JSART: JavaScript Assertion-based Regression Testing

Shabnam Mirshokraie and Ali Mesbah

University of British Columbia
Vancouver, BC, Canada
{shabnamm,amesbah}@ece.ubc.ca

Abstract. Web 2.0 applications rely heavily on JAVASCRIPT and client-side runtime manipulation of the DOM tree. One way to provide assurance about the correctness of such highly evolving and dynamic applications is through regression testing. However, JAVASCRIPT is loosely typed, dynamic, error-prone, and challenging to analyze and test. We propose an automated technique for JAVASCRIPT regression testing, which is based on on-the-fly JAVASCRIPT source code instrumentation and dynamic analysis to infer invariant assertions. These obtained assertions are injected back into the JAVASCRIPT code to uncover regression faults in subsequent revisions of the web application under test. Our approach is implemented in a tool called JSART. We present our case study conducted on nine open source web applications to evaluate the proposed approach. The results show that JSART is able to effectively infer stable assertions and detect regression faults with a high degree of accuracy and minimal performance overhead.

Keywords: JavaScript, regression testing, invariants, assertions, dynamic analysis

1 Introduction

JAVASCRIPT is increasingly being used to create modern interactive web applications that offload a considerable amount of their execution to the client-side. JAVASCRIPT is a notoriously challenging language for web developers to use, maintain, analyze and test. It is dynamic, loosely typed, and asynchronous. In addition, it is extensively used to interact with the DOM tree at runtime for user interface state updates.

Web applications usually evolve fast by going through rapid development cycles and are, therefore, susceptible to regressions, i.e., new faults in existing functionality after changes have been made to the system. One way of ensuring that such modifications (e.g., bug fixes, patches) have not introduced new faults in the modified system is through systematic regression testing [3]. Regression testing is also used to determine whether a change in one part of the software affects other parts of the software. While regression testing of classical web applications has been difficult because of the dynamism in web interfaces, modern JAVASCRIPT-based applications pose an even greater challenge [19]. To the best of our knowledge, JAVASCRIPT regression testing has not been addressed in the literature yet.

In this paper, we propose an automated technique for JAVASCRIPT regression testing, which is based on dynamic analysis to infer invariant assertions. These obtained

```
1  function setDim(height, width) {
2   var h = 4*height, w = 2*width;
3   ...
4   return(h:h, w:w);
5  }

7  function play(){
8   $(#end).css("height", setDim($('body').width(), $('body').height()).h + 'px');
9   ...
10 }
```

Fig. 1. Motivating JAVASCRIPT example.

assertions are injected back into the JAVASCRIPT code to uncover regression faults in subsequent revisions of the web application under test. Our technique automatically (1) intercepts and instruments JAVASCRIPT on-the-fly to add tracing code (2) navigates the web application to produce execution traces, (3) generates dynamic invariants from the trace data, (4) transforms the invariants into stable assertions and injects them back into the web application for regression testing.

Our approach is orthogonal to server-side technology, and it requires no manual modification of the source code. It is implemented in an open source tool called JSART (JAVASCRIPT Assertion-based Regression Testing). We have empirically evaluated the technique on nine open-source web applications. The results of our evaluation show that the approach generates stable invariant assertions, which are capable of spotting injected faults with a high rate of accuracy.

2 Motivation and Challenges

Figure 1 shows a simple JAVASCRIPT code snippet. Our motivating example consists of two functions, called `setDim` and `play`. The `setDim` function has two parameters, namely `height` and `width`, with a simple mathematical operation (Line 2). The function returns local variables, `h` and `w` (Line 4). `setDim` is called in the `play` function (Line 8) to set the `height` value of the CSS property of the DOM element with ID `end`. Any changes to `height` and `width` would influence the returned values of `setDim` as well as the property of the DOM element. One possible error that can happen is to mistakenly swap the order of `height` and `width` when they are respectively assigned to local variables `h` and `w`. A similar error can occur when the `setDim` is called in the `play` function by changing the order of function arguments. Such regression faults happen in practice when a programmer modifies parts of the original code while updating or adding new functionality. Detecting such regression errors is a daunting task for web developers, especially in programming languages such as JAVASCRIPT, which are known to be challenging to test due to their loosely typed, dynamic, and asynchronous nature.

