Delft University of Technology
Software Engineering Research Group
Technical Report Series

Automatic Invariant Detection in Dynamic Web Applications

Frank Groeneveld, Ali Mesbah, and Arie van Deursen

Report TUD-SERG-2010-037

TUD-SERG-2010-037

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: This paper is currently under review.

Automatic Invariant Detection in Dynamic Web Applications

Frank P. Groeneveld
Delft University of Technology
The Netherlands
F.P.Groeneveld@student.tudelft.nl

Ali Mesbah
Delft University of Technology
The Netherlands
A.Mesbah@tudelft.nl

Arie van Deursen
Delft University of Technology
The Netherlands
Arie.vanDeursen@tudelft.nl

ABSTRACT

The complexity of modern web applications increases as client-side JavaScript and dynamic DOM programming are used to offer a more interactive web experience. In this paper, we focus on improving the dependability of such applications by automatically inferring invariants from the client-side and using those invariants for testing. By combining JavaScript code instrumentation and tracing we infer runtime program invariants. Furthermore, we dynamically analyze DOM-trees and use learning algorithms to detect template-based invariants per user interface state, across various states, as well as across multiple execution runs. Our open source implementation of the technique is agnostic to server-side technology and capable of automatically using the detected invariants for testing web applications. We demonstrate through a series of case studies that (1) code-level and structural invariants exist in web applications with a large client-side state, (2) they can be automatically detected, (3) they can serve as regression test oracles.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification, Design

Keywords

Web applications, invariant detection, test automation

1. INTRODUCTION

There is a growing trend to move software applications towards the Web. Facilitated by advances made in recent *Web 2.0* technologies, people are in increasing numbers using the web as a programming platform to create a wide range of web-based systems. Thus, web-browsers not just offer the possibility to navigate through a sequence of HTML pages, but enable rich user interaction via graphical user interface components.

While this positively affects user-friendliness and interactiveness of web applications, it comes at a price: Today's web applications rely on the use of (untyped) JavaScript, contain client-side programmatic manipulation of the browser's Document Object Model (DOM-tree), and are stateful as well as asynchronous in nature. This combination of techniques (collectively called Ajax [8]) is hard to master, making programming web applications an error-prone endeavor [14].

The purpose of this paper is to investigate how we can improve the dependability of modern web applications by means of *invariants*. Invariants can be used for documentation purposes at the design and code level; they can be used to integrate self-monitoring capabilities into applications; and they can serve as oracles in automatically generated test suites.

Unfortunately, creating invariants manually is difficult and time consuming, which may be one of the reasons that widespread adoption of invariants in practice is not yet achieved [4]. To remedy this, a substantial body of research is available aimed at the automated *generation* of invariants. The best-known of these approaches is Daikon, which can infer all sorts of invariants based on observations of concrete variable values [7].

The central question of this paper is whether we can apply automatic invariant detection to today's web applications, and to what extent this can help to increase the dependability of these applications.

To answer this question, we first look at the pure JavaScript level. We trace JavaScript variables and changes to the browser's DOM-tree, use Daikon to infer invariants from these traces, and show how we can inject the resulting invariants back into the application. The resulting solution is agnostic to server-side technology, and capable of automatically using the detected invariants for testing web applications. Furthermore, we study the DOM-trees in successive user interface states, in order to find commonalities (invariants) within states, across states, and across multiple runs.

We provide an implementation of our approach in an (open source) tool we called InvarScope, as an extension upon our existing Crawljax[1] infrastructure for testing and analysis of web applications. Using this implementation, we evaluate the effectiveness of our approach on five different open source web applications. The evaluation compares the de-

Submitted for review

[1] `http://crawljax.com`

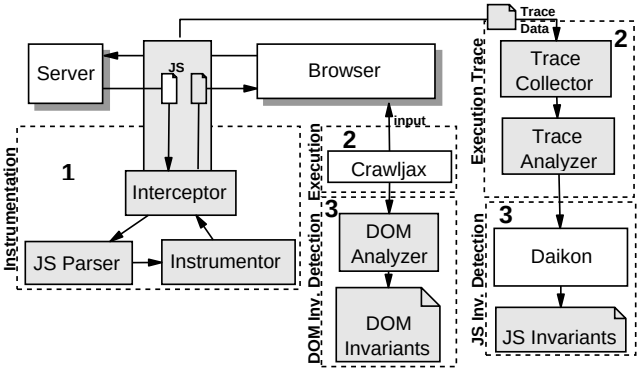

Figure 1: Processing view of our JavaScript and DOM invariant detection approach.

rived invariants to hand-written invariants. It demonstrates that the approach is (1) more effective as the client is more stateful; (2) at least as effective in finding (seeded) faults as hand-written assertions; and (3) capable of covering about 70% of the hand-written assertions.

This paper is further structured as follows. In the next section, we provide an outline of our overall approach for automatically detecting dynamic invariants in web applications. In Sections 3 and 4, we describe the methods for detecting JavaScript and DOM invariants respectively, and how the inferred invariants can be used to test web applications. Section 5 presents the implementation details of our tool InvarScope and Section 6 reports on the results of our empirical evaluation. We follow up with a discussion, related work, and concluding remarks.

2. APPROACH

Our approach for finding invariants in web applications is primarily concerned with the client-side (browser) and it is agnostic to server-side technology. The approach is comprised of two main parts, namely, JavaScript *invariant* and *DOM invariant* detection.

JavaScript is becoming increasing important in modern web applications. Its primary use is to retrieve data from the server, perform some computation, and update the DOM of the page to reflect state changes. The first part of our approach focuses on automatically detecting dynamic JavaScript program invariants by tracing and analyzing client-side JavaScript variables.

