Vol. 45 No. 10 IPSJ Journal Oct. 2004

Regular Paper

An Approach for Debugging Client Dynamic Web Applications

Mohamed Sharaf Aun,[†] Shoji Yuen[†,††] and Kiyoshi Agusa[†]

Beside the emergence of the Internet and the Web, an equal (or probably more important) trend has been the propagation of scripting technology. Client scripting involves the use of light, yet powerful languages in making the Web more interactive and dynamic. In view of software description, however, the light-weight feature turned out to be a major drawback to develop and maintain the scripts with a good quality. Once an unexpected behavior emerges, sometimes it is difficult to find the cause even for a short script. This paper, in addition to discussing the challenges facing the debugging process, presents an approach proposed for debugging client Web applications focusing on debugging scripts as they are the dynamic enabling components and the means by which execution of the business rules can be captured. In addition to identifying the challenges, our aim is at facilitating the developing and the debugging process. For this purpose, we focus on imposing syntactic constrains over a script and on visualizing its state while under debugging. For the purpose of proving the effectiveness of the proposed new approach, an experimental debugging tool is prototyped. As the result, it is shown that our approach is helpful in finding implicit bugs and in inspecting the state of the application under debugging.

1. Introduction

Beside the emergence of the Internet and the Web, an equal (or probably more important) trend has been the propagation of scripting[1] technology. Client scripting involves the use of light, yet powerful languages in making the Web more interactive and dynamic. The script is required to be executed in a compact and light-weight manner. In view of software description, however, the light-weight feature turned out to be a major drawback to develop and maintain an application with good quality. Once an unexpected behavior emerges, sometimes it is difficult to find the cause even for a short script. Since a script language itself is usually as expressive as a conventional programming language, the debugging process should be powerful enough as that for the general programming language.

This work represents an on-going effort to develop a computer-aided environment that can support the construction of high quality Web applications. In this paper, in addition to discussing the challenges facing the debugging process of client part applications, we present an approach proposed for debugging such applications focusing on debugging scripts.

In addition to identifying the challenges, we aim at facilitating the debugging process by extending conventional techniques that support mix of static and dynamic capabilities. Because of the nature of the client-side scripts, the major differences are as follows: (1) since the script is executed on demand, the program validation technique such as type-checking does not fit. (2) since the server can control the script loosely, the current state is hard to be determined in the present Run-Time Environments (RTEs). The present RTEs are not capable of reporting the termination of a faulty script, even do not report the closer information.

To check more correctness without sacrificing the light-weight nature of executing the scripts, we first separate the script analysis from the RTE. By imposing more strict syntactic checking, we identify the defects before execution. Since such defects are not reported by the present RTE, we categorize such defects as *"silent bugs"*. Next, by inserting inspection library codes, we identify the defects in behavior. We categorize such defects as *"active bugs"* since we can observe the unexpected behavior. We provide a systematic mechanism to distinguish the dynamic behavior through the inspection library.

Based on this idea, we present our approach

† Department of Information Engineering, Graduate School of Information Science, Nagoya University
†† JST PRESTO

A draft of this paper has appeared in the proceedings of the 11th World Wide Web (WWW2002) International Conference[2]

with an experimental prototype tool. The tool consists of the following components: overall checker and scripts extractor, silent bugs identifier, debugging library functions, source code transformer, and interpreting facilitator.

Section 2 discusses the challenges facing the debugging process. Section 3 analyzes bugs in client applications identifying their main categories. The new proposed approach is described in Section 4 and an overview of an experimental prototype tool is presented in Section 5. Section 6 discusses the characteristics of our approach and Section 7 compares it with other related works. Finally the paper is concluded with a conclusion and future directions.

2. Challenges in Debugging

Debugging client Web applications is a challenge process due to many factors. Some of these factors are languages-dependent and some others are languages-independent.

2.1 Language-dependent Challenges

Web languages, specially scripting, have many new and specific features, imposing many challenges and difficulties on the debugging process. These features include the following:

- *Embedded components*: Script are usually embedded within mark-up languages which themselves embedded in web pages that are processed by browsers. The browsers work on operating systems which are running on a certain hardware. Thus, there is a variety of possible error sources.
- *Neither Pure OOL nor Structural Languages*: Scripting languages are neither pure structural nor pure object-oriented languages. They are not pure structural because they have to deal with objects and they are not OO languages since the objects to deal with are mainly environmental or of special nature.
- *Immediate Interpretation*: Consequently, traditional debugging techniques can not be used since they are mainly based on either the compilers themselves or on integrating such techniques to the compilers 3). This also means that bugs are only noticed during run-time.
- *Type-less*: Like other scripting languages 1), Web scripting languages are type-less. This implies that all variables look and behave the same so that they are interchangeable. 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.

2.2 Languages-independent Challenges

Among the languages-independent challenges are those due to the nature of the applications themselves, and those due to the nature of web RTEs.

