DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing

Karthik Pattabiraman
University of British Columbia
karthikp@ece.ubc.ca

Benjamin Zorn
Microsoft Research (Redmond)
ben.zorn@microsoft.com

***Abstract*—Web 2.0 applications are increasing in popularity. However, they are also prone to errors because of their dynamic nature. This paper presents *DoDOM*, an automated system for testing the robustness of Web 2.0 applications based on their Document Object Models (DOMs). DoDOM repeatedly executes the application under a trace of recorded user actions and observes the client-side behavior of the application in terms of its DOM structure. Based on the observations, DoDOM extracts a set of invariants on the web application's DOM structure. We show that invariants exist for real applications and can be learned within a reasonable number of executions. We further use fault-injection experiments to demonstrate the uses of the invariants in detecting errors in web applications. The invariants are found to provide high coverage in detecting errors that impact the DOM, with a low rate of false positives.**

***Keywords*-Web 2.0, Dynamic Invariants, Robustness Testing, Error Detection**

I. Introduction

The web has evolved from a static medium that is viewed by a passive client to one in which the client is actively involved in the creation and dissemination of content. The evolution is enabled by the use of dynamic technologies such as JavaScript, Silverlight, and Flash, which allow the execution of client-side scripts in the web browser to provide a rich, interactive experience for the user. Applications deploying these technologies are called *Rich Internet Applications* or *Web 2.0* applications. Web 2.0 applications, such as Gmail and Facebook, require little installation or maintenance and have the ability to run on a wide variety of platforms [1]. Consequently, they are increasing in popularity and are being rapidly adopted.

Unfortunately, Web 2.0 applications are also complex and prone to errors. First, the distributed nature of the application's logic between the server and client makes it difficult to understand and debug such applications. This problem is compounded by the asynchronous behavior of the application. Second, web applications often integrate data and code from multiple domains, and the failure of any of these domains can make the application unstable. Finally, unlike desktop applications, web applications are rarely designed with a fail-fast philosophy. Rather, they tend to continue executing even if some component of the application experiences an error (e.g., an event-handler throws an exception). This behavior has its roots in the historical evolution of the web and is advantageous from the point of view of compatibility and co-existence. However, it makes it difficult to contain and localize faults and is hence disadvantageous from the point of view of reliability. Therefore, it is important to test the robustness of web applications using fault-injection experiments.

An important challenge in testing Web 2.0 applications' robustness is lack of determinism from one execution to another (even with the same input sequence). This is due to a number of factors, including (1) small changes introduced by the server, (2) asynchrony in network messages being sent or received out of order, and (3) small variations in the timing of events at the client. The non-determinism makes it difficult to ascertain if an observed change in the execution of a web application was because of an injected fault. Non-determinism has been addressed in the context of Web 1.0 applications by controlling server-side execution [2]. However, these solutions address only the first source of non-determinism in Web 2.0 applications.

In this paper, we propose a first step towards characterizing the client-side behavior of web applications with the goal of testing their robustness using fault injection. Our approach characterizes the structure of the Document Object Model (DOM) data-structure in the application using dynamically derived invariants, and considers deviations from the derived structure as erroneous executions. The invariants are derived dynamically by recording a sequence of user interactions with the application and observing the changes to the DOM by repeatedly replaying the sequence. Because we repeatedly execute the web application and derive invariants over its DOM, we call this approach DoDOM (the DO symbolizes iteration as in a do-while loop).

The DOM is the central data structure maintained by the web browser for representing and displaying a web application's output. It is organized hierarchically with each node in the DOM representing an entity of the web page corresponding to the application (e.g., list elements, text). Only objects in the DOM can be displayed by the web browser. Hence, the DOM is what users ultimately see and interact with in a web application. Further, multiple client-side scripts in the application share state and communicate with one another predominantly through the DOM. *Thus, the correctness of the DOM is essential for the correctness of the application and hence we focus on DOM-based invariants in this paper.*

DoDOM extracts invariants over the DOM structure of a web application by capturing a user-interaction sequence with the application, replaying it multiple times and observing the DOM after each execution. This dynamic approach of learning invariants is language-neutral and does not need to statically analyze the code of the web application. This is important as (1) languages and frameworks for writing web applications are rapidly evolving and a single application may combine code from multiple languages and frameworks, and (2) client-side languages such as JavaScript are notoriously difficult to analyze statically due to the presence of dynamic constructs [3].

Prior work has shown that dynamic invariants can be used in general-purpose programs for testing and error detection [4], [5], [6]. However, there has been relatively little work on deriving dynamic invariants for web applications. We show that DOM-based invariants (1) exist in web applications, (2) can be learned automatically, and (3) are useful in detecting errors. *To the best of our knowledge, DoDOM is the first technique to automatically extract invariants in Web 2.0 applications for the purpose of error detection*[1].