One way to check for these regressions is to define invariant expressions of expected behaviour over program variables and assert their correctness at runtime. This way any modification to `height`, `width`, `h`, or `w` that violates the invariant expression will be detected. However, manually expressing such invariant assertions for web applications with thousands of lines of JAVASCRIPT code and several DOM states, is very challenging and time-consuming. Our aim in this work is to provide a technique that automatically captures regression faults through generated JAVASCRIPT assertions.

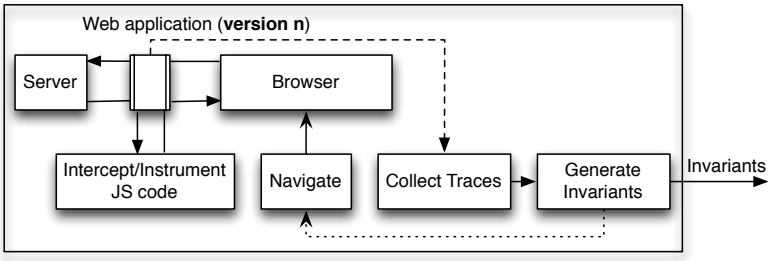

Fig. 2. Overview of the JAVASCRIPT tracing and invariant generation steps (web application version *n*).

3 Our Approach

Our regression testing approach is based on *dynamic analysis* of JAVASCRIPT code to infer invariants from a given web application. We use the thus obtained invariants as *runtime assertions* in the JAVASCRIPT code to automatically uncover regression errors that can be introduced after changes have been made to the web application in a subsequent reversion. Our approach is largely based on two assumptions (1) the current version of the web application, from which invariants are being generated, is bug-free (2) the inferred invariants capture program specifications that are unlikely to change frequently in the following revisions. Our regression testing technique is composed of the following main steps:

1. **JavaScript Tracing:** keeps track of and logs value changes during program execution;
2. **Invariant Generation:** infers program expressions that remain unchanged during multiple executions;
3. **Filtering Unstable Invariant Assertions:** removes reported false positives, i.e., any falsely detected invariant;
4. **Regression Testing through Invariant Assertions:** injects the generated invariants into the web application's code as assertions for detecting faults on a modified version of the web application;

Our technique is concerned with the client-side code (e.g., JAVASCRIPT, DOM) and is orthogonal to server-side technology. In the following subsections, we will describe each step in details.

3.1 JAVASCRIPT Tracing

In order to infer useful program invariants, we need to collect execution traces of the JAVASCRIPT code. The idea is to log as much program variable value changes at runtime as possible. Figure 2 depicts a block diagram of the tracing step. Our approach automatically generates trace data in three subsequent steps: (i) JAVASCRIPT interception and instrumentation, (ii) navigation, and (iii) trace collection. In the following, we explain each step in details.

JAVASCRIPT Interception and Instrumentation. The approach we have chosen for logging variables is on-the-fly JAVASCRIPT source code transformation to add instrumentation code. We intercept all the JAVASCRIPT code of a given web application, both in JAVASCRIPT files and HTML pages, by setting up a proxy [2] between the server and the browser. We first parse the intercepted source code into an Abstract Syntax Tree (AST). We then traverse the AST in search of interesting program points to add our instrumentation code. We instrument *program variables* as well as *DOM modifications* as described below.

Tracing Program Variables. Our first interest is the range of values of JAVASCRIPT program 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 code to monitor value changes of *global variables*, *function arguments*, and *local variables*. Per program point, we yield information on *script name*, *function name*, and *line number*, used for debugging purposes. Going back to our running example (Figure 1), our technique adds instrumentation code to trace `width`, `height`, `h`, and `w`. For each 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.

Tracing DOM Modifications. In modern web applications, JAVASCRIPT code frequently interacts with the DOM to update the client-side user interface state. Our recent study [15] of four bug-tracking systems indicated that DOM-related errors form 80% of all reported JAVASCRIPT errors. Therefore, we include in our execution trace how DOM elements and their attributes are modified by JAVASCRIPT at runtime. For instance, by tracing how the CSS property of the 'end' DOM element in Figure 1 is changed during various execution runs, we can infer the range of values for the `height` attribute.

Based on our observations, JAVASCRIPT DOM modifications usually follow a certain pattern. Once the pattern is reverse engineered, we can add proper instrumentation code around the pattern to trace the changes. In the patterns that we observed, first a JAVASCRIPT API is used to find the desired DOM element. Next, a function is called on the returned object responsible for the actual modification of the DOM-tree. After recognizing a pattern in the parsed AST, we add instrumentation code that records the value of the DOM attribute before and after the actual modification. Hence, we are able to trace DOM modifications that happen programmatically through JAVASCRIPT.