2.2.1 The Nature of Applications

Web applications differ from traditional software applications in several critical dimensions due to the following features:

- *Coexistence of Multiple Technologies*: Client Web applications involve a variety of embedded components such as scripting and rendering components. These components are realized with different categories of languages. An important issue often faced is how bugs are going to be identified and eliminated.
- *Interactive and Dynamic*: One of the main reasons behind the popularity of web is its capability in being more interactive and dynamic. As for client-side, applications may contain scripts that define additional dynamic behavior and often interact with the browser, page content and additional controls such as Applets. The dynamic behavior imposes many difficulties on debugging.
- *Reflective*: A program can change itself during execution or generate a new program or script on-the-fly. This feature implies that the state of a program at any time is a function not only of the global variables but a function of the combination of both the variables and the program itself.
- *Event-driven Applications*: Web applications, unlike structural programs, are typically event-driven programs. This makes the portion of the program to be executed next specified by an event.
- *Manipulate Environment Objects*: Most of the objects manipulated by client scripts are *environment objects*. Therefor, any bug due to manipulating them can not be treated far from the environment which creates such objects.

2.2.2 The Nature of Web RTEs

Unlike the RTEs of most programming languages, ones of web applications impose challenges for the debugging process because of the following:

- *High Volatility*: Web RTEs are much more volatile as web changes tremendously over the course of a few milliseconds. Variables, threads, stack frames, and heap objects rapidly come and go. More concurrent activities are involved on the web.

- *Multipurpose Environments*: As for the client-side, the RTE have no longer to be simple HTML rendering tools, instead they have to support Applets, embedded scripts, stream media, 3D graphics, click-able maps and virtual machines.
- *Lack of Debugging Capabilities*: Such feature is a consequence of the previous one. An example where a browser neither tells what is wrong nor executes the function is when using a keyword such as the word *case* as a function name.
- *Difficult to Control*: Facilitating the debugging process requires controlling the RTE in order for the developer to be able to stop execution at a certain point and be able to investigate the state of the program under execution. However, controlling the web RTEs like browsers is a difficult process due to many reasons including their specific features.

3. Analyzing Bugs

To be able to determine the main categories of bugs, we have first to look at how scripts are executed.

3.1 Execution of Client Applications

When code to be executed on a client, there are two alternatives for achieving that. The first is by executing abstract code (or byte-code) in virtual machines. An example of such case is the execution of applets. Byte-code generation is proceeded by lexical, syntax and semantic analysis. Such analysis provides the necessary mechanisms for checking both lexical and syntactics faults.

However, scripting components are immediate interpreted. They are executed by browsers, where they are interpreted line by line as they arrive, see the bold line in **Fig. 1**. In this case no abstract code is generated, causing faults to be identified only during run-time.

3.2 Main Categories of Bugs

While executing a script, excluding non-scripting bugs and the cases where the RTEs pop up error messages due to browser version or type incompatibility, there are four possibilities. These possibilities are: (1) the RTE keeps silent (neither it runs the script nor it produces any helpful error message), (2) the RTE runs the scripts until some error found and it stops there, (3) the RTE runs the script but unexpected or undesirable results are produced, and (4) the RTE runs the script correctly.

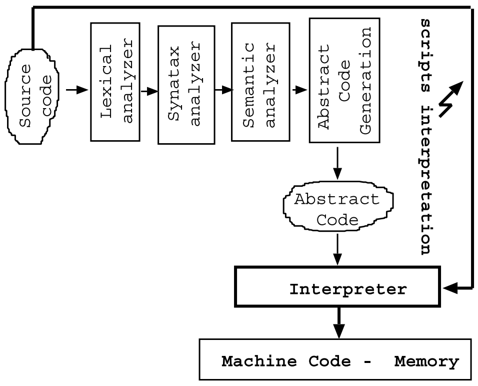


Fig. 1 Immediate interpretation of scripts.

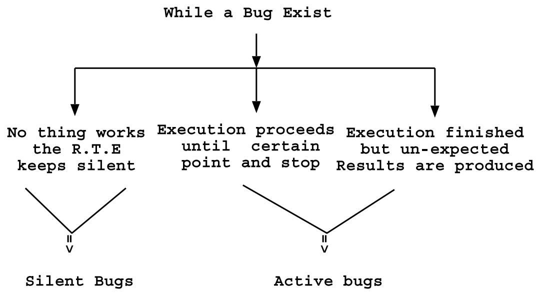

While a Bug Exist
No thing works the R.T.E keeps silent
Execution proceeds until certain point and stop
Execution finished but un-expected Results are produced
Silent Bugs
Active bugs

Fig. 2 Possible cases while a bug exists and the broad categories of bugs.

By analyzing the first three possibilities (**Fig. 2**), we found that there are two main categories of bugs:

- The first category causes the RTE neither to run the application nor to produce any helpful error message. In another word, the RTE keeps silent. We refer to such bugs as *silent bugs*. An example of this category is when using a keyword such as the word *case* as a function name, (see Fig. 7).
- The second main category occurs when either the execution stops before completion or when execution is completed but unexpected or undesirable results are produced. We refer to this kind of bugs as *active bugs*.