The main contributions of the paper are as follows:

- We show that Web 2.0 applications exhibit invariants over their DOM structures, and build a tool called DoDOM to replay web applications and extract their invariants.
- We demonstrate the invariant extraction capabilities of DoDOM for three real Web 2.0 applications: Slashdot, CNN, and Java Petstore. We show that the invariants can be learned within 6 executions of each application.
- Using fault-injection experiments, we find that the error-detection coverage of the invariants is close to 100% for errors that impact the web page's DOM. The remaining errors do not have any effect on the DOM and are hence not detected by the technique[2].

II. Overview

In this section, we outline the reliability issues with Web 2.0 applications and present our proposed solution. We first present a brief background on Web 2.0, which may be skipped by the reader familiar with this paradigm.

A. Background

Typical Web 2.0 applications consist of both server-side and client-side code. The server-side code is written in traditional web-development languages such as PHP, Perl, Java and C. The client-side code is written using dynamic web languages such as JavaScript (JS) which are executed within the client's browser. Unlike in the Web 1.0 model, where the application executes primarily at the server with the browser acting as a front-end for rendering and displaying the server's responses, in a Web 2.0 application the client is actively involved in the application's logic. This reduces the amount of data exchanged with the server and makes the application more interactive. *Henceforth, when we say web applications, we mean Web 2.0 applications unless we specify otherwise.*

[1]ATUSA [7] proposes the use of invariants for error detection in Web 2.0 applications but requires the programmer to define the invariants.

[2]Such errors may be detected through checks in the application's backend, but are outside this paper's scope.

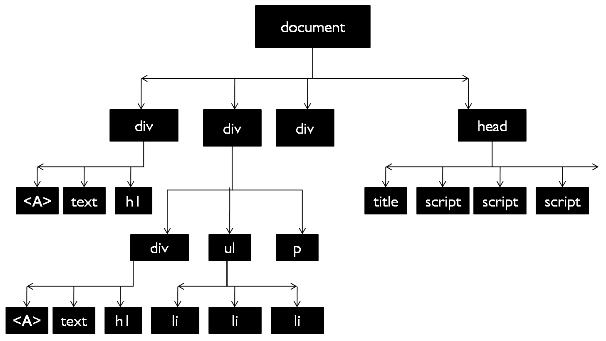

Fig. 1. Example of a DOM tree for a web application.

Typical web applications are event-driven, i.e., they respond to user events (e.g., mouse clicks), timeouts and receipt of messages from the server. The developer writes handlers for each of these events. The event-handlers can (1) invoke other functions or write to global variables stored on the heap, (2) read/modify the web page's contents through its DOM, or (3) send asynchronous messages to the server through a technology known as AJAX (Asynchronous JavaScript and XML) and specify the code to be executed upon receiving a response. The above actions are executed by client-side scripts in the web browser.

Web applications typically execute within a single web page. A web page is represented internally by the browser as its DOM [8], which as mentioned before, consists of the page's elements organized in a hierarchical format. Figure 1 shows an example of a DOM for a web page. In the figure, the web page consists of multiple HTML div elements, which are represented in the top-level nodes of the tree. The "div" elements are logical partitions of the page, each of which consists of nodes representing text and link elements. Further, the web page has a head node with two client-side scripts as its child nodes. JavaScript code executing in the web browser can read and write to the DOM through special APIs. Any changes made to the DOM cause the web page to be re-rendered by the web browser. User actions are converted into events by the browser and sent to the nodes of the DOM on which they are trigerred. If the node has an installed event handler for the event, then the event-handler is executed.

B. Scenario

Consider an application developer who wants to test the robustness of a web application to faults. She would interact with the application on the client, inject a fault into it and check if the application behaves as expected. However, this approach is time consuming and requires the user to repeat the same interactions with the application for each injected fault. Further, the user needs to rely on visual perception to determine whether an injected fault affects the application[3]. Finally, web applications exhibit variations in their behavior from one execution to another (as we show later in this paper).

[3]While it can be argued that faults that are not perceived by the user do not matter, it may be that a different user perceives the fault.