The DOM forms a central component of today's dynamic web applications, through which all user interface updates and modifications take place in a browser. Structural invariants over the DOM-tree can act as an oracle for conducting sanity checks on the run-time behavior of web applications [14]. The second part of our approach analyzes client-side user interface state changes to infer dynamic invariants over the DOM-tree.

Similar to most invariant derivation techniques [5, 7], our invariant detection method is based on a workflow composed of the following main steps:

1. Finding a way to log value changes;
2. Executing the program to produce an execution trace;
3. Deriving possible invariants using the trace data that was produced in the previous step.

Figure 1 depicts the architectural view of our overall invariant detection approach. The bold numbers correspond to each of the main steps.

The challenges that we face in this work are imposed by the fact that our execution environment is the browser: To trace JavaScript variable and DOM state changes we need access to the run-time properties of the web application in the browser. To produce an execution trace, the web application needs to be driven in a real web browser. The execution should preferably run as much of the JavaScript code, as possible and execute it in different ways, for instance, with different values for function arguments. It should also visit as many DOM state changes as possible. To find useful invariants, it is necessary to have an extensive execution trace, which would be too expensive to produce by hand. In order to automate the execution step, it is possible to use a test suite that uses browser controlling libraries such as Selenium[2] or Watir.[3] In case test suites are either non-existent or do not produce enough trace data, automated crawling techniques [13] can be used to produce an execution trace, as done in our approach.

Once an execution trace is obtained, the data has to be analyzed to infer likely invariants. Extensive research has been carried out on detecting likely program invariants [3, 6, 7, 9]. Most of these techniques are based on a brute-force method [18]: For all variables, consider all possible invariants to be true, iterate over the list of found values, and remove any invariant item that fails with these values. Our JavaScript invariant detector is based on an extension of the Daikon tool [7], and our DOM invariant detector uses our own brute-force algorithm to analyze and find invariants on the DOM elements and their attributes.

We use the derived JavaScript and DOM invariants for automatically adding assertions in JavaScript source code and creating conformance checkers for the DOM-tree, respectively. These checks can be turned on or off depending on the state of the web application development and testing. During regression testing, with the checks on, the results can be used to debug detected errors. They can provide precise information on where the invariants failed, including the filename, line number, and function name of the program point for JavaScript invariants. For failing DOM invariants, a snapshot of the DOM state as well as the location of the failing invariant on the tree are provided.

3. JAVASCRIPT INVARIANTS

3.1 JavaScript Instrumentation

The approach we have chosen for logging JavaScript variables is on-the-fly JavaScript source code transformation to add instrumentation code. We intercept all the JavaScript code that passes from the server to the browser, using a proxy [2]. First we parse the intercepted source code into an Abstract Syntax Tree (AST). We then traverse the AST in search of program points to add instrumentation code.

JavaScript Program Variables. Our first interest is the range of values of JavaScript variables. We probe function entry and function exit points, by identifying function definitions in the AST and injecting statements at the start, end, and before every `return` statement. We instrument the

[2] `http://code.google.com/p/selenium/`
[3] `http://watir.com`

```
function update() {
/* set the 'class' attribute */
getElementById('contentPane').setAttribute("class",
                                           "red");
...
/* change the 'class' attribute using jQuery */
$(#contentPane).attr('class', "blue");
}
```

Figure 2: DOM manipulation through JavaScript.

```
save(new Array('example.js:::POINT12', addvariable(
   "$('#contentPane').attr('class')",
   $('#contentPane').attr('class')));

$('#contentPane').attr('class', 'blue');

save(new Array('example.js:::POINT13', addvariable(
   "$('#contentPane').attr('class')",
   $('#contentPane').attr('class')));
```

Figure 3: Instrumented JavaScript code for logging DOM modifications.

code to monitor value changes of *global variables*, *function arguments*, and *local variables*.

Per program point, we yield information on *script name*, the *function name* and the *line number*, used for debugging purposes. Per variable we collect information on *name*, *runtime type* and actual *values*. The runtime type is stored because JavaScript is a loosely typed language, i.e., the types of variables cannot be determined syntactically, thus we log the variable types at runtime.

Dynamic DOM Modifications through JavaScript. The other interesting case to include in the execution trace is how certain DOM elements and their attributes are modified at runtime through JavaScript. For instance, by tracing how a `class` attribute of an anchor (i.e., `A`) element is changed during various execution runs, we can infer the range of values for the `class` attribute of the anchor tag.

Based on our observations, DOM modifications are usually exercised in a certain "pattern" through JavaScript. Once the pattern is reverse engineered, we can add proper instrumentation code around the pattern to trace the changes. As an example, Figure 2 shows such a pattern. First, some JavaScript function or framework is used to find the desired DOM element. Next, a function is called on the returned object. This function does the actual modification of the DOM-tree. The range of possible values for the `class` attribute of the `contentPane` element are `red` and `blue` in this example.

Our current approach is capable of recognizing DOM modification patterns that are carried out through standard JavaScript API, as well as the widely used jQuery Ajax library.[4] After recognizing a pattern in the parsed AST, we add instrumentation code that reads and logs the value of the DOM attribute before and after the actual modification.

Figure 3 shows part of the instrumented JavaScript code of the example in Figure 2. The `save` function, is called at each program point and its argument is a new array, which has a program point identifier as the first element, followed by a number of arrays for the variables. These arrays contain the name, type, and value of the variables. The `addvariable` function finds out the types of the variables passed by the

[4] `http://jquery.com`

```
var c = $('#'+id).attr('class');
if (!(c == 'blue' || c == 'red')) {
  /* not true, add an entry to the assertionFailure
     list */
  entry = new Array();
  /* scriptname, functionname, line number */
  entry.push('example.update.21');
  /* invariant that failed */
  entry.push(id + 'attr: class=' +
   $('#'+id).attr('class') + ' not in [blue, red]');
  window.assertionFailures.push(entry);
}
```

Figure 4: JavaScript invariant assertion code.

`save` function, at runtime and creates the arrays.

3.2 Logging the Trace Data

Since JavaScript does not support writing to files from the browser[5] there is no way to communicate with the native file system of the machine running the browser. Hence, saving the trace data generated on the browser poses a challenge.

Keeping the trace data in the browser's memory or sending each data item to the server can practically make the browser very slow, due to the huge amounts of data and high frequency of HTTP requests. Our solution is a hybrid approach, in which we buffer a certain amount of trace data in the memory (in an array), send them to the proxy server as an HTTP request when the buffer's size reaches a predefined threshold, and immediately clear the buffer afterwards. Since the data arrives at the server in a synchronized (ordered) manner, all we have to do on the server is concatenate the tracing data into a single trace file.

3.3 Deriving JavaScript Invariants

In this step, using the obtained trace data, we generate input files for Daikon, and feed those files to Daikon. Daikon then derives likely invariants and the output is saved in a file. We have extended Daikon with support for generating output in JavaScript syntax. Thus the inferred invariants can be used directly to create JavaScript assertions acceptable by the browser's JavaScript engine.

3.4 Using JavaScript Invariants for Testing

Once the invariants are detected, we use them for generating assertions that can be used in regression testing. Some of such assertions need access to local variables within JavaScript functions. Since, generally speaking, most unit testing tools (e.g., Selenium) only provide access to global JavaScript variables in the browser, we had to find a way to gain access to all the variables.

To tackle this problem, we use on-the-fly transformation to inject the assertions directly into the JavaScript code, in a similar fashion as adding the instrumentation code through the proxy. This way the assertions gain access to all the variables needed and can save the results, through the proxy, to generate a test report after a test execution.

Figure 4 shows the automatically injected invariant assertions, checking the `class` attribute of the example shown in Figure 2.

[5] A specification has been proposed `http://dev.w3.org/2006/webapi/FileAPI`.

```
<html>
  <body>
    <h1>First state</h1>
    <p>This page contains:</p>
    <ul id='elementlist' class='list'>
      <li class='menuitem'>A paragraph</li>
      <li class='menuitem'>A list</li>
    </ul>
    <a href='/secondstate/'>Second state</a>
  </body>
</html>
```

Figure 5: String representation of a simple DOM-tree.

```
//HTML
  //BODY
    //H1
    //P
    //UL[@id='elementlist' and @class='list']
      //LI[@class='menuitem']
    //A[@href='/secondstate/']
```

Figure 6: DOM Invariant for Figure 5.

4. TEMPLATE-BASED DOM INVARIANTS

In this section, we present our technique for automatically analyzing the web application's user interface state changes to detect a specific type of invariants in web applications, namely, template-based DOM invariants, i.e, unchanging parts of the user interface. Compared to JavaScript invariants, the main difference here lies in the fact that these invariants are not inferred from *static* program *points* but from *dynamic* program *states*.

4.1 Detecting DOM-based State Changes

In order to infer invariants on the DOM-tree, we first need access to the runtime representation of the tree in the browser, as well as a way to visit different user interface states to be able to compare DOM changes. Our approach for driving the application state in the browser and accessing different DOM states is based on our previous work and Ajax crawling tool Crawljax [14]. Crawljax automates the state change detection phase, thus we can focus on analyzing the dynamic DOM to detect the unchanging parts. More details about the specifics of the crawler can be found in [13, 14]. We analyze the DOM to infer invariants in three different ways: per state, across multiple states, and across multiple runs.

4.2 Invariants per DOM State

Our invariant derivation algorithm for a certain DOM-tree generates an expression in XPath format for each element on the tree, which describes the element, the attributes, and the attribute values. We ignore textual values of elements. The expressions are stored in such a way that the order, parents, and children of each element are retrievable. Figure 6 depicts an example of a derived DOM invariant for the simple DOM instance shown in Figure 5.

4.3 Invariants across Multiple DOM States

To find DOM invariants that hold across multiple DOM states, we use a brute-force algorithm. We first consider every possible invariant to be true, and when a violation occurs in a subsequent DOM state, the invariant is adapted accordingly by removing the failing expression(s). This means the DOM invariant list will decrease or stay equal in size, for every iteration.

Algorithm 1 Inferring DOM invariants per state across multiple runs.

Require: R: Set of runs, S: Set of DOM states for each run
1: **procedure** Derive (S, R)
2: **for** $(i = 1, i =< S.size)$ **do**
3: $inv_{temp} \leftarrow infer(DOM(R_1(S_i)))$
4: **for** $(j = 2, i =< R.size)$ **do**
5: $inv_{temp} \leftarrow compare(inv_{temp}, DOM(R_j(S_i))$
6: **end for**
7: $inv[S_i] \leftarrow inv_{temp}$
8: **end for**
9: **end procedure**
10:
Require: INV: Set of invariants, DOM: DOM-tree, τ: threshold
11: **procedure** Compare (INV, DOM)
12: **for** $(i = 0, i < INV.size)$ **do**
13: $match \leftarrow$ false
14: $invariant \leftarrow INV[i]$
15: $element \leftarrow DOM$.exactMatch($invariant$)
16: **if** $(element == null)$ **then**
17: $element \leftarrow DOM$.fuzzyMatch($invariant$, τ)
18: **end if**
19: **if** $(element$!= $null)$ **then**
20: $children \leftarrow element$.getChildren()
21: **if** ($invariant$.getChildren().equals($children$)) **then**
22: $match \leftarrow$ true
23: **end if**
24: **end if**
25: **if** (!$match$) **then**
26: INV.remove(invariant)
27: **end if**
28: **end for**
29: **return** INV
30: **end procedure**

We start with the DOM-tree of the first state (i.e., *index*) and infer the invariant expressions. From that state the rest of the web application is then automatically crawled and for each detected new state, the corresponding DOM-tree is used to check which elements are changed (or missing) and need to be removed from the DOM invariant instance. Figure 7 depicts our DOM invariant approach schematically. For this step, the invariants are inferred by comparing the different DOM states, from left to right.

4.4 Invariants across Multiple Execution Runs

Modern Ajax user interfaces are dynamic in nature, i.e., revisiting the same state n times, can result in m subtle differences in terms of content and structure [21]. To improve the quality of the inferred DOM invariants for coping with this dynamic characteristic, we infer the invariants through multiple execution runs of the web application. This is shown in Figure 7 by comparing each state vertically in different execution runs.

Algorithm 1 shows our algorithm for inferring invariants across multiple execution runs. For each state S_i, the corresponding DOM states from all available runs are compared against the inferred invariant document at that moment, as shown in the `DERIVE` procedure. At each iteration, the invariants are adjusted if necessary to reflect dynamic conditions. Eventually, after a number of execution runs, merely the unchanging parts of each S_i remain in the corresponding invariant instance.

Once robust invariants have been detected for each dynamic state, we derive the DOM invariant that holds over all the detected invariants, resulting in a *site-wide DOM*

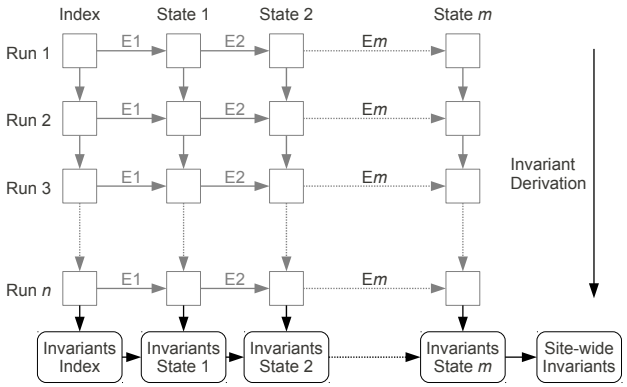

Figure 7: DOM invariant derivation across multiple states and execution runs.

invariant. This site-wide invariant captures the unchanging skeleton or template of the whole web application DOM state space.

4.5 Matching DOM Invariants

To cope with subtle differences when finding invariants over multiple states, our algorithm checks the DOM-tree at each new state against the already detected DOM invariants at that moment. This checking algorithm is presented in Algorithm 1 (lines 11-30), which is described in the following paragraphs.

Exact Matching. The algorithm first checks whether elements of the invariant can be found in the DOM-tree using the corresponding XPath expressions. These expressions search for a match in the current DOM-tree using the element type and attributes. For instance, the algorithm tries to resolve `//UL[@id='elementlist' and @class='list']` to a DOM element. If no match is found, the algorithm moves to the fuzzy matching phase.

Fuzzy Matching. The fuzzy matching call (line 17) first tries to find elements of the same type, by searching for the XPath expressions without the attributes. For instance, in the list example, we search only for `//UL`. This expression will return all present `UL` elements from the DOM-tree. For each of these elements, we then compare the attributes and their values with the attributes and values of the invariant element.

The comparison is carried out by calculating the number of equal characters in the same order, which are contained in the two inputs (the attribute value of the invariant element and the attribute value of the actual DOM element). After finding the number of equal characters, we use the Sørensen similarity index [15] to compare these two samples. The Sørensen similarity index is calculated using the formula:

$$index = \frac{2a}{(b+c)}$$

in which b and c represent the total number of characters of each input and a is the number of equal characters found in both b and c. For each DOM element, we take the average of all Sørensen indices for its attribute values. The elements are considered equal if this average is larger than a threshold τ ($0 <= \tau <= 1$).

Matching Based on Children. When at least one of the previous methods (exact or fuzzy) finds a match (line 19), we use the children of both the found DOM element (line 20) and the invariant to assess the quality of the match. If the children are matched (checked recursively), we can call it a match, i.e., the DOM element has the same tag name and children as the invariant, so there is a high probability that we have found a match.

Consider the DOM invariant shown in Figure 6 as the current invariant. Running a simple "exact match" against the `UL` element of Figure 5 will return one element. Next, the children matching algorithm is used to check whether the element has the same type of children as in the DOM invariant. This turns out to be true, and thus we consider the element to be present. If all the approaches fail to find a match, an invariant is violated and that invariant element is removed from the list (line 26).

4.6 Using DOM Invariants for Testing

The inferred invariant document can be used, for instance, for search engine optimization (e.g., every page should contain a H1), accessibility testing (e.g., menu must appear after content), and regression testing.

In order to find the correct inferred invariant document $INV[S_i]$ for a state S_i during testing, we use the source state along with the event causing the state transition to S_i as guidelines. Checking the invariants against the DOM-tree of S_i is carried out using the same algorithm as described in Section 4.5, meaning that invariants fail when both the exact and the fuzzy algorithm fail or the children can not be correctly mapped. As an extra check, the order of the elements are checked to determine whether the DOM elements are at the correct position according to the invariants.

The failures found by the testing algorithm are saved in a report with detailed data about the failure, e.g, the current DOM, the XPath and the XPath of the children, and the failing invariant. This information is vital in making the violations traceable.

5. TOOL IMPLEMENTATION

We have implemented our JavaScript and DOM invariant detection approach in a tool called InvarScope. InvarScope is released as open source and is available for download. [6]

In the JavaScript instrumentation component, we use Mozilla Rhino[7] to parse JavaScript code to AST, and back to the source code after instrumentation. The AST generated by Rhino's parser has traversal API's, which we use to search for program points where instrumentation code has to be added. For instrumenting JavaScript code on-the-fly, we use an integrated version [2] of WebScarab's proxy,[8] to intercept all the incoming requests from the server. This enables us to analyze and modify the content of responses that contain JavaScript code, embedded in HTML files or in separate JavaScript files.

To generate an execution trace of the JavaScript code and analyze the dynamic DOM changes, once the JavaScript code is instrumented and sent to the browser, we use Crawljax [13, 14] to automatically crawl the web application. Crawljax opens the web application in a real browser,

[6] http://spci.st.ewi.tudelft.nl/content/invarscope/

[7] http://www.mozilla.org/rhino/

[8] http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

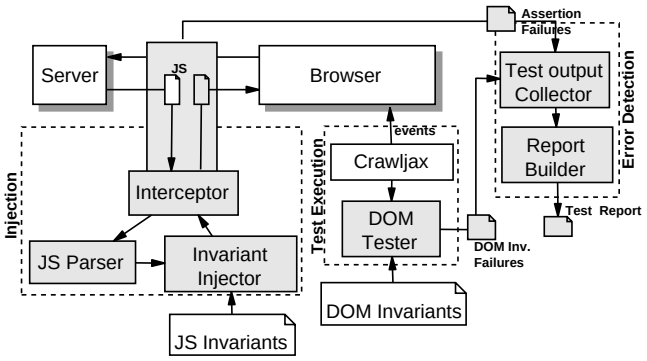

Figure 8: Processing view of the invariant checking phase of InvarScope.

dynamically finds all the present clickable elements for each state, and executes them to find new states recursively. This way, we explore the web application's user interface state in the browser systematically to produce a large enough execution trace. Subsequently, we feed the produced JavaScript execution trace to Daikon [7] to detect JavaScript invariants.

Our DOM invariant detection algorithm is implemented as a plugin for Crawljax. This plugin is executed every time Crawljax visits a new state. For each new state, it obtains the current DOM-tree and reverse engineers the XPath invariants for each DOM element. The click-path, a concatenation of the names and XPaths of all elements that were clicked to reach that state, is used to derive a unique filename to store the invariants. In each run, the invariant finding/checking algorithm is applied and the results using the unique filename are stored.

Figure 8 shows how the detected JavaScript and DOM invariants are used for regression testing. The JavaScript invariants are injected into the JavaScript source code as assertions using the proxy (similar to 3.4). The DOM invariants are checked in each newly detected state as well. Any failure detected is stored in a test report.

6. EMPIRICAL EVALUATION

In order to evaluate our approach, we have conducted case studies for assessing the (1) JavaScript (code-level) invariants and (2) DOM (structural) invariants detected by our technique. The research questions can be summarized as follows:

RQ1 In what kind of web applications can we automatically derive invariants?

RQ2 How useful are the automatically derived invariants?

RQ3 What is the manual effort involved, in comparison to hand-written invariants?

Our (additional) experimental data are available at this link. [6]

6.1 Study 1: JavaScript Invariants

Experimental Subjects. For our first study, our selection criteria for experimental subjects include web applications that make use of JavaScript and standard HTML (no flash) on the browser.

Table 2: Manually and automatically found JavaScript invariants.

	Same Game	Tunnel	Organizer
Custom JS code (LOC)	250	370	310
Automatically Found Invariants			
Total Number of Invariants	150	3852	10
- Function Entry	53	1531	4
- Function Exit	91	2319	6
- DOM Manipulations	6	2	0
Unique Invariants	34	291	6
Manual Effort (minutes)	4	2	3
Manually Found Invariants			
Total Number of Invariants	30	20	5
- Function Entry	13	10	3
- Function Exit	7	3	1
- In Middle of Function	10	7	1
Manual Effort (minutes)	70	60	20
Fault Detection			
Detected by Automatic Inv.	50%	80%	-
Detected by Manual Inv.	40%	40%	-

Same Game: Our first subject is a web-based implementation of the *Same Game*[9] puzzle. This game was implemented, using jQuery, by two graduate students of our group who have experience in developing modern web applications. It consists of about 250 lines of custom JavaScript code.[10]

Tunnel: Our second subject is an open source web-based implementation of a tunnel game.[11] In this game, the player controls an airplane and the objective is to avoid hitting a moving wall. It is written using jQuery and consists of about 370 custom lines of JavaScript code.

The Organizer: The Organizer is an open source web application[12] that can be used as a task manager and organizer. It is written as a Java EE application using WebWork, Spring JDBC, and the Prototype Ajax library.

Setup. To address RQ1, we selected two extreme subjects (*Same Game* and *Tunnel*), in which JavaScript code is used extensively and the entire state is maintained in the browser. We also included the third case (*The Organizer*), as an instance of a simple Ajax web application, to make a comparison. To answer RQ2, we decided to compare automatically derived invariants with invariants that are found manually by developers. Therefore, we asked the students to manually examine each subject and document possible JavaScript invariants as well as the time taken to come up with the invariants. In total 30 manual invariants were documented for *Same Game* (70 minutes), 20 for the *Tunnel* (60 minutes), and 5 for *The Organizer* (20 minutes). Afterwards, we ran InvarScope on each subject to automatically detect invariants. To answer RQ3, we documented the time that was needed to infer the invariants manually and automatically. For each web application, InvarScope instrumented the custom JavaScript code and dynamically crawled the states to produce an execution trace. The trace data for Same Game, Tunnel, and The Organizer became 11 MB, 63 MB, and 3 MB respectively. The traces were then subsequently fed to Daikon to infer likely invariants.

Detected Invariants. The results are shown in Ta-

[9] http://en.wikipedia.org/wiki/SameGame

[10] http://crawljax.com/same-game

[11] http://arcade.christianmontoya.com/tunnel/

[12] http://www.apress.com/book/downloadfile/2931

Table 1: Examples of JavaScript faults injected and detected (Study 1).

Subject	Fault injected	Aut. Inv.	Man. Inv.
Same Game	`x` and `y` arguments of the `mark` function call were swapped	✓	✓
	Dash was removed from HTML attribute of the cell, `x` and `y` coordinates were concatenated without a separator.	X	X
	The `updateBoard` function draws the board and checks whether the game has finished. The check whether all colors are removed was modified to always evaluate to `true`. This change has the annoying effect that every time a cell is clicked, a "game won" message is displayed	✓	X
Tunnel	The starting score of the game was changed to be negative instead of zero.	✓	✓
	Code that is used to verify that the tunnel is never wider than a certain value was removed. This means the plane can be kept in the middle of the screen without ever touching the tunnel.	X	X
	The code that modifies the position of the plane was removed. This means the plane could not be moved anymore.	✓	X
	The score increment rate was changed to be much faster, almost equal to the frame rate.	✓	X

ble 2. In the *Same Game*, InvarScope inspected 44 program points, of which 10 where function entry points, 18 were function exit points, and 16 were DOM modification points. For these program points, a total of 150 invariants were found, of which 34 were unique ones. For the function entry and exit points, we found approximately 5 assertions per point. Some interesting invariants were found: the method that marks a cell as "to be removed" has three parameters, namely, two coordinates and the color of the cell that is clicked. InvarScope came up with invariants that made sure the x and y coordinates were always valid, i.e., `x >= 0`, `width > x`, `y >= 0` and `height > y`. These invariants are very useful to detect off-by-one errors. Furthermore, it found an invariant to make sure the `value` (color) argument was a valid color: `(value == 1.0 || value == 2.0 || value == 3.0)`. This invariant also makes sure we cannot mark cells that were already empty, because those cells have a color value of zero.

Another interesting invariant detected is the fact that the `height` and `width` variables that define the board size are considered to always be equal to some constant value, which is exactly what the developers of the application had documented as an invariant. Furthermore, some interesting DOM modification invariants were detected. For example, a function that adds the clickable *class* attribute to elements that, according to the game rules, should be clickable was extended with an invariant check to verify the class was actually added to the elements.

For the *Tunnel* application, in total 102 program points were inspected, 51 entry points and 51 exit points. The total number of invariants found was 3852, of which 291 were unique. Approximately 37 assertions were found per program point. Some of the interesting results includes, for instance, invariants to check that the plane is always positioned between the two walls and the space between the two walls is always large enough (280, 300 or 320 pixels). However, some false positives were also detected. One example is the check `score < ship_x`. When the user plays long enough, he might get a score that is higher than the x coordinates of the ship, which will result in failures. To avoid these kind of false positives, a more extensive execution trace will probably help, since it can invalidate these invariants.

Merely 10 invariants were automatically discovered in *The Organizer*, mostly constants, out of which only 6 were unique. Most of the JavaScript code contains only Ajax functions to retrieve data from the server and update the page, which makes it quite difficult to manually document JavaScript invariants for this subject as well.

Fault Injection. To assess the usefulness of the invariants, the students were first asked to turn the manually detected invariants into actual assertions in the JavaScript code of each subject. Then, they were asked to inject 10 faults in the web applications, without telling us about their nature. Mutation testing has been proven to be an effective vehicle for assessing the quality of a testing technique [1, 23]. Hence, mutation operators (mostly taken from [1, 16]) were also manually injected into the subjects: Replace an integer constant C by 0, 1, -1, ((C)+1), or ((C)-1); Flip an arithmetic, relational, logical, increment/decrement, or arithmetic-assignment operator; Negate the decision in an `if` or `while` statement; Delete a statement; Swap function arguments.

Afterwards, each faulty version of the applications was first checked with the manually added assertions. Then that same faulty version was tested by InvarScope. InvarScope automatically inserted the detected invariants in the JavaScript code through the proxy and ran the application. The number and type of errors detected by each approach (manual and automatic) were noted.

Table 1 shows some examples of the faults introduced in *Same Game* and *Tunnel* and how they were detected by the manual and automatic invariant assertions. Since the number of invariants was not significant in *The Organizer*, we ignored the fault seeding phase for this experimental subject. In total, the automatically detected invariants found 50% of the seeded faults in the *Same Game* and 80% of them in the *Tunnel*. The manual invariants detected 40% of the faults in both the *Same Game* and *Tunnel*.

Findings. Based on our experimental results we can conclude that:

- It is possible to automatically derive program invariants in web applications that make extensive use of JavaScript in the browser. The higher the degree of the state on the client, the more invariants are likely to be found. This is evident from the two games, which have their entire application state in the browser. *The Organizer* on the other hand only retrieves data from the server and updates the DOM-tree using JavaScript, and thus does not result in many invariants.
- The automatically detected invariants are of such quality that it is possible to use them as assertions for automatic fault detection. They score even higher than manually written invariant assertions.
- The amount of manual effort required to automatically instrument, trace, and infer invariants is only a frac-

Table 3: Examples of DOM faults injected and detected (Study 2).

Subject	Fault injected	Detected by DOM Inv.
The Organizer	modified the menu that is shown on all pages	✓
	removed the image tag of the logo from the header.	✓
	replaced the menu, which is located in a table row, to the bottom of the table. This means that the menu is shown at the bottom of the page instead of at the top	✓
	re-ordered elements	✓
Bookstore	re-ordered elements of registration form	✓
	replaced the search block	✓
Yellow Pages	removed enclosing `TR` and `TD` elements of a link in the menu.	X

tion of the time needed for manually looking for and documenting invariants.

- We also compared the manually defined invariants with those derived automatically. Similar to the discoveries of Polikarpova *et al.* [19], the automatically detected invariants cover about 70% of the manually written assertion clauses in our case studies.

6.2 Study 2: DOM Invariants

Experimental Subjects. To evaluate our DOM invariant derivation approach, we used three subjects. The first one is The Organizer as explained in Section 6.1.

The second subject is called Bookstore,[13] which is an open source web application that can be used to sell books online. It includes a user registration system, product voting, categories, shopping cart, and administration of various web shop aspects.

The third subject is Yellow Pages,[14] an open source web application in which the user can find contact information by browsing different categories or searching for specific terms.

Setup and Results. The DOM invariant detection plugin in InvarScope was used in three runs to generate invariants on the DOM-tree of each web application. We set the threshold (τ) to be 0.7, which provided the best results for this case study. Here, we were mainly interested in how useful the inferred DOM invariants are (RQ2). Thus again we asked our students to inject a number of faults affecting the DOM into each of our experimental subjects. Table 3 shows some examples of the types of DOM faults injected (remove, modify, and replace DOM elements) and whether they were detected by the invariants.

Each faulty version was automatically checked by our tool and the results can be seen in Table 4. For each subject, the table presents the total number of dynamic states and DOM elements examined, the total number of inferred invariants, the minimal and maximal state invariants, and the number of faults injected and successfully detected.

Findings. Based on our data we can conclude that:

- It is possible to automatically derive DOM invariants in any type of web application that is based on HTML and DOM (this excludes Flash and other proprietary implementations). In our case study, we found a huge number of invariants automatically by analyzing the DOM-trees.
- The detected invariants can indeed be used for automatic fault detection and regression testing. We were able to detect around 85% of the injected faults automatically.

[13] `http://gotocode.com/apps.asp?app_id=3`

[14] `http://gotocode.com/apps.asp?app_id=4`

Table 4: DOM evaluation results after 3 runs.

	The Organizer	Bookstore	Yellow Pages
Dynamic States	20	67	93
DOM Elements	2957	8914	10731
Invariants Detected	2389	4717	4843
Minimal Number of Invariants	118	68	41
Maximal Number of Invariants	120	110	57
Injected Faults	14	10	12
Detected Faults	12	9	9

7. DISCUSSION

Applicability. During the development and evaluation of our JavaScript invariant deriver we found out that it cannot find satisfying results for all types of web applications. For example, simple websites that use JavaScript to merely show and hide HTML elements are not good candidates. We believe the best applications are computation intensive web applications that carry out most of the computation and maintain a significant part of the application state on the client side. Our DOM invariant deriver is, however, applicable to all kinds of web applications, varying from static web pages to very dynamic web applications. The DOM invariants inferred are a very specific type of structural invariant (template-based) and thus not capable of capturing the exact structure of complex DOM nodes. Our testing and invariant derivation methods are fairly generic, i.e., they can be used with manual web application "crawling" or automation tools such as Crawljax or Selenium.

Generated, Compressed, or Obfuscated JavaScript. A number of frameworks, such as the Google Web Toolkit (GWT), exist that automatically generate most of the client-side code. While our approach infers useful invariants for hand-written JavaScript code, invariants or assertion failures found in generated code are not meaningful to developers, because errors detected by the invariants cannot easily be traced back to their source, e.g., Java for GWT applications.

The use of a proxy to intercept JavaScript source code infers a limitation as well, namely the fact that it can merely be used for web applications that use no encrypted connections. Finally, during the evaluation, we found out that minified, compressed, or obfuscated JavaScript files might not be parsed correctly with Rhino and thus no representative execution trace can be obtained.

Threats to Validity. Concerning *external validity*, our study is performed on a limited number of web applications. Generalizing the results based on these studies might harm

validity, although the selected cases represent the type of web applications targeted by our research. We did conduct more case studies on different kinds of web applications, but we had problems in correctly deriving invariants due to bugs in Rhino, Daikon and our own tool implementation. A list of these cases and the cause of their failure can be found at. [6] Our DOM modification through JavaScript implementation is currently limited to web applications that use plain JavaScript (without any libraries) or jQuery. Adding support for recognizing patterns of other libraries, such as Dojo or Prototype, can be added without much effort, however, the results need to be evaluated in the future.

The fault injection was carried out by two graduate students, and although they have experience in developing Ajax web applications, they do not replace real web developers and the way they document invariants for web applications. Another threat to validity could be the nature of the faults injected. Although some of the injected faults are actual faults, mutation testing was also used to produce different (faulty) versions of the web applications. The use of mutation operators is, however, shown to yield trustworthy results [1] in empirical assessments of test techniques.

With respect to *internal validity*, we tried to minimize the number of bugs in the tools developed by writing JUnit tests for the JavaScript invariant deriver and tester. However, we also use various third party tools. We did encounter several problems in some of them, so these libraries and tools might harm the internal validity of our results.

As far as *reliability* is concerned, all the web applications used in the evaluation as well as InvarScope are publically available.

8. RELATED WORK

Dynamic Invariants. The concept of using invariants to assert program behavior at run-time is as old as programming itself [4]. A more recent promising development is the automatic detection of dynamic invariants through dynamic analysis. Ernst *et al.* have developed Daikon [7], a tool capable of inferring *likely invariants* from program execution traces of several languages, including Java, C, and C++. To derive the JavaScript invariants, we have used and extended Daikon with support for JavaScript. Other related similar tools that detect invariants include Agitator [3], DIDUCE [9], and DySy [6]. DySy is somewhat different than the rest, since it is based on an algorithm that uses symbolic execution of the program as well as its concrete execution to detect likely invariants. Lorenzoli *et al.* [12] generate behavioral models from program executions in the form of EFSMs, which represent constraints on data values, and properties of interaction patterns and their interplay. Swaddler [5] is an invariant detection tool for PHP that uses Daikon.

JavaScript Analysis. Most of the existing work on JavaScript analysis is focused on detecting security vulnerabilities in dynamic web applications.

Kudzu [22] is a symbolic execution system for JavaScript aimed at automated security vulnerability analysis. This is done by automatically generating a test suite using symbolic execution of the JavaScript source code. This test suite can then be used to search for client-side code injection vulnerabilities. BrowserShield [20] applies dynamic instrumentation to rewrite JavaScript code to conduct vulnerability driven filtering using so-called policies. These policies are basic JavaScript functions that can, for example, disable the construction of certain vulnerable ActiveX objects. Yu *et al.* [24] propose a method in which untrusted JavaScript code is analyzed and instrumented [11] to identify and modify questionable behavior.

JavaScript instrumentation has also been applied to monitor client-side behavior. AjaxScope [10] is a dynamic instrumentation platform that enables cross-user monitoring and just-in-time control of web application behavior.

To the best of our knowledge, our work is the first to instrument JavaScript code to produce an execution trace and infer dynamic invariants from dynamic web applications.

DOM Analysis. In our previous work [14], we proposed, Atusa, an approach to use generic and application-specific invariants on the DOM-tree to detect faulty states in web applications. DoDOM [17] is a recently developed tool to infer DOM invariants. Our approach differs from DoDOM in a few aspects. DoDOM needs an actual user to interact with the web application to produce invariants. DoDOM derives invariants over a number of state transitions, while our tool derives invariants per state, and site-wide invariants. Finally, our tool analyzes DOM elements and their children, while DoDOM looks at the DOM element and its content, ignoring the children.

9. CONCLUDING REMARKS

Thanks to recent advances made in browser and web technologies, more and more applications are moved to the web. The enabling client-side technologies such as JavaScript and the DOM-tree, however, enlarge the complexity and pose an increasing threat to dependability of such applications. In this paper, we proposed a technique in which program and structural invariants can automatically be inferred through dynamic analysis of web applications. The invariants detected as such form a useful vehicle for regression testing. The contributions of this paper can be summarized as:

- A method for detecting JavaScript invariants by instrumenting JavaScript code and tracing program state changes, including programmatic manipulation of DOM elements and their attributes;
- An algorithm for automatically analyzing web user interface changes to detect template-based DOM invariants per state, across different states, and across multiple program executions;
- The implementation of our technique in an open source tool called InvarScope. InvarScope is integrated as plugins in our Ajax crawling and testing tool Crawljax, providing a mechanism in which the derived invariants can be used for regression testing of web applications;
- An empirical evaluation, by means of a number of case studies, to demonstrate the efficacy and application of our approach.

Future work encompasses conducting more case studies, especially on industrial web applications. Augmenting our JavaScript DOM modifications detector so that it is capable of coping with more patterns in other JavaScript

libraries forms part of our future work as well. Another direction we foresee is capturing the exact structure of complex DOM structures in invariants. We currently only remove elements from the initial invariant, in a brute-force manner. We intend to add/remove elements based on their statistical stability across multiple states/runs in the future.

10. REFERENCES

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering (ICSE'05)*, pages 402–411. ACM, 2005.

[2] C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of ESEC/FSE'09*, pages 81–91, New York, NY, USA, 2009. ACM.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180, New York, NY, USA, 2006. ACM Press.

[4] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[5] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, 2007.

[6] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 281–290. ACM, 2008.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[8] J. J. Garrett. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, February 2005.

[9] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 291–301. ACM Press, 2002.

[10] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP'07: Proceedings of 21st ACM SIGOPS symposium on Operating systems principles*, pages 17–30. ACM, 2007.

[11] H. Kikuchi, D. Yu, A. Chander, and H. I. I. Serikov. JavaScript instrumentation in practice. In *APLAS'08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 326–341. Springer-Verlag, 2008.

[12] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 501–510. ACM, 2008.

[13] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, July 2008.

[14] A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 210–220. IEEE Computer Society, 2009.

[15] D. Mueller-Dombois and H. Ellenberg. *Aims and methods of vegetation ecology*. Wiley, Chichester, UK, 1974.

[16] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, 1996.

[17] K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM invariants for web 2.0 application reliability. Technical Report MSR-TR-2009-176, Microsoft Research, 2009.

[18] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32, November 2–4, 2004.

[19] N. Polikarpova, I. Ciupa, and B. Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA'09)*, pages 93–104. ACM, 2009.

[20] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11–43, 2007.

[21] D. Roest, A. Mesbah, and A. van Deursen. Regression testing Ajax applications: Coping with dynamism. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*, pages 128–136. IEEE Computer Society, 2010.

[22] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[23] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA'09)*, pages 69–80. ACM, 2009.

[24] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'07)*, pages 237–249. ACM, 2007.

TUD-SERG-2010-037
ISSN 1872-5392