say that our work is oriented toward developing a computer aided environment rather than to added a certain functionality to some propriety browser.

7 Conclusions and Future Work

The main aim of this paper is to identify the challenges involved in assuring quality attributes of CD Web applications and to describe the set of components that incorporate the essential architecture design of an environment dedicated for addressing such challenges. We believe that, we have achieved that aim. It was found that, several challenges are involved in assuring quality attributes of CD Web applications. These challenges include: the need for separating features out of the implementation artifacts, the need for fulfilling the analysis gap due to the lack of compilation, the need for managing the challenges introduced by the dynamic property of the applications, the need for providing program execution controlling mechanisms, and the need for coordinating and cooperating activities with the surrounding environments.

To address such challenges, our approach combines static (preprocessing) and dynamic processing together with features engineering techniques. Preprocessing is necessary for filling the analysis gap due to the immediate interpretation. Dynamic processing serves several purposes. It provides the necessary mechanisms for controlling program execution, and deal with the dynamicity and the reflectiveness properties. By expanding the source code with the required testing and debugging functionalities, we can avoid many problems which may arise due to controlling the RTE through external components. This also allows them to be uploaded by the interpreting facilitator without the need for extra mechanisms or techniques.

For overcoming the challenges introduced by coexistence of multiple technologies in one applications and to deal with a wide range of applications, the approach involves features separating techniques. We gave special focus on these techniques as they can be regarded as the most suitable mechanisms for facilitating quality assurance of Web applications in general and CD Web applications in particular. They have several inherentadvantages. To determining intra-page features (or features tangled in a page), the key idea is to treat a web page as a collection of tagged regions rather than treating it as a single entity. Among these regions are nest, data and call dependencies. Once these regions and dependencies are determined, a graph with nodes representing the regions and edges representing the dependencies is constructed. We call such graph intra-page graph. After constructing such graph, we apply level-based technique for treating scripts components and heuristic means to treat non-scripting nodes to build features. For determining inter-page features (or features tangled across multiple pages), representing the source code in a graph is recognized to be helpful. Before building the graph, pages composing the application are first classified and transformed. Code transformation is necessary so that, instead of dealing with individual pages composing the application, whose number can be very big, we need only to deal with less number of pages. The graph is then built by the selection of pages as vertices and structural links as edges. In contrast to the previous case, each node represents a page rather than tagged region, and each page linked with structural link in forward manner is a chilled node. We call this type of graph as inter-page graph. After constructing the graph, every path starting at the root and finishing at a leaf node is likely to represent a feature.

In spite of the dynamic nature of CD Web applications, we found that substantial part of the conventional static analysis techniques are necessary. Preprocessing is mainly necessary for eliminating faults that can cause the application not to run. Finally, dynamic processing can serve several purposes. It provides the necessary mechanism for controlling program execution, facilitates the execution of the applications from within the context of the new

environment, and helps in managing the difficulties, which arise due to the reflectiveness and dynamic properties of the applications.

This work highlights several interesting research issues in relation to developing and assuring quality attributes of Web applications. We plan to extend it in several different ways. The focus of this paper was at identifying the challenges involved in assuring the quality attributes of CD Web applications and on the architecture design of an environment which can address such challenges. We focused mainly on separating features out of the implementation artifact. Assuring quality was primarily performed through facilitating debugging and conformance to standards. Although there are many other quality attributes to assure (such as understandability, maintainability, and so on), the inherent advantages of our approach enables it to be target-able for assuring other quality attributes. Once a feature is separated, it can help in understanding, maintaining the overall system or it can be reused. More implementation will be accomplished specially to handle features separating techniques. After that, we will consider the case, where server-side scripts are involved.

Acknowledgments

The authors would like to thank S. Yamamoto (associate professor in Aichi Prefectural University), T. Hamaguchi (research associate in Agusa Lab.) and all members of our Laboratory for their comments.