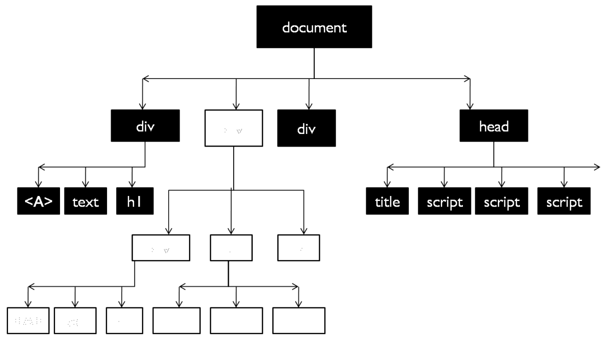

Fig. 2. Proposed solution in the context of the example DOM.

In the face of such variations, it becomes challenging to identify whether a perceived difference is the result of a fault or if it is due to the natural behavior of the application. Further, the variations can occur in the middle of the web application's execution which makes them difficult to detect.

This paper proposes a systematic method to characterize the correct behavior of a web application for robustness testing. We assume that the web developer has one or more user interaction sequences (i.e., sequences of user actions) under which she wants to test the application's robustness. Our solution involves extracting an invariant characteristic of the web application's DOM from multiple executions of the application. We characterize the expected behavior of the application based on the invariants. We then inject faults into the application and consider significant deviations from the invariant behavior of the application as erroneous executions.

C. Dynamic Invariant Extraction

This section illustrates our proposed solution for the problem illustrated in Section II-B. The crux of the solution is in characterizing the invariant portions of the DOM for the web application under a given sequence of user interactions. Specifically, we characterize the common portions of the DOM of the application's web page over multiple executions, and the changes made to the DOM by the application in response to various events (i.e., user actions, timeouts, and network messages). After each event executes, we check the conformance of the resulting DOM to the invariant portions. A deviation indicates an error. We first illustrate using the example in Figure 1 and then present the general case.

Figure 2 shows the invariant portion of the DOM for the example considered in Figure 1. In the figure, the darkly-shaded nodes represent the invariant portions of the tree (also called the web page's backbone) while the lightly-shaded nodes represent the non-invariant portions. We consider two example faults to illustrate the error-detection process. First, consider the case where the user clicks on a specific DOM node which in turn triggers an event handler on the node. The event handler is supposed to update the left most element (A) in the invariant DOM but fails to do so because of an error (e.g., the handler throws an exception). This error will be detected as the resulting DOM would deviate from the invariant DOM for the event (in this case, the event is the mouse click).

Executions	Event 1	Event 2	...	Event n
Execution 1	T_{1_1}	T_{2_1}	...	T_{n_1}
Execution 2	T_{1_2}	T_{2_2}	...	T_{n_2}
...	...	...	...	...
Execution M	T_{1_M}	T_{2_M}	...	T_{n_M}
Invariants	T_{1_I}	T_{2_I}	...	T_{n_I}

TABLE I
INVARIANT DOMS.

Second, assume that the web application is supposed to import a script from a domain but fails to do so because of the domain being unavailable. Assume that the script is supposed to modify the 'div' element in the far left branch of the tree. This element is part of the invariant DOM and hence the lack of modification will be detected as an error. *Our hypothesis is that the majority of the DOM structure is invariant in a web application and hence the invariant DOM can be used to detect the majority of errors in a web application.*

In the above example, the invariant DOM in Figure 2 represents a snapshot of the DOM during the course of its evolution in response to events. In reality, the technique will capture the entire sequence of invariant DOMs and use the invariant sequence to check for deviations as we show below.

An invariant DOM is a subset of the web application's DOM that is shared by multiple executions of the application. We derive an invariant DOM for each event in the application (an event refers to a user action, network message or timeout). Table I shows the DOMs obtained during multiple executions of the web application under a given sequence of events. The rows of the table indicate different executions of the web application, while the columns indicate the events in the application. The DOMs are indicated by T_{i_j}, where i is the event after which the DOM is obtained and j is the corresponding execution. The invariant DOMs T_{i_I} are derived from the DOMs of individual executions T_{i_j} corresponding to the event i. As can be seen from the table, the invariant DOMs generalize across multiple executions to incorporate only the common features of each DOM. However, they are specific to a given sequence of events in order to ensure high error-detection coverage. We consider the implication of this trade-off in Section VI.

D. Fault Model

This section discusses the errors injected in this study to evaluate the derived invariants. Note that we inject errors (i.e., manifestations of faults) rather than the underlying faults. However, in keeping with convention, we refer to these experiments as *fault injections*.

Event errors: These correspond to errors encountered when processing events in the client-side code of the application. These can be caused by exceptions in the corresponding event handlers or the events not being triggered correctly. In some cases, the web browser may abort an event handler if it executes for too long.

Domain failures: These can be caused by a network failure or by the unavailability of the domains' servers. They can also be caused by client-side plugins or administrative proxies which may block scripts from certain domains.