An example of the second category is presented in **Fig. 3**. In this listing: execution of the script proceeds until line 4 and stops there. This happens because the variable "S" is not defined before being used in the expression. Another example of the second category, where ex-

```
function calcDist(x1,x2,y1,y2){
  Sx = Math.pow((form.x1.value - form.x2.value),2);
  Sy = Math.pow((form.y1.value - form.y2.value),2);
  distance = Math.sqrt(S +Sy);
  }
 var p1x = 2; p1y = 3; p2x =10; p2y= 7;
 calcDist(p1x, p1y, p2x, p2y);
```

Fig. 3 An example of active bugs.

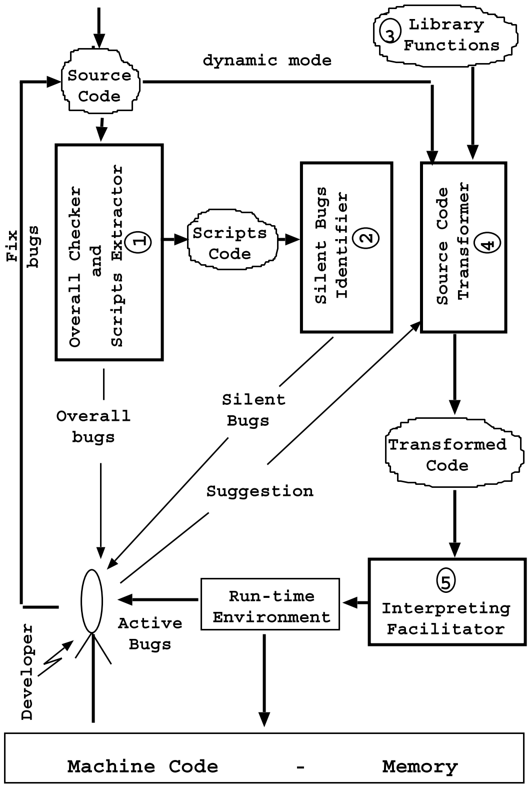


Fig. 4 Reference model of our debugging approach.

ecution is finished with unexpected results, is presented in Fig. 10.

4. The Debugging Approach

To address the challenges and to fulfill the specific debugging requirements, our approach consists of five complementary and interacting components (**Fig. 4**) and will operate with the guidance from the developer.

4.1 Components and their Functionalities

Each of the five components has its own purpose and function as described in the next subsection. More descriptions related to implementation are presented in Section 5.

4.1.1 Overall Checker and Scripts Extractor

This component is responsible for dealing with the complexity introduced by the embedded nature of the scripts and the situation where many technologies have to coexist within one application. The source code of the whole application is read, overall checked, and the scripts components are extracted for further processing. The input to this component is the whole application and the output is the script components in addition to overall bugs' identifications. An example of identifying overall bugs is presented in Fig. 7.

Compared to available checking tools such as weblint [4)], this subsystem have also to extract the scripts found for further processing. Moreover, the checker can serve several other purposes such as checking cross-browsers scripting, code re-writing, code beautification and code documentation.

4.1.2 Silent Bugs Identifier

This component concentrates on identifying the first category of bugs (i.e., silent bugs) that causes the RTE neither to run the application nor to produce any helpful error messages. It 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.

The input of this component is the scripts extracted by the first component. The output is the silent bugs identification.

4.1.3 Library Functions

This component, together with the source code transformer (described next), is necessary for identifying active bugs. It is also necessary for efficiency and re-usability purposes.

This component is devoted for developing debugging techniques and methodologies realized as functions that can be augmented within the source code during the source expansion process and then called as necessary. These functions must include specialized code for inspecting both objects and scalar variables.

4.1.4 Source Code Transformer

The purpose of this component is to perform the source code instrumentation. Instrumentation is the process of inserting additional statements into a program for the purpose of gathering information about its dynamic behavior.

Unlike traditional instrumentation, instrumentation, here, is accomplished by expanding the code under debugging automatically not

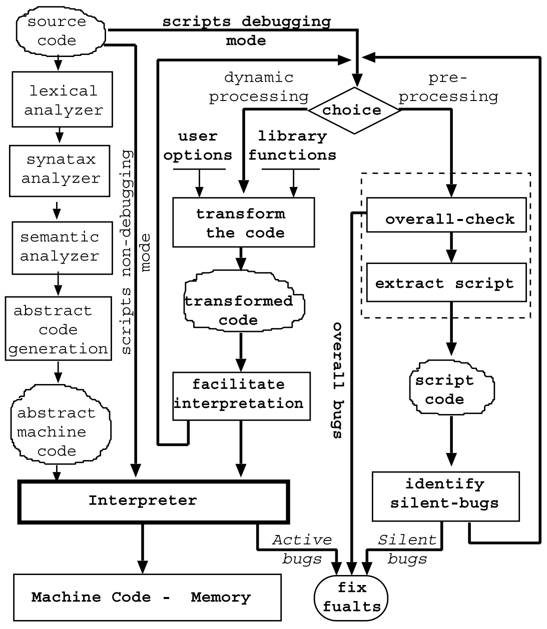


Fig. 5 Flow of the approach and its integration with the RTE.

only with additional monitoring statements but also with the necessary debugging functionalities required for monitoring the state of the program or controlling its execution. While monitoring statements can be injected at any point of the original source code, the debugging library functions have to be inserted prior to the original source code.