References

1. G. Costagliola, F. Ferrucci and R. Francese (2002), "Web Engineering: Models and methodologies for the design of hypermedia applications", in "Software Engineering and Knowledge Engineering hand book", vol.2, Emerging Technologies, World Scientific Publishing Co.
2. S. Murugesan, Y. Deshpande, S. Hansen and A. Ginige (2001), "Web engineering: Managing diversity and complexity of Web applications development," LNCS2016, pp3-13, Springer.
3. A. Ginige and S. Murugesan (2001), "Web engineering: An introduction", IEEE Multimedia, vol.8, no.1, pp.15-18.
4. J. Offutt (2002), "Quality Attributes of Web Software Applications", IEEE Software, vol.19, no.2, pp.25-32.
5. T. Isakowwitz, E. A. Stohr, and P. Balasubramanian (1995), "RMM: A methodology for structured Hypermedia design. ", Commun. ACM vol.38, no.8, pp.34-44.
6. E. Kirda, M. Jazayeri and C. Kerer (2001), "Experience in engineering flexible web services", IEEE Multimedia, vol.8, no.1, pp.82-87.
7. J. Nanard and M. Nanard (1995), " Hypertext design environments and the hypertext design process", Commun. ACM vol.38, no.8, pp.49-56.
8. C. Nam, G. Jang and J. Bae (2003), "An XML-based active document for intelligent web applications", Expert Systems with Applications, Vol. 25, pp. 165-176.
9. E. Koppen and G. Neumann (1998), "A practicle approach towards active hyperlinked documents", Computer Networks and ISDN Systems, Vol. 30, pp. 251-258.
10. C. Turner, A. Fuggetta, L. Lavazza, and A Wolf (1999), "A conceptual basis for feature engineering", the Journal of Systems and Software, vol.49, pp.3-15.
11. M. Sh. Aun, S. Yuen and K. Agusa (2002), "A framework for debugging client-side reflective and dynamic web applications", proc. 11th International World Wide Web Conf. WWW2002, Hawaii, USA, Available at: http://www2002.org/CDROM/alternate/690/index.html.
12. M. Sh. Aun, S. Yuen and K. Agusa (2004), "Identifying and addressing the extra issues involved in assuring quality attributes of client-side reflective and dynamic web applications", Proceedings

of the IRMA2004 International Conference, New Orleans, Louisiana, USA, pp. 912-916.
13. M. Sh. Aun, S. Yuen and K. Agusa (2003), “Towards Assuring the Quality Attributes of Web Applications: An Approach for Separating Features”, Proceedings of the IADIS International Conference WWW/Internet 2003, Algarve, Portugal, Vol. 2, pp. 1253-1254.
14. M. Sh. Aun, S. Yuen and K. Agusa (2004), “An approach for debugging client dynamic web applications”, IPSJ Journal, Vol.45 No. 10, pp.2373-2383.
15. J. K. Ousterbout (1998), “Scripting: Higher Level Programming for the 21st Century”,IEEE Computer, Vol. 31, No. 3, pp. 23-30.
16. Weblint - Html systax checker, http://filewatcher.org/sec/weblint.html
17. W3C, “W3C HTML validation services”, available at: http://validator.w3.org.
18. W3C, “HTML tidy”, available at: http://www.w3.org/People/Raggett/tidy/
19. C. Brabrand, A. Moller and M. I. Schwartzbach (2001), ”Static validation of dynamically generated HTML”, Proc. of (PASTE 2001), Snowbird, Utah USA, Available at: http://domino.research.ibm.com/confrnc/paste/paste01.nsf.
20. “Netscape JavaScript debugger 1.1”, available at: http://developer.netscape.com/docs/manuals/jsdebug/index.htm
21. Paolo Atzeni, Paolo Merialdo, and Giansalvatore Mecca (2001), “Data-Intensive Web Sites: Design and Maintenance”, World Wide Web, 4, 21-47.
22. C. Barry and M. Lang (2001), “A survey of multimedia and web development techniques and methodology usage”,IEEE Multimedia, vol.8, no.1, pp.82-87.
23. L. Ando, J. Santos, M. Caeiro, J. Rodrigues, M. Fernandez and M. Liamas (2001), “Moving the business logic tier to the client: Cost effective distributed computing for the WWW”, Software Practice and experience, vol.31, no.14, pp.1331-1350.
24. K. Agsteiner, D. Monjau, S. Schulze (1995), “Object-oriented High Level Modeling of Complex Systems”, In: Proceedings European Workshop on Computer Aided Systems Technology (EUROCAST), Innsbruck, Austria.
25. V. Quint, C. R. and I. Vatton (1995), “A structured authoring environment for the World-Wide Web”, Proc. of the 3rd International World-Wide Web Conference WWW95, Darmstadt, Germany.
26. J. Huang (1978), “Program instrumentation and software testing”, Computer, vol.11, no.4.
27. A. E. Hassan and R. Holt (2002), “Architecture Recovery of Web Applications”, Proc. of ICSE 2002: International Conference on Software Engineering, Orlando, Florida, pp.19-25.
28. R. Tesoriero and M. Zelkowitz (1998), “A web-based tool for data analysis and presentation”, IEEE Internet Computing, pp.63-69.
29. J. Boustedt (2002), “Automated analysis of dynamic web services”, Master thesis in computer science, Uppsala University, Sweden.
30. Netscape: Netscape JavaScript Debugger 1.1, Available at: http://developer.netscape.com/docs/manuals/ jsdebug/index.htm.
31. Microsoft: Microsoft Script Debugger, Available at: http://msdn.micrsoft.com/ library/en-us/sdbug/Html/sdbug_1.asp.