While the above errors may seem somewhat simplistic, it is a first step towards characterizing faults in Web 2.0 applications. Prior work has characterized fault-models for Web 1.0 applications [9], which comprise only server-side code. However, to the best of our knowledge, there has been no similar effort to characterize common failure modes of Web 2.0 applications which involve significant amounts of client-side processing. We also note that our method to extract invariant DOMs is independent of the fault model, which is only used to evaluate the invariants. Future work will attempt to extend the scope of the injections and consider more realistic faults.

III. Approach

The overall approach for extracting and learning invariants over the DOMs of web applications consists of the following steps: First, we record a sequence of user interactions and events on a page (trace). We then replay this trace over multiple executions and capture the sequence of DOMs generated after each event in the trace. Finally, we extract invariants over the set of all DOM sequences using an offline learning process. We built a tool, DoDOM, to automate the process of recording a user-interaction sequence with a web application, replaying it, and extracting invariants from the observed DOM sequences.

In this section, we first describe the challenges encountered in the above process and then discuss how DoDOM address these challenges. We also discuss the design choices and the trade-offs made in DoDOM.

A. Challenges

We identify three main challenges in extracting invariants using DoDOM. First, we need to record the sequence of user-interactions with the web application in an unobtrusive manner because we want to obtain realistic user-interaction sequences. Second, we need to replay the user-interaction events on the web application on the same DOM nodes on which they occurred when recording the sequence. In particular, the web page rendered by the application may undergo minor changes from one execution to another (because of server-side changes[4]) and hence the replay should be robust to such changes. Finally, it is desirable that the method be independent of the web browser and that it not require any changes to the same.

DoDOM addresses the above challenges as follows. First, during recording, it passively records the interaction of a user with the web application without requiring the user to perform any additional steps. Second, in order to find nodes during replay, it uses both the contents of a node in the DOM and its relative position to other nodes. Further, the comparison is not exact, but is based on heuristic measures, thereby ensuring that events are played back on the original nodes on which they occurred even if the web page has undergone small changes during replay. Finally, the DoDOM tool is predominantly implemented using JavaScript (with a small portion implemented as a proxy server), and hence does not require any modifications to the web browser.

[4]For ease of deployment, we do not require the execution of the server code to be controlled. This is especially important when testing real web sites.

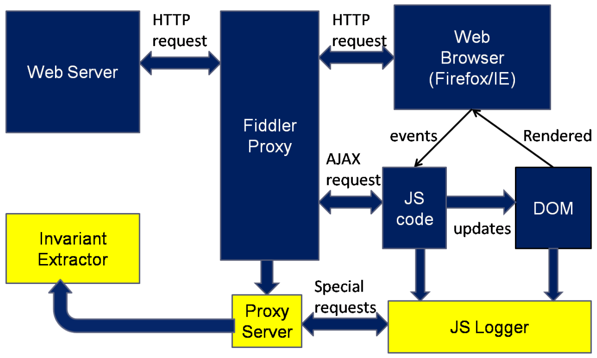

Fig. 3. Architecture of DoDOM system: The components we added are lightly shaded.

B. DoDOM Operation

Figure 3 shows the architecture of *DoDOM*. DoDOM consists of three components as follows.

(1) The **proxy** is a client-side proxy server written as a plugin in the Fiddler web application testing framework [10]. The proxy's main purpose is to inject the JS logger code into the web page(s) of the application, collect the events and responses sent by the JS logger and record them.

(2) The **JS logger** is a piece of JavaScript code that is executed on the client's browser for each iFrame in the page. The JS logger can read/write to the web page's DOM, install event handlers that trap the page's handlers and log changes to the DOM. It can also intercept messages sent by the client through the XMLHttp interface and the corresponding response (i.e., AJAX messages).

(3) The **invariant extractor** performs offline analysis of multiple executions recorded by the proxy and extracts the invariants. It runs outside the web browser.

The proxy injects the JS logger script into every page loaded by the browser (a page is defined as any entity that consists of a head tag). The proxy also assigns to each JS logger script a unique tag called the pageID. The pageID is used to distinguish individual iframes in a multi-frame web application. The JS logger script is instantiated at the client after the page completes loading (after the *onLoad* event), upon which it performs the following actions (in sequence): (1) Creates a compact representation of the web page's DOM, and sends it back to the proxy. (2) Installs a new replacement handler for all DOM elements that have event handlers and stores the old handler as part of the element. (3) Replaces the *setTimeout* and *setInterval* API calls in the window object with custom versions (after storing the old handler) to intercept timeouts. (4) Replaces the *XMLHttpRequest* object with a custom version that intercepts all messages sent to the server using the AJAX interface and their corresponding responses. (5) Installs change handlers on each element of the DOM tree to track any additions, modifications, and removals of the subtree rooted at the element. The above operations are performed using the DOM API calls in JavaScript.