The input to this part is the source-code under debugging, the required debugging library functions and the developer options (or suggestions). The source code is to be transformed and the debugging functions are to be augmented. Developer options are necessary for providing more flexible debugging process. The source code is instrumented and it is called the transformed code.

4.1.5 Interpreting Facilitator

This part is necessary for facilitating the execution of the applications directly from the context of the debugger. It works as an interface between the transformed code, produced by the previous component, and the interpreter.

4.2 Integrating the Approach with the RTE

Figure 5 shows how the proposed debugging model can be augmented as an intermediate component between the source code and the interpreter. In the debugging mode, both preprocessing and dynamic debugging are facilitated depending on the choice made by the developer. If the developer chooses preprocessing, the tool will examine the source code for silent and overall bugs. If the choice is dynamic processing, the tool will look for active bugs.

Comparing this figure with Fig. 1, we can find that rather than only immediate interpretation, the code can be checked for bugs statically and dynamically. While dynamic processing, code interpretation is performed under the control of the debugger.

5. An Experimental Prototype Tool

To prove the effectiveness of our approach, an experimental prototype tool has been implemented. The current implementation is able to deal with and debug a wide range of client-side applications containing JavaScript [5] programs. JavaScript was chosen because of its popularity on the Web. The flow of the tool can be expressed as in the right-side of Fig. 5 presented above. While dynamic processing involves an interpreter, preprocessing dose not.

The current prototype system is not standalone from the viewpoint of the Web client (browser). It was implemented as a Web system runnable by a browser used for interpreting the scripts under development. Such choice avoids building an interpreter from scratch. It also makes the debugger to be integrative with the RTE. This is necessary as the debugger has to be able to control the execution of a program under debugging. Finally, such choice has made the system easy to use and the user interface to be more friendly.

5.1 Tool Subsystems

Based on the approach and to work within its context, the tool was implemented with five subsystems as presented above. More description related to implementation is presented in this section.

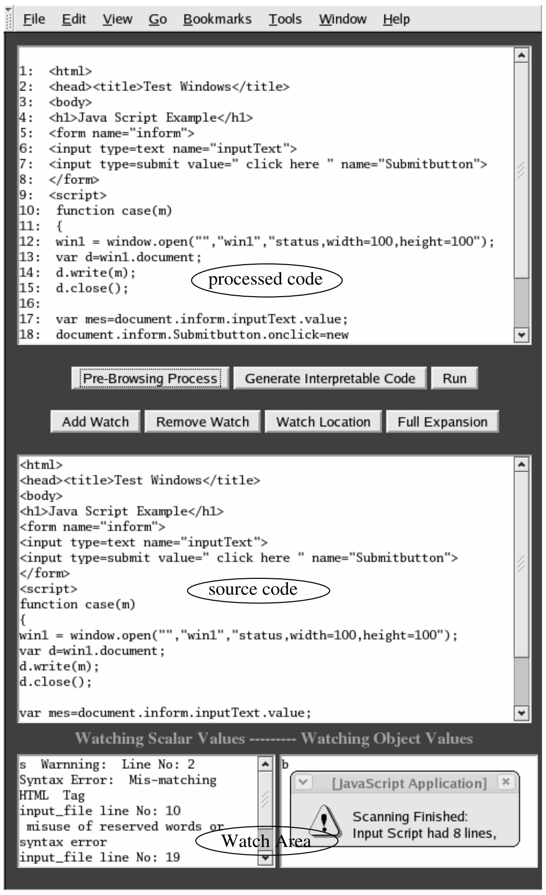
5.1.1 Overall Checker and Scripts Extractor

This subsystem involves the use of scanner that can make primary check for the application as a whole and be able to extract the scripts components for further processing.

Figure 6 illustrates how the main function of this subsystem was implemented. First the components, which may exist before any embedded script are analyzed by a special subroutine called (*look_for_script*). This subroutine also check for over-all bugs at the line level and saves some information to be used

```
function js_scan_main()
{
  while  (SCAN_source_code.indexOf(NL)>-1){
   look_for_script();//process  untile a script is found
   if (script_Found){
      extract-script-i;
      parseAndLexScript(script_i);
   }
  }
  check_for_HTML_Error();//Chek for possible overall bugs
  SCAN_reset();
}
```

Fig. 6 Overall checker and script extractor main function.



```
1:  <html>
2:  <head><title>Test Windows</title>
3:  <body>
4:  <h1>Java Script Example</h1>
5:  <form name="inform">
6:  <input type=text name="inputText">
7:  <input type=submit value=" click here " name="Submitbutton">
8:  </form>
9:  <script>
10:  function case(m)
11:  {
12:  win1 = window.open("","win1","status,width=100,height=100");
13:  var d=win1.document;
14:  d.write(m);
15:  d.close();
16:
17:  var mes=document.inform.inputText.value;
18:  document.inform.Submitbutton.onclick=new
```

```
<html>
<head><title>Test Windows</title>
<body>
<h1>Java Script Example</h1>
<form name="inform">
<input type=text name="inputText">
<input type=submit value=" click here " name="Submitbutton">
</form>
<script>
function case(m)
{
win1 = window.open("","win1","status,width=100,height=100");
var d=win1.document;
d.write(m);
d.close();

var mes=document.inform.inputText.value;
```

```
s  Warnning:  Line No: 2
Syntax Error:  Mis-matching
HTML  Tag
input_file line No: 10
 misuse of reserved words or
syntax error
input_file line No: 19
```

```
b
Scanning Finished:
Input Script had 8 lines,
```

Fig. 7 Example of identifying overall and silent bugs.

for identifying overall bugs. When any script is found, it is extracted and sent for further processing. This process is repeated until the end of the file is found. Further overall bugs are then checked, based on the information collected while looking for the scripts. This is accomplished by special sub-program called *check_for_HTML_error()*.

An example for identifying overall bugs is illustrated in in the lower left window of **Fig. 7**. The error (missing of the closing of the head tag) is indicated together with its line number. The processed source code is reproduced in the upper window with line numbers in order to simplify the task of finding the location where a bug has occurred.

5.1.2 Silent Bugs Identifier

By studying silent bugs more closely, we found most of them are mainly due to syntax errors, like the use a reserved ward as function name or the use of undefined variable. Thus, preprocessing is necessary for identifying such category of bugs.

Preprocessing requires the need for implementing a lexer and a parser for the language used for developing scripts. The lexer generates tokens and the parser checks them according to the language grammar. Therefore the language grammar is an important factor for achieving this part.

Virtually every programming language (including JavaScript) constructs can be represented by a context-free grammar. JavaScript has LL(1) grammar as described in[6]. Based on that, this part was adapted and built.

Figure 7 illustrates how this subsystem works. In this figure, the input file shown in the middle part has two silent bugs each can cause the RTE to freeze while trying to execute script directly. The first bug is due to the use of the reserved word *case* as a function name. The second bug is due to the unclosed block found in the function in line 11. There is a block start indicator (i.e., {) but such opening block is not closed any where within the script. Both bugs are identified together with their line numbers as shown in the bottom window of the figure.

An important issue to mention here is that although this part may borrow some techniques of building lexers and parsers from traditional ones, the tokens to generate and to deal with are of different natures. An example for that is the necessity of regarding the dot operator as a subcomponent of a token. As such, the string: *"document.myForm.myInput.myValue"* represents only one token.

5.1.3 Library Functions

In case of JavaScript language[5], there are two features that make the use of debugging libraries possible. The first is the capability of the scripting document to include external files. Such files are written with *.js* extension. The other key feature is the statement (*for* ... *in* ... *object*) provided by the language[7),8]. Such statement can iterate over all the properties of an object.

A variety of debugging library functions are

```
function dbFunc(label, item, mode){
 if (typeof(item) != "object")
  creator.document.isn.watchout.value=(label+"="+item);
 else{ var msg = "";
  for (i in item) {
   msg +=label + "." + i + "=" + item[i];
   creator.document.isn.watchoutobj.value=msg;msg ="";
  }
}
```

Fig. 8 An example of library debugging functions.

```
option <= get-developer-options;
if (option = full) then
   extract-expand-full(watch-items);
else
   extract-expand-short(watch-items)
endif
```

Fig. 9 The algorithm for extending the code.

possible for different options. These options may include the debugging mode (e.g., tracing, single stepping), the required DOM hierarchy, and so on. **Figure 8** shows an example of such debugging library functions.

5.1.4 Source Code Transformer

In the implementation which we have made (see **Fig. 9**), source code transformation can take place in tow manners depending on the developer option. In the first case, the resulted code contain the original code proceeded by the necessary debugging function injected as scripts. This case is referred as full transformation and it is helpful when object hierarchy [9] is important. In the second option, the transformed code contain the original source code proceeded only by the declaration of the location where the library debugging functions exist. The later case is referred as short transformation.

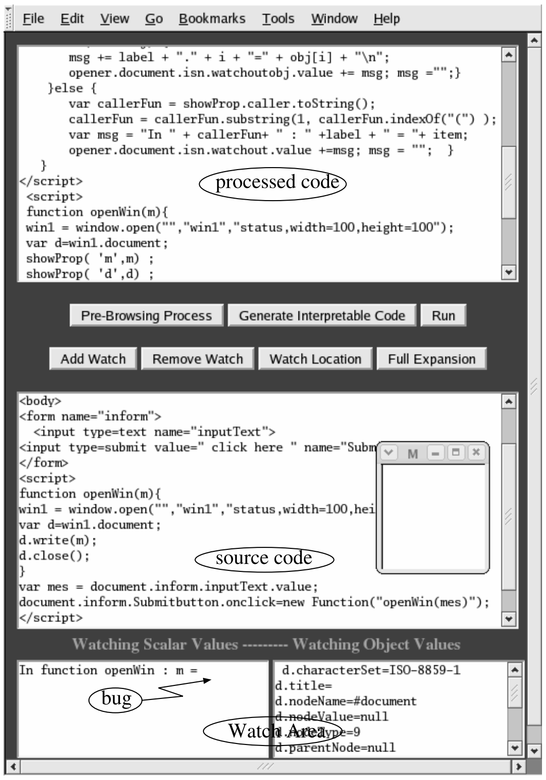
Figure 11 shows an example of how the transformed code (in the upper-widow) is produced by expanding the source code (presented in source code area) with the whole debugging function plus the calling statements required. The debugging library function was inserted as a whole at the beginning and then called as necessary. Such transformed code is now ready to be processed by the interpreting facilitator described next.

5.1.5 Interpreting Facilitator

As an interface between the expanded code and the RTE, this subsystem initiates the execution of such code from the context of the debugger. It communicates with the RTE by sending it the code to be interpreted. Since the transformed code contains the necessary debugging functionalities, program execution control is accomplished from the context of the application itself rather than by external components.

```
<html>
<head><title>Test Example</title></head>
<body>
<form name="inform">
<input type=text name="inputText">
<input type=submit value=" click here " name="Submitbutton">
</form>
<script>
 function openWin(m){
  win1 = window.open("","win1","status,width=100,height=100");
  var d=win1.document;
  d.write(m);
  d.close();
 }
 var mes=document.inform.inputText.value;
 document.inform.Submitbutton.onclick=new Function("openWin(mes)");
</script>
</body>
</html>
```

Fig. 10 A debugging example.


File Edit View Go Bookmarks Tools Window Help
processed code
Pre-Browsing Process
Generate Interpretable Code
Run
Add Watch
Remove Watch
Watch Location
Full Expansion
source code
M
Watching Scalar Values --------- Watching Object Values
bug
Watch Area

```
        msg += label + "." + i + "=" + obj[i] + "\n";
        opener.document.isn.watchoutobj.value += msg; msg ="";}
    }else {
        var callerFun = showProp.caller.toString();
        callerFun = callerFun.substring(1, callerFun.indexOf("(") );
        var msg = "In " + callerFun+ " : " +label + " = "+ item;
        opener.document.isn.watchout.value +=msg; msg = "";  }
   }
</script>
 <script>
 function openWin(m){
 win1 = window.open("","win1","status,width=100,height=100");
 var d=win1.document;
 showProp( 'm',m) ;
 showProp( 'd',d) ;
```

```
<body>
<form name="inform">
  <input type=text name="inputText">
<input type=submit value=" click here " name="Subm
</form>
<script>
function openWin(m){
win1 = window.open("","win1","status,width=100,hei
var d=win1.document;
d.write(m);
d.close();
}
var mes = document.inform.inputText.value;
document.inform.Submitbutton.onclick=new Function("openWin(mes)");
</script>
```

```
In function openWin : m =
```

```
 d.characterSet=ISO-8859-1
d.title=
d.nodeName=#document
d.nodeValue=null
d.nodeType=9
d.parentNode=null
```

Fig. 11 Results sample: Debugging active bugs with full expansion and the debugging data written to watch out windows.

5.2 Example of Dynamic Debugging

The example shown in **Fig. 10** has one script written in JavaScript and is embedded in HTML document. The script has one function that opens a new window dynamically and 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 example (**Fig. 11**), we can notice the following: the source code is in the middle part of the figure, the extended code is in the upper-side, and the result of the debugging process is shown at the lower part of the same figure. The value of the variable m (shown at the bottom-left window) is null indicating the bug. The variable m is null because the assignment in line 14 takes place before the user can enter some other value in the text area defined in line 5.

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 and the lack of compilation type-checking.

6. Characteristics of the Approach

The new proposed approach consists of five components each of which has its own purpose and function. By looking closely at these components, we will find that they, collectively, can overcome the challenges and fulfill the requirements for the debugging process.

The first component (or the overall checker) is responsible for dealing with the difficulties introduced by the fact that on the Web multiple technologies have to coexist. The second component identifies the first category of bugs. These two components, together, overcome the challenges due to the immediate interpretations, the type-less property of the languages used and the lack of the debugging capabilities of the RTE.

The third, the fourth and the fifth components together deal with active bugs. They overcome the difficulties introduced by the reflectiveness and dynamic properties together with the nature of the manipulated objects. They also fulfill the need of the debugger to cooperate and coordinate its activity with the RTE providing the necessary mechanisms for controlling program execution. Moreover, the fifth component facilitates the execution of the applications from the context of the debugger.

Therefor, our approach combines both static and dynamic debugging capabilities. Preprocessing is necessary for eliminating silent bugs. Dynamic debugging identifies active bugs and enables controlling program execution from the context of the application itself rather than by external components. Consequently, it facilitates the debugging process without the need for re-building RTE from scratch or extending the existing ones. It also avoids the problems, which may arise due to controlling the RTE through external components.

7. Related Work

Web engineering[10)~12)] has emerged as a new discipline for the establishment and use of sound scientific and engineering approaches to the successful development, deployment and maintenance of high quality Web-based systems and applications. It is still a very young discipline and has just started gaining attention of researchers, developers, and academics.

To support the design phase of Web applications, several frameworks, architectures and models, such as those described in[13)~18)], have already been designed and proposed. However, new application developing models have evolved, new languages have gain popularity; and yet testing and debugging methodologies have not changed to cope with such new trends.