Navigation. Once the AST is instrumented, we serialize it back to the corresponding JAVASCRIPT source code file and pass it to the browser. Next, we navigate the application in the browser to produce an execution trace. The application can be navigated in different ways including (1) manual clicking (2) test case execution (3) or using a web crawler. To automate the approach, our technique is based on automated dynamic crawling [13]. The execution needs to run as much of the JAVASCRIPT code as possible and execute it in various ways. This can be achieved through visiting as many DOM state changes as possible as well as providing different values for function arguments.

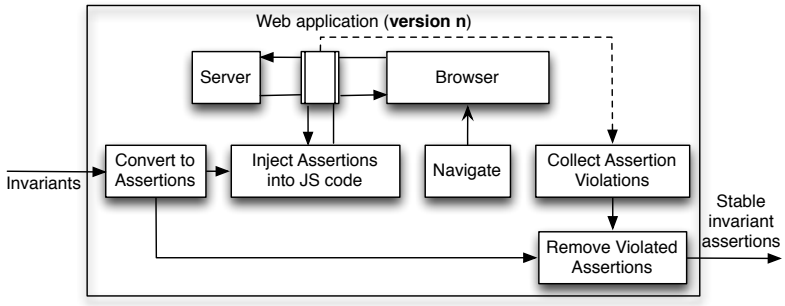

Fig. 3. Overview of the filtering step to remove unstable invariant assertions, for web application version *n*.

Trace Collection. As the web application is navigated, the instrumented JAVASCRIPT code produces trace data, which needs to be collected for further analysis. Keeping the trace data in the browser's memory during the program execution can make the browser slow when a large amount of trace data is being produced. On the other hand, sending data items to the proxy as soon as the item is generated, can put a heavy load on the proxy, due to the frequency of HTTP requests. In order to tackle the aforementioned challenges, we buffer a certain amount of trace data in the memory in an array, post the data as an HTTP request to a proxy server when the buffer's size reaches a predefined threshold, and immediately clear the buffer in the browser's memory afterwards. Since the data arrives at the server in a synchronized manner, we concatenate the tracing data into a single trace file on the server side, which is then seeded into the next step (See Figure 2).

3.2 Invariant Generation

The invariant generation phase is involved with analyzing the collected execution traces to extract invariants. Substantial amount of research has been carried out on detecting dynamic program invariants [4, 6, 7, 10]. Similar to these techniques, our approach is based on a brute-force method: 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 any of these values.

As indicated with the dotted line in Figure 2, we cycle through the navigation and invariant generation phases until the size of generated invariant file remains unchanged, which is an indication that all possible invariants have been detected.

3.3 Filtering Unstable Invariant Assertions

The next step is to make sure that the generated invariants are truly invariants. An invariant assertion is called unstable when it is falsely reported as a violated assertion. Such assertions result in producing a number of false positive errors during the testing phase. To check the stability of the inferred invariants, we use them in the same version of the web application as assertions. Theoretically, no assertion violations should

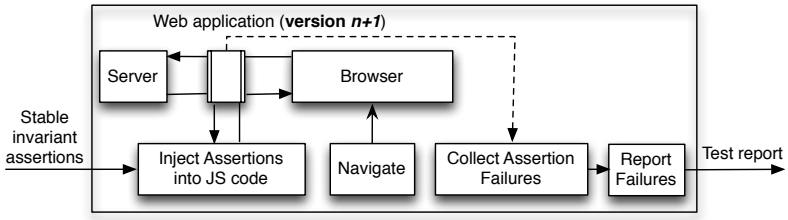

Fig. 4. Overview of the JAVASCRIPT regression testing step through invariant assertions, for web application version *n+1*.

be reported because the web application has not changed. Hence, any assertion violation reported as such is a false positive and should be eliminated. Our filtering process, shown in Figure 3, consists of the following four processes:

- Converting the inferred invariants into checkable assertions;
- Injecting the invariant assertions in the same version of the web application;
- Navigating the web application;
- Collecting assertion violations and removing them;

From each of the inferred invariants, we generate an assertion in JAVASCRIPT format. We use on-the-fly transformation to inject the assertions directly into the JAVASCRIPT code of the same version of the web application. Since we have all the information about the program points and the location of the invariants, we can inject the assertions at the correct position in the JAVASCRIPT source code through the proxy, as they are served by the server. This way the assertions gain access to the values of all program variables needed at runtime. Once the assertions are injected, we execute the web application in the browser and log the output. Next we collect and remove any violated assertions. The output of this step is a set of *stable* invariant assertions, which will be used for automated regression testing in the next step.

3.4 Regression Testing through Invariant Assertions

Once a set of stable invariant assertions are derived from version `n` of a web application, we use them for automatic regression testing a subsequent version (`n+1`). The regression testing phase is depicted in Figure 4.

We inject the inferred stable assertions to the JAVASCRIPT source code of the modified web application, in a similar fashion to the filtering step in Section 3.3. Once the assertions are injected, the new version of the web application is ready for regression testing. Any failed assertion during the testing phase generates an entry in the test report, which is presented to the tester at the end of the testing step. The generated test report provides precise information on which invariant assertion failed, as well as the file name, the line number, and the function name of the assertion.

Figure 5 shows the automatically injected invariant assertions for our running example of Figure 1. Note that we do not show all the assertions as they clutter the figure. Each `assert` call has the invariant as the first parameter and the corresponding debugging information in the second parameter, which includes information about script

```
1  function setDim(height, width) {
2   assert((width < height), 'example.js:setDim:ENTER:POINT1');
3   var h = 4*height, w = 2*width;
4   ...
5   assert((width < height), 'example.js:setDim:EXIT:POINT1');
6   assert((w < h), 'example.js:setDim:EXIT:POINT1');
7   return(h:h, w:w);
8  }

10 function play(){
11  $(#end).css("height", setDim($('body').width(), $('body').height()).h + 'px');
12  assert(isIn($('#end').css('height'), {100, 200, 300}),'example.js:play:POINT3');
13  ...
14 }
```

Fig. 5. Invariant assertion code for JAVASCRIPT function parameters, local variables and DOM modifications. Injected assertions are shown in bold.

name, function name, and line number. In this example, the inferred invariants yield information about the inequality relation between function arguments, `width` and `height`, as well as local variables, `w` and `h`. The assertions in lines 2 and 5-6 check the corresponding inequalities, at entry and exit points of the `setDim` function at runtime. The example also shows the assertion that checks the `height` attribute of the DOM element, after the JAVASCRIPT DOM modification in the `play` function. The assertion that comes after the DOM manipulation (Line 12) checks the `height` value by calling the auxiliary `isIn` function. `isIn` checks the value of `height` to be in the given range, i.e., either 100, 200, or 300. Any values out of the specified range would violate the assertion.

4 Tool Implementation

We have implemented our JAVASCRIPT regression testing approach in a tool called JSART. It is written in Java and is available for download.[1]

For JAVASCRIPT code interception, JSART uses an enhanced version of Web-Scarab's proxy [2]. This enables JSART to automatically analyze and modify the content of HTTP responses before they reach the browser. To instrument the intercepted code, Mozilla Rhino[2] is used to parse JAVASCRIPT code to an 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 needs to be added. For the invariant generation step, we have extended Daikon [7] with support for accepting input and generating output in JAVASCRIPT syntax. The input files are created from the trace data and fed through the enhanced version of Daikon to derive dynamic invariants. The navigation step is automated by making JSART operate as a plugin on top of our dynamic AJAX crawler, CRAWLJAX [13].[3]

[1] http://salt.ece.ubc.ca/content/jsart/

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

[3] http://www.crawljax.com

Table 1. Characteristics of the experimental objects.

App ID	Name	JS LOC	# Functions	# Local Vars	# Global Vars	CC	Resource
1	SameGame	206	9	32	5	37	http://crawljax.com/same-game
2	Tunnel	334	32	18	13	39	http://arcade.christianmontoya.com/tunnel
3	TicTacToe	239	11	22	23	83	http://www.dynamicdrive.com/dynamicindex12/tictactoe.htm
4	Symbol	204	20	28	16	32	http://10k.aneventapart.com/2/Uploads/652
5	ResizeMe	45	5	4	7	2	http://10k.aneventapart.com/2/Uploads/594
6	GhostBusters	277	27	75	4	52	http://10k.aneventapart.com/2/Uploads/657
7	Jason	107	8	4	8	6	http://jasonjulien.com
8	Sofa	102	22	2	1	5	http://www.madebysofa.com/archive
9	TuduList	2767	229	199	31	28	http://tudu.ess.ch/tudu

5 Empirical Evaluation

To quantitatively assess the accuracy and efficiency of our approach, we have conducted a case study following guidelines from Runeson and Höst [20]. In our evaluation, we address the following research questions:

RQ1 How successful is JSART in generating stable invariant assertions?
RQ2 How effective is our overall regression testing approach in terms of correctly detecting faults?
RQ3 What is the performance overhead of JSART?

The experimental data produced by JSART is available for download.[4]

5.1 Experimental Objects

Our study includes nine web-based systems in total. Six are game applications, namely, SameGame, Tunnel, TicTacToe, Symbol, ResizeMe, and GhostBusters. Two of the web applications are Jason and Sofa, which are a personal and a company homepage, respectively. We further include TuduList, which is a web-based task management application. All these applications are open source and use JAVASCRIPT on the client-side.

Table 1 presents each application's ID, name, and resource, as well as the characteristics of the custom JAVASCRIPT code, such as JAVASCRIPT lines of code (LOC), number of functions, number of local and global variables, as well as the cyclomatic complexity (CC). We compute the cyclomatic complexity across all JAVASCRIPT functions in the application.

5.2 Experimental Setup

To run the experiment, we provide the URL of each experimental object to JSART. In order to produce representative execution traces, we navigate each application several times with different crawling settings. Crawling settings differ in the number of visited

[4] http://salt.ece.ubc.ca/content/jsart/

states, depth of crawling, crawling time, and clickable element types. To obtain representative data traces, each of our experimental objects is navigated three times on average. Although JSART can easily instrument the source code of imported JAVASCRIPT libraries (e.g., jQuery, Prototype, etc), in our experiments we are merely interested in custom code written by developers, since we believe that is where most programming errors occur.

To evaluate our approach in terms of inferring stable invariant assertions (RQ1), we count the number of stable invariant assertions generated by JSART before and after performing the filtering step. As a last check, we execute the initial version of the application using the stable assertions to see whether our filtered invariant assertions are reporting any false positives.

Once the stable invariant assertions are obtained for each web application, we perform regression testing on modified versions of each application (RQ2). To that end, in order to mimic regression faults, we produce twenty different versions for each web application by injecting twenty faults into the original version, one at a time. We categorize our faults according to the following fault model:

1. **Modifying Conditional Statements**: This category is concerned with swapping consecutive conditional statements, changing the upper/lower bounds of loop statements, as well as modifying the condition itself;
2. **Modifying Global/Local Variables**: In this category, global/local variables are changed by modifying their values at any point of the program, as well as removing or changing their names;
3. **Changing Function Parameters/Arguments**: This category is concerned with changing function parameters or function call arguments by swapping, removing, and renaming parameters/arguments. Changing the sequence of consecutive function calls is also included in this category;
4. **DOM modifications**: Another type of fault, which is introduced in our fault model is modifying DOM properties at both JAVASCRIPT code level and HTML code level.

For each fault injection step, we randomly pick a JAVASCRIPT function in the application code and seed a fault according to our fault model. We seed five faults from each category.

To evaluate the effectiveness of JSART (RQ2), we measure the precision and recall as follows:

Precision is the rate of injected faults found by the tool that are correct: $\frac{TP}{TP+FP}$
Recall is the rate of correct injected faults that the tool finds: $\frac{TP}{TP+FN}$

where TP (true positives), FP (false positives), and FN (false negatives) respectively represent the number of faults that are correctly detected, falsely reported, and missed.

To evaluate the performance of JSART (RQ3), we measure the extra time needed to execute the application while assertion checks are in place.

5.3 Results

In this section, we discuss the results of the case study with regard to our three research questions.

Table 2. Properties of the invariant assertions generated by JSART.

App ID	Trace Data (MB)	# Total Assertions	# Entry Assertions	# Exit Assertions	# DOM Assertions	# Total Unstable Assertions	# Unstable Entry Assertions	# Unstable Exit Assertions	# Unstable DOM Assertions	# Total Stable Assertions	# Stable Entry Assertions	# Stable Exit Assertions	# Stable DOM Assertions
1	8.6	303	120	171	12	0	0	0	0	303	120	171	12
2	124	2147	1048	1085	14	14	9	5	0	2133	1039	1080	14
3	1.2	766	387	379	0	16	8	8	0	750	379	371	0
4	31.7	311	138	171	2	14	7	7	0	297	131	164	2
5	0.4	55	20	27	8	0	0	0	0	55	20	27	8
6	2.3	464	160	266	38	3	1	2	0	461	159	264	38
7	1.2	29	4	6	19	0	0	0	0	29	4	6	19
8	0.1	20	2	2	16	0	0	0	0	20	2	2	16
9	2.6	163	58	104	1	0	0	0	0	163	58	104	1

Generated Invariant Assertions. Table 2 presents the data generated by our tool. For each web application, the table shows the total size of collected execution traces (MB), the total number of generated JAVASCRIPT assertions, the number of assertions at entry point of the functions, the number of assertions at exit point of the functions, and the number of DOM assertions. The unstable assertions before the filtering as well as the stable assertions after the filtering step are also presented. As shown in the table, for applications 1, 5, 7, 8, and 9, all the generated invariant assertions are stable and the filtering step does not remove any assertions. For the remaining four applications (2, 3, 4, 6), less than 5% of the total invariant assertions are seen as unstable and removed in the filtering process. Thus, for all the experimental objects, the resulting stable assertions found by the tool is more than 95% of the total assertions. Moreover, we do not observe any unstable DOM assertions. In order to assure the stability of the resulting assertions, we examine the obtained assertions from the filtering step across multiple executions of the original application. The results show that all the resulting invariant assertions are truly stable since we do not observe any false positives.

As far as RQ1 is concerned, our findings indicate that (1) our tool is capable of automatically generating a high rate of JAVASCRIPT invariant assertions, (2) the unstable assertions are less than 5% of the total generated assertions, (3) the filtering technique is able to remove these low numbers of unstable assertions, and (4) all the remaining invariant assertions that JSART outputs are stable, i.e., they do not produce any false positives on the same version of the web application.

Effectiveness. In Table 3, we present the accuracy of JSART in terms of its fault finding capability. Note that since applications 3, 4, and 9, do not contain sufficient number of DOM manipulations, we were not able to inject them with 5 faults in the DOM modification category. Therefore, we randomly choose more number of faults from the remaining categories. The table shows the number of false negatives, false positives, true positives, as well as the percentages of precision and recall. As far as RQ2 is con-

Table 3. Precision and Recall for JSART fault detection.

App ID	# FN	# FP	# TP	Precision (%)	Recall (%)
1	2	0	18	100	90
2	4	0	16	100	80
3	1	0	19	100	95
4	2	0	18	100	90
5	0	0	20	100	100
6	1	0	19	100	95
7	0	0	20	100	100
8	0	0	20	100	100
9	1	0	19	100	95

cerned, our results show that JSART is very accurate in detecting faults. The precision is 100%, meaning that all the injected faults, which are reported by the tool, are correct. This also implies that our filtering mechanism successfully eliminates unstable assertions as we do not observe any false positives. The recall oscillates between 80-100%, which is caused by a low rate of missed faults (discussed in Section 6 under Limitations). Therefore, as far as RQ2 is concerned, JSART is able to successfully spot the injected faults with a high accuracy rate.

Performance. Figure 6 depicts the total running time needed for executing each web application with and without the assertion code. Checking a fairly large number of assertions at runtime can be time consuming. Thus, to capture the effect of the added assertions on the execution time, we exploit a 2-scale diagram. As shown in Figure 6, each experimental object is associated with two types of data. The left-hand Y-axis represents the running time (seconds), whereas the right-hand Y-axis shows the number of assertions. This way we can observe how the number of assertions relates to the running time. As expected, the figure shows that by increasing the number of assertions, the running time increases to some degree. While the time overhead of around 20 seconds is more evident for the experimental object 2 (i.e., Tunnel with 2147 number of assertions), it is negligible for the rest of experimental objects. Considering that Tunnel has 260 statements in total, the number of assertions instrumented in the code is eight times more than the number of statements in the original version. Therefore, it is reasonable to observe a small amount of overhead. Though assertions introduce some amount of overhead, it is worth mentioning that we have not experienced a noticeable change (i.e, freezing or slowed down execution) while running the application in the browser.

Thus, as far as RQ3 is concerned, the amount of overhead introduced by our approach is 6 seconds on average for our experimental objects, which is negligible during testing. Furthermore, based on our observations, the assertions do not negatively affect the observable behaviour of the web applications in the browser.

6 Discussion

Unstable Assertions. As mentioned in Section 3.3, we observe a few number of unstable invariant assertions initially, which are removed by our filtering mechanism. By analyzing our trace data, we observe that such unstable assertions arise mainly because

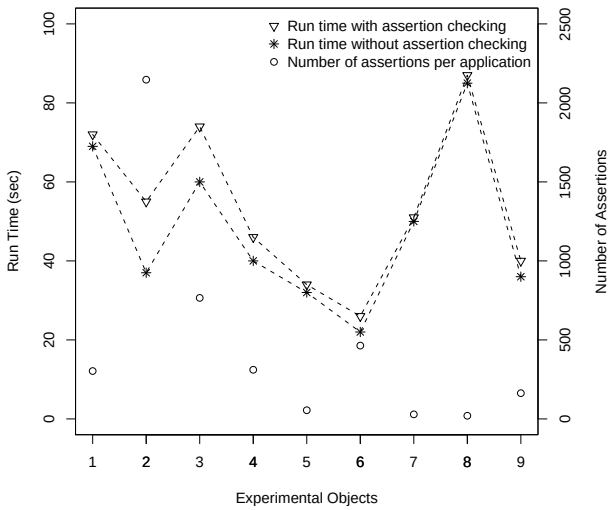

Fig. 6. Performance plot of JSART.

of the following two reasons (1) the precision of comparing two floating point numbers in Daikon; (2) multiple runtime types of JAVASCRIPT variables. The first is concerned with the number of digits in the fractional part of the floating point numbers, which is used by Daikon while comparing two floats. We resolve this by increasing the precision of float comparisons in Daikon configurations. The second reason arises from the fact that in JAVASCRIPT it is possible to change the type of a variable at runtime. However, Daikon treats variables as single type, selects the first observed type, and ignores the subsequent types in the trace data. This results in producing a few number of unstable invariant assertions for JAVASCRIPT. We remove such unstable assertions in our filtering step. A drawback of removing these assertions, is that our tool might miss a fault during the regression testing phase. However, according to our observations, such unstable assertions form only around 5% of the total generated assertions. Thus, we are still able to achieve high accuracy as presented in the previous section.

Limitations. Our approach is not able to detect syntax errors that are present in the JAVASCRIPT code. Furthermore, tracing DOM manipulations using APIs other than the standard DOM API or jQuery is currently not supported by JSART. Further, a regression fault either directly violates an invariant assertion, or it can violate closely related assertions, which have been affected by the fault. However, if the tool is not able to infer any invariants in the affected scope of the error, it fails to detect the fault. This results in observing a low rate of false negatives as illustrated in Section 5. In addition, we assume that the invariants are collected from a bug-free version of the web application. However, if the application does contain faults, the generated assertions might reflect the fault as well.

Table 4. Manual effort imposed by our approach for deriving stable invariant assertions.

App ID	Total Time (min)	Manual Effort (min)
1	13	4
2	11.5	3
3	15.5	5
4	11	3
5	6.5	2.5
6	9	4.5
7	7.5	3.5
8	6.5	2
9	18	13

Automation Level. While the testing phase of JSART is fully automated, the navigation part requires some manual effort. Although the crawling is performed automatically, we do need to manually setup the tool with different crawling configurations per application execution. Moreover, for each application run, we manually look at the size of the invariant output to decide whether more execution traces (and thus more crawling sessions) are needed. We present the manual effort involved with detecting stable invariant assertions in Table 4. The table shows the total time, which is the duration time of deriving stable assertions including both automatic and manual parts. The reported manual effort contains the amount of time required for setting up the tool as well as the manual tasks involved with the navigation part. Note that since application 9 requires authentication, we wrote (10 minutes) a plugin to log into the web application automatically. The results show the average manual effort is less than 5 minutes.

7 Related Work

We classify related work into two broad categories: JAVASCRIPT analysis and program invariants.

JavaScript Analysis Automated testing of modern web applications is becoming an active area of research [1, 12, 14, 16]. Most of the existing work on JAVASCRIPT analysis is, however, focused on spotting errors and security vulnerabilities through static analysis [8, 9, 23]. Kudzu [21] is a symbolic execution system for JAVASCRIPT aimed at automated security vulnerability analysis. BrowserShield [17] applies dynamic instrumentation to rewrite JAVASCRIPT code to conduct vulnerability driven filtering. Yu *et al.* [22] propose a method in which untrusted JAVASCRIPT code is analyzed and instrumented [11] to identify and modify questionable behaviour.

Program Invariants. The concept of using invariants to assert program behaviour at runtime is as old as programming itself [5]. A more recent development is the automatic detection of program invariants through dynamic analysis. Ernst *et al.* have developed Daikon [7], a tool capable of inferring likely invariants from program execution traces. Other related tools for detecting invariants include Agitator [4], DIDUCE [10], and DySy [6]. Rodríguez-Carbonell and Kapur [18] use inferred invariant assertions for program verification.

In our previous work [14], we proposed ATUSA, a framework to manually specify generic and application-specific invariants on the DOM-tree and JAVASCRIPT

code. These invariants were subsequently used as test oracles to detect erroneous behaviours in modern web applications through an automated crawling technique [13].

To the best of our knowledge, our work in this paper is the first to generate JAVASCRIPT assertions dynamically and use them at runtime for automated regression testing.

8 Conclusions and Future Work

JAVASCRIPT is playing a prominent role in modern Web 2.0 applications. Due to its loosely typed and dynamic nature, the language is known to be error-prone and difficult to test. In this paper, we present an automated technique for JAVASCRIPT regression testing based on generated invariant assertions. The contributions of this work can be summarized as follows:

- A method for detecting JAVASCRIPT invariants across multiple application executions through on-the-fly JAVASCRIPT instrumentation and tracing of program variables and DOM manipulations;
- A technique for automatically converting the inferred invariants into stable assertions, and injecting them back into the web application for regression testing;
- The implementation of our proposed technique in an open source tool called JSART;
- An empirical study on nine open source JAVASCRIPT applications. The results of our study show that our tool is able to effectively infer stable assertions and detect regression faults with minimal performance overhead;

Our future work encompasses conducting more case studies to generalize the findings as well as extending the current JAVASCRIPT DOM modifications detector so that it is capable of coping with more patterns in other JAVASCRIPT libraries. In addition, we will explore ways of fully automating the navigation part by generating crawling specifications through search-based techniques.

References

1. S. Artzi, J. Dolby, S. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of the Intl. Conference on Software Engineering (ICSE)*, pages 571–580. ACM, 2011.
2. C.-P. Bezemer, A. Mesbah, and A. van Deursen. Automated security testing of web widget interactions. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC-FSE'09)*, pages 81–91. ACM, 2009.
3. R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000.
4. M. Boshernitsan, R. Doong, and A. Savoia. From Daikon to Agitator: lessons and challenges in building a commercial tool for developer testing. In *Proc. Int. Sym. on Software Testing and Analysis (ISSTA'06)*, pages 169–180. ACM, 2006.
5. 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.

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. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
8. S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for JavaScript code. In *Conference on USENIX security symposium*, SSYM'09, pages 151–168, 2009.
9. A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Intl. conference on World Wide Web (WWW)*, pages 561–570, 2009.
10. 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.
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. A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *Proc. 1st Int. Conference on Sw. Testing Verification and Validation (ICST'08)*, pages 121–130. IEEE Computer Society, 2008.
13. A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proc. of the 8th Int. Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.
14. A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering (TSE)*, 38(1):35 –53, 2012.
15. F. J. Ocariza, K. Pattabiraman, and A. Mesbah. AutoFLox: An automatic fault localizer for client-side JavaScript. In *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation (ICST'12)*. IEEE Computer Society, 2012.
16. K. Pattabiraman and B. Zorn. DoDOM: Leveraging DOM invariants for Web 2.0 application robustness testing. In *Proc. Int. Conf. Sw. Reliability Engineering (ISSRE'10)*, pages 191–200. IEEE Computer Society, 2010.
17. 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, 2007.
18. E. Rodríguez-Carbonell and D. Kapur. Program verification using automatic generation of invariants. In *Proc. 1st International Colloquium on Theoretical Aspects of Computing (ICTAC'04)*, volume 3407 of *Lecture Notes in Computer Science*, pages 325–340. Springer-Verlag, 2005.
19. D. Roest, A. Mesbah, and A. van Deursen. Regression testing Ajax applications: Coping with dynamism. In *Proc. 3rd Int. Conf. on Sw. Testing, Verification and Validation (ICST'10)*, pages 128–136. IEEE Computer Society, 2010.
20. P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.
21. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proc. Symp. on Security and Privacy (SP'10)*, pages 513–528. IEEE Computer Society, 2010.
22. 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.
23. Y. Zheng, T. Bao, and X. Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the Intl. Conference on the World-Wide Web (WWW)*, pages 805–814. ACM, 2011.