The JS logger operates in two modes: record and replay. During *record mode*, the user interacts normally with the page by moving the mouse, clicking on objects, etc. The browser translates the user's actions into user-events and invokes the replacement event handlers installed by the JS logger on the corresponding DOM nodes. The handlers create a snapshot of each event and send it to the proxy, which in turn adds the events to a global queue for the page. The JS logger periodically polls the proxy for outstanding events, upon which the enqueued events are sent to the client (in order). The JS logger then invokes event's original handers on the node on which they occured.

During *replay*, the proxy reads in the list of events from the event log and populates the queue with the events. It injects the JS logger into the web page as before. However, when the JS logger polls for events, the proxy retrieves the events from the queue and sends them to the client one at a time along with the corresponding node on which the event occurred[5]. The JS logger attempts to identify the node using the node's contents sent by the proxy and its relative position in the DOM tree (i.e., pre-order traversal index). If an exact match is not found, it searches the DOM for the closest match starting from the node with the same pre-order index as the original node[6].

The proxy records the changes made to the web page's DOM tree after every event. The invariant extractor post-processes these traces to obtain a sequence of invariant DOM trees for each event. Each tree in the invariant sequence is learned independently based on the corresponding trees in the individual executions.

The algorithm for learning each of the invariant DOM trees from a set of execution traces is as follows (the pseudo-code for the complete algorithm can be found in [11]). We set the initial invariant tree to the tree obtained from the first execution trace. For each event in the execution trace, we compare its tree with the corresponding invariant tree recursively starting from the root nodes and traversing the tree in post order. When any of the following three conditions are met, the node is removed and a dummy node substituted in its place to hold the tree together. (1) the contents of a node of the invariant tree do not match the corresponding nodes in the execution tree, (2) a node in the invariant tree has more children than its corresponding node in the execution tree, (3) the invariant tree has nodes that are not present in the execution tree. The comparison among nodes' contents is based on the fraction of its field-value pairs that match each other. If this fraction is higher than a value known as the *match threshold*, then the nodes are considered to match each other. The comparison does not take into account the order of the field-value pairs in each node.

[5]It also introduces a time delay corresponding to the occurrence of the event in the recording mode.

[6]The closest match is the node(s) with the highest fraction of matching field-value pairs.

The match threshold thus determines how much of the invariant tree is pruned away because of differences among the DOMs of individual executions. A high match threshold means that only nodes that match closely across executions will be retained in the invariant tree. On the other hand, a low match threshold indicates that that the invariant tree may contain nodes that exhibit high variation in their contents among executions. Therefore an invariant tree with a high match threshold has high content similarity with individual executions.

C. Trade-offs and Limitations

For portability and ease of deployment, DoDOM is implemented predominantly in JS with a small part implemented as a client-side proxy. However, the use of JS incurs certain limitations. First, the JS logger is executed only after the *onLoad* event in a web page. Therefore, it cannot trap events that occur before the firing of the onLoad event and would hence ignore them. Second, DoDOM traps events by hooking into the event handlers of DOM nodes and replacing the existing functions with a custom wrapper. This behavior requires the web application to use the DOM 1.0 event model because the DOM 2.0 event model does not provide any way to remove a handler from the chain of event-listeners on a DOM node, or to ensure that the event handlers are invoked in a specific order. However, most applications are written using the DOM 1.0 model for compatibility reasons[7]. Finally, DoDOM's reliability is limited by the reliability of the web browser's JavaScript Virtual Machine (JS VM). In our experiments, we did not find cases where the JavaScript VM crashed or hung. Nonetheless, we provide a heartbeat service to detect crashes or hangs of the JS VM and reload the page.

Currently, DoDOM only supports single web page applications, i.e., web applications where the user interacts with the application on a single web page. However, even single web page applications often consist of multiple iframes each of which contains its own DOM. The user may interact with multiple iframes, and therefore it is necessary to record the interactions on a per-frame basis. This is done by assigning a unique tag to each frame (pageID) and using the tag to disambiguate interactions for different iframes. A similar mechanism can be applied for multi-page web applications which consist of multiple pages displayed in sequence.

Finally, DoDOM assumes that the web application uses standard mechanisms such as AJAX for communication between the client and the server. However, some frameworks such as Dojo use their own custom mechanisms for handling communications with the server (e.g., hidden iframes). DoDOM cannot currently support such mechanisms.