To our knowledge, there is no previous work in the academic literature on debugging techniques targeting the dynamic behavior of the web with the exception of the work by the ⟨*Bigwig*⟩ project team (Claus Brabrand, et al.)[19)], which addresses the static validation of dynamic generated HTML in the context of the ⟨*Bigwig*⟩ language. While the aim of their work is at validating HTML components in the context of specific language, our work aims at debugging scripts components as they are the main reason behind making the web more interactive and dynamic.

The software engineering literature includes some works on debugging, but it is generally aimed for debugging traditional software systems. Web applications differ from traditional software systems in several dimensions. Moreover, looking at the evolution of traditional debugging techniques[3)], we find that they are built based on the compilers. Web applications, however, are mainly interpreted rather than compiled.

In the Web engineering literature, to our best knowledge, there is no paper that investigates the testing and debugging process except from mentioning it as an open research topic[20),21)].

Recently, some browser vendors claim the provision of tools for debugging scripts. An example for that, is the JavaScript debugger[22)].

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[23]. 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. As stated in the previous Section, it combines both static and dynamic capabilities. In spite of the dynamic nature of client applications, we found that substantial part of the conventional static debugging process is applicable. Combining both static and dynamic capabilities has pointed out a new direction for building new debugging tools as it has several advantages.

Furthermore, 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.

As for comparing our prototype implementation with the above tools, our prototype tool can interact with the interpreter through the interpreting facilitator. It does not interfere with the internal structure or components of the browser. To be more concrete, **Fig. 12** provide the necessary notation for representing the activities performed by the tool while debugging. As we can see, unlike other works, the tool dose not expose the web browser's internal event object. This makes our approach portable rather than to be as an added feature to some propriety browser.

8. Conclusions and Future Work

In response to the lack of existing approaches specifically designed for debugging client Web applications, we have proposed an approach for that purpose and described an experimental prototype tool for proving its effectiveness. We focused on debugging scripts, as they are the main reason behind making the Web more interactive and dynamic.

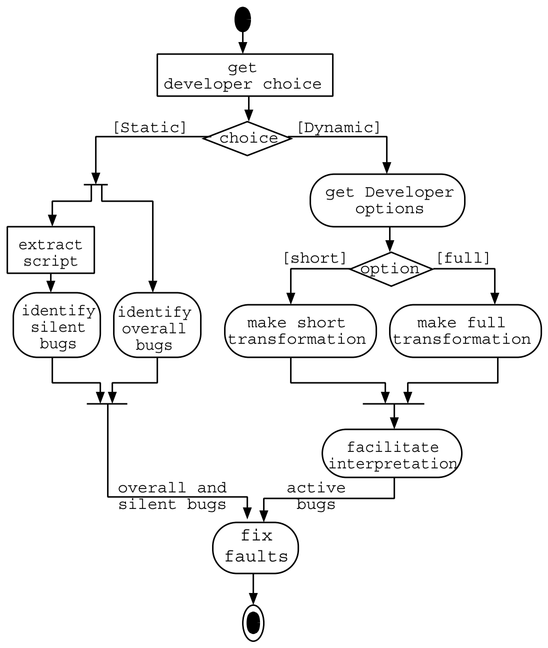


Fig. 12 UML activity diagram representing the prototype implementation.

We have also identified the issues that make debugging client applications different from debugging traditional software systems, explaining why new approaches are required.

It was found that, the major differences are as follows: (1) since the script is executed on demand, the program validation technique such as type-checking does not fit. (2) since the server can control the script loosely, the current state is hard to be determined in the present runtime environment. Thus, making a pure transposition of debugging techniques (like other techniques[24]) from software engineering to the Web is both difficult and inadequate.

To check more correctness without sacrificing the light-weight nature of executing the scripts, we first imposed more strict syntactic checking identifying one category of defects before execution. Since such defects are not reported by the present runtime environment, we categorized such defects as "silent bugs". Next, by inserting inspection library codes, we were able to identify the defects in behavior. We categorized such defects as "active bugs" since we can observe the unexpected behavior.

Our approach is effective in the sense that facilitating the debugging process on the Web is not trivial due to the nature of the applications themselves, the nature of the RTEs and the nature of the languages used. Debugging requires either to rebuild web RTE from scratch incorporating debugging functionalities or adapting the existing ones, or at least control them remotely by external components. Each of this approaches has its own problems in addition to their in-feasibility. Our approach, however, can facilitate the debugging process avoiding all such problems. The approach requires no modification for the RTE making its inherit advantages to be platform and RTE independent. It also combines the advantages of both preprocessing and dynamic debugging.

Several interesting research issues have been highlighted by this work. We plan to extend it in several different ways. Currently the approach can handle only client applications focusing on scripts components. We plan to enhance it for other components and for handling components generated dynamically at the server-side and presented on-the-fly to the client. In this direction, we have proposed an approach through separating features[25].

Acknowledgments The authors would like to express their thanks to S. Yamamoto (associate professor in Aichi Prefectural University), T. Hamaguchi (research associate in Agusa Lab.) and all members of our Lab. for their useful comments. We also would like to thank Dr. A. Mustafa for providing valuable feedback on various drafts of this paper.

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.

References

1) Ousterbout, J.K.: Scripting: Higher-Level Programming for the 21st Century, *IEEE Computer*, Vol.31, No.3, pp.23–30 (1998).
2) Aun, M. Sh., Yuen, S. and Agusa, K.: A Framework for Debugging Client-side Reflective and Dynamic Web Applications, *Proc. 11th World Wide Web (WWW2002) International Conference*, Hawaii, USA (2002). available at: http://www2002.org/CDROM/alternate/690/index.html
3) Kolawa, A.: The Evolution of Software Debugging, ParaSoft, available at: http://www.parasoft.com/papers/vision.htm.
4) Weblint — HTML Syntax Checker, available at: http://filewatcher.org/sec/weblint.html.
5) Netscape: *JavaScript 1.1 Language Specification*. http://wp.netscape.com/eng/javascript/index.html.
6) Rossi, M.: NGS JavaScript Interpreter, (1998), available at: http://people.ssh.fi/mtr/js/manual/js_toc.html.
7) Goodman, D.: *Dynamic HTML: the Definitive Reference*, O'Reilly (2002).
8) Ayke, A.W., Gilliam, J.D. and Ting, C.: *Pure JavaScript*, Samas Publishing (1999).
9) Clinger, P.: *JavaScript Beans.*, available at: http://www.netpedia.com/features/javascript/beans/
10) Murugesan, S., Deshpande, Y., Hansen, S. and Ginige, A.: Web engineering: A New Discipline for Web-based System Development, *Proc. 1st Int'l Conf. Software Eng., Workshop on Eng.*, On-line http://fistserv.macarthur.uws.edu.au/san/icse99-WebE/ICSE99-WebE-Proc/San.doc
11) Ginige, A. and Murugesan, S.: The Essence of Web Engineering, *IEEE Multimedia*, Vol.8, No.1, pp.22–25 (2001).
12) Deshpande, Y. and Hansen, S.: Web Engineering: Creating a Discipline among Disciplines, *IEEE Multimedia*, Vol.8, No.1, pp.82–87 (2001).
13) Isakowwitz, T., Stohr, E.A. and Balasubramanian, P.: RMM: A Methodology for Structured Hypermedia Design, *Comm. ACM*, Vol.38, No.8, pp.34–44 (1995).
14) Garzotto, F., Paolini, P. and Schwabe, D.: HDM: A Model Based Approach to Hypermedia Application Design, *ACM Trans.Info.Syst.*, Vol.11, No.1, pp.1–26 (1993).
15) W3C: *Document Object Management*, http://www.w3.org/DOM (1999).
16) Nam, C., Jang, G. and Bae, J.: An XML-based Active Document for Intelligent Web Applications, *Expert Systems with Applications* (2003).
17) Conallen, J.: *Building Web Applications with UML*, Addison-Wesley, Canada (2000).
18) Ando, L., Santos, J., Caeiro, M., Rodrigues, J., Fernandez, M. and Liamas, M.: 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 (2001).
19) Brabrand, C., Moller, A. and Schwartzbach, M.I.: Static Validation of Dynamically Generated HTML, *Proc. PASTE 2001*, Utah, USA (2001). http://domino.research.ibm.com/confrnc/paste/paste01.nsf.
20) Ginige, A. and Murugesan, S.: Web Engineering: An Introduction, *IEEE Multimedia*, Vol.8, No.1, pp.15–18 (2001).
21) Ginige, A. and Murugesan, S.: The Essence of Web Engineering, *IEEE Multimedia*, Vol.8, No.1, pp.22–25 (2001).

22) Netscape: Netscape JavaScript Debugger 1.1, http://developer.netscape.com/docs/manuals/jsdebug/index.htm.
23) Microsoft: Microsoft Script Debugger, http://msdn.micrsoft.com/library/en-us/sdbug/Html/sdbug_1.asp.
24) Nanard, J. and Nanard, M.: Hypertext Design Environments and the Hypertext Design Process, *Comm. ACM*, Vol.38, No.8, pp.49–56 (1995).
25) Aun, M. Sh., Yuen, S. and Agusa, K.: Towards Assuring Quality Attributes of Web Applications: An Approach for Separating Features, *Proc. IADIS WWW/Internet2003 International Conference*, Algarve, Portugal, Vol.2, pp.1253–1254 (2003).

(Received January 6, 2004)
(Accepted September 3, 2004)

Mohamed Sharaf Aun received M.E. degree in information engineering from Nagoya University in 2001. Currently, he is a Ph.D. candidate of the Graduate School of Information Engineering at Nagoya University. His research interests include Web engineering, quality assurance methodologies and document-computation.

Shoji Yuen received a DrEng degree from Nagoya University (1997). He is currently an associate professor of the Graduate School of Information Science at Nagoya University. He is interested in the formal model of concurrency, and the formal verification of concurrent and network software based on the formal concurrent models.

Kiyoshi Agusa is a professor of Department of Information Systems, Graduate School of Information Science, Nagoya University. He received Ph.D. degree in computer science from Kyoto University in 1982. His research area covers software engineering, program repository, software reuse. He joins e-Society project which is a part of e-Japan project, and researches on a high reliable Web-based Applications. He is a member of IPSJ, ISSST, IEICE, ACM and IEEE.