[7]The DOM 3.0 specification allows for removing event handlers and controlling their order, but unfortunately, is not implemented by most browsers.

IV. Experimental Setup

This section describes the experiments performed and the benchmarks used to evaluate DoDOM. We used an Intel Core2 Duo Intel dual-core processor (running at 3 GigaHertz) with 4 GigaBytes RAM. We used Firefox version 3.5 on Windows Vista as the platform for evaluation.

The main research questions are:

Q1. How many executions do we need to learn the invariant DOMs for a web application?

Q2. How many event errors impact the web application's DOM and how effective are the invariants at detecting these errors?

Q3. How many domain failures impact the web application's DOM and how effective are the invariants at detecting these errors?

Invariant Extraction: The goal of this experiment is to answer Q1. We first record the sequence of user interactions and create a log of events. We then replay the user events multiple times using DoDOM. From the set of all executions (i.e., replay sequences), we randomly choose a subset of executions used to learn the invariants, called the *training set*. In these experiments, we vary the size of the training set in order to understand how quickly the invariants converge to a stable value. We also vary the match threshold described in Section III to understand how much variation is present among individual executions.

We measure the following characteristics of the DOM tree in order to measure its convergence. (1) number of nodes in the DOM, (2) average number of children per node, i.e., its fanout, (3) maximum number of levels from each node, i.e., the height of the sub tree, and (4) average number of total descendants per node.

We also compare the invariant DOM sequence learned from the training set with the DOM sequences from all executions. If any of the DOMs in its sequence exhibits a mismatch with the corresponding DOM in the invariant sequence, we consider the execution a false positive.

Event Errors: The goal of this experiment is to answer Q2. We measure the error-detection coverage of the invariant sequences for event errors corresponding to those in Section II-D. Table II shows the types of faults introduced and the injection method. Each run injects at most one fault to ensure that the fault's effects can be uniquely determined. After a fault is injected, the sequence of DOMs corresponding to the execution is compared with the sequence of invariant DOMs. We classify the execution as a successful detection if any of the DOM trees in the sequence exhibits a mismatch with the corresponding DOM in the invariant sequence. The coverage obtained by DoDOM for a fault is calculated as the percentage of successful detections among the total number of executions corresponding to the fault.

Domain Failures: The goal of this experiment is to answer Q3. It emulates the effect of domain failures as described in Section II-D using the NoScript plugin in Firefox[8]. First, the invariant DOM sequence is obtained from multiple executions. Then, each domain in the web page is blocked one at a time and the corresponding DOM sequences are obtained. The DOM sequence for a blocked domain is compared to the invariant DOM sequence, and a mismatch indicates that the domain's failure is detected by the invariants.

[8]Available at *http://noscript.net/.*

Fault Type	**Injection Method**
User-Event Error	Do not replay the event at the client
Message Error	Do not forward the message to the server
Timeout Error	Do not replay the timeout at the client

TABLE II
Faults injected and their characterization.

Website	**Lines of JS code**	**No. of domains**	**No. of events**	**No. of DOM nodes**
Java Petstore	499	1	211	398
Slashdot	9647	5	13	5162
CNN	15603	9	9	2417

TABLE III
Characteristics of the web applications.

Benchmarks: We demonstrate DoDOM on three representative Web 2.0 applications: Slashdot, CNN and JavaPetStore. Slashdot aggregates technology-related news from different web sites and allows users to comment on a news story. Java Petstore is a freely available Web 2.0 application that mimics an e-commerce web site for buying pets [12]. CNN is a popular news web site that delivers customized content to its readers.

Table III summarizes the characteristics of the web applications which include the number of domains in the application, the total number of lines of JS code (obtained with Firefox's Phoenix plugin[9]), and the number of events in the recorded trace for the application.

We choose the Slashdot application as the primary source of measurements as it represents a middle ground among the applications in terms of lines of code and number of domains. We interact normally with a Slashdot news story and replay the interactions with DoDOM. The results reported are for a specific news story on Slashdot with close to 300 comments. We obtain a total of 13 events for the story including user interactions and timeout, and perform a total of 58 replays. We also repeated the measurements with a different sequence of interactions, but the results were similar and are not reported.

For the other two applications, we measured only the invariant extraction capabilities of DoDOM, using 50 replays each. Java Petstore allows the user to browse through pet listings and choose a pet corresponding to the user's preferences. We interact with the first page of the application by moving over and clicking on different elements of the page. For CNN, we interact with the main page of the application, which displays the daily news, by moving the mouse over various elements of the page and clicking on them as a normal user would.

[9]Available at *https://addons.mozilla.org/en-US/firefox/addon/11708/.*

V. Results

In this section, we present the results corresponding to the research questions in Section IV. We first summarize the main results and then present the details. The results presented pertain to the Slashdot application. We present the results for the other applications at the end of this section.

R1: corresponds to Q1: We show that the invariant DOM sequences converge with a training set size of 6 executions, which corresponds to 10% of the total executions. We further show that the invariant DOM converges to 99% of the original DOM size, suggesting that most of the DOM is invariant.

R2: corresponds to Q2: The invariants detect 100% of the injected event errors that affect the DOM.

R3: corresponds to Q3: Only one the 5 domains included by Slashdot has an effect on the DOM. DoDOM provides 100% coverage for failures of this domain.

Invariant Extraction: In this experiment, we vary the training set size[10] from 1 to 10 over 58 executions and obtain the invariant DOM sequence. The characteristics of the invariant DOM corresponding to the metrics listed in Section IV are shown in Figure 4 (a) to (d). In each graph in the figure, the X-axis represents the event number and the Y-axis represents a metric corresponding to the event. The lines in each graph represent the invariants obtained with a training set of a specific size. Note that the Y-axis in each graph does not start from 0, and there is very little variation among the different training set sizes.

We observe that the number of nodes monotonically increases with the event number, while the maximum number of levels in the DOM monotonically decreases with the event number. The other two metrics, namely the number of children and the number of descendants, show no consistent trend.

Figure 4 (a) shows that as the training set size increases, the number of nodes in the DOM decreases because more and more nodes are eliminated from the invariant DOMs. However, the number stops decreasing once the training set reaches a size of 6 (roughly 10% of the 58 executions). Nonetheless, the converged values are within 1% of their original values (i.e., the number of DOM nodes for any given execution). This shows that the amount of variation among executions, while non-trivial, is within 1% of each other. Similarly, the other three metrics, namely the maximum number of levels, the average number of children, and the average number of descendants, steadily increase with increase in the training set size, but also stabilize at a training set size of 6. This shows that the invariant DOMs can be learned using only 10% of the executions.

False Positives: To further confirm the convergence of the invariants, we measure the false-positive rate for the executions[11]. Table IV shows the false-positive rate as a function of the size of the training set for different values of the match threshold (introduced in Section III). As can be seen in the table, the false-positive rate initially starts out high when the training set is very small (2 executions), but quickly decreases with increase in the training set size. For a training set size of 6 or more, the false-positive rate is nearly zero. This confirms the earlier observation that a training set size of 6 is the point at which the invariant DOMs stabilize.

Training Set Size	**Match Threshold** 0.95	0.50	0.05
2	53	53	53
4	29	29	29
6	1	1	0
8	1	1	0
10	1	1	0

TABLE IV
False positives versus training set size of a total of 58 executions.

Interestingly, the false positives do not drop to 0 when the match thresholds are 0.95 and 0.50, but remain stable at 1 up to a total of 10 executions. This suggests that one of executions exhibits significant content differences from the invariant DOM. Nonetheless, the false-positive curve drops to 0 when the match threshold is decreased to 0.05, suggesting that the execution conforms to the structural characteristics of the invariant DOM. We cannot fully explain this deviation as we are using a live web-site whose server-side code is not available to us.

Coverage for Event Errors: We measure the error-detection coverage of the invariants for event errors through fault-injection experiments shown in Table II. For each event in the trace (13 in all), we inject a fault corresponding to the event and compare the resulting DOM sequence with the invariant DOM sequence. A mismatch among the sequences indicates that the fault was successfully detected. For each fault, the application is executed five times, so there are a total of 65 executions (= 5 * 13) performed in this experiment. Table V shows the error-detection coverage for each fault (event-number) that was injected. Based on the previous results for false positives, we focus on the invariants derived with training sets of 6 or more. As before, the match threshold was set to 0.95 for these experiments.

Table V shows that the detection rate of the invariants is either 0% (0 detections) or 100% (5 detections) depending on the injected fault. The reason for the differences among events is as follows. Either an event-handler affects the DOM or it does not. The events that have 0 detection rates (namely 1, 3, 7, 9, 11 and 12) are timeout events, and their handlers do not update the DOM, an observation confirmed by examining the handler's source code). The other events are mouse-clicks and message-handling events, and the corresponding handlers either add or remove nodes from the DOM. Thus, the invariants detect all event errors that affect the DOM.

Domain Failures: The goal of this study is to measure the error-detection coverage of the invariant DOMs for failures of

[10]We also varied the inputs included in the training set, but the results were similar and are hence not reported.

[11]Recall that a false positive is an execution that deviates from the invariant DOM sequence derived from the training set.

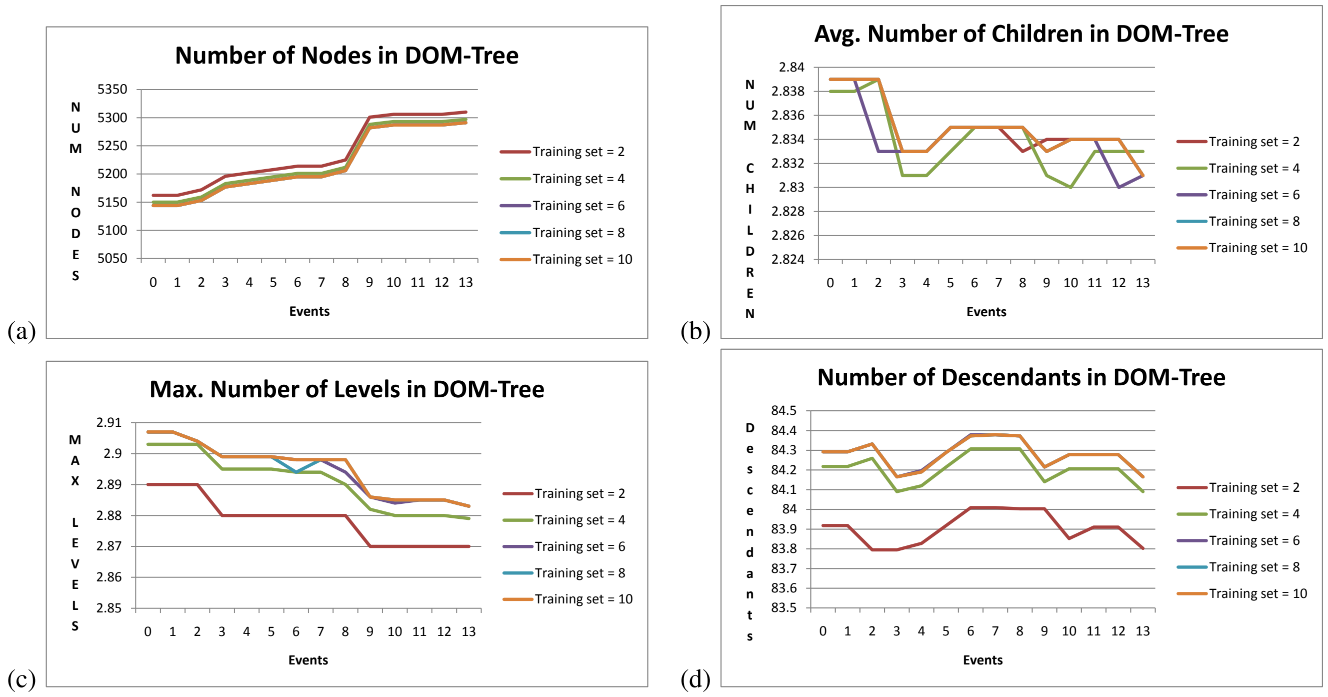

Fig. 4. Invariant Characteristics of the DOM.

Event No.	Affects DOM?	Injected	Detected
1	No	5	0
2	Yes	5	5
3	No	5	0
4	Yes	5	5
5	No	5	0
6	Yes	5	5
7	No	5	0
8	Yes	5	5
9	No	5	0
10	Yes	5	5
11	No	5	0
12	No	5	0
13	Yes	5	5

TABLE V
ERROR-DETECTION COVERAGE OF THE DOM INVARIANTS.

Domain Name	Affects DOM?	Total executions	No. of detections
No domain	No	90	0
doubleClick.net	No	16	0
fsdn.com	Yes	27	27
Google Ads	No	31	0
mediaplex.com	Maybe	81	2
2mdn.com	No	25	0

TABLE VI
DOMAIN FAILURES FOR SLASHDOT: THE LEFT-MOST COLUMN SHOWS THE BLOCKED DOMAIN.

the domains in Slashdot. The experiment involves blocking each domain in the application using the NoScript plugin and replaying the events with DoDOM. Table VI shows the results. In Table VI, the first column shows the blocked domain while the second and third column respectively show the number of executions performed for that domain and the number of executions that resulted in a mismatch between the invariant DOM and the observed DOM[12]. The *no domain* case in which no domain was blocked, yields 0 mismatches, showing that the invariants incur no false positives for this experiment.

We consider two questions with Table VI. First, how many domains affect the DOM and second, how many of these are detected using the invariants. From the table, one can observe that only *Fsdn.com* and *mediaplex.com* exhibit mismatches between the invariant and observed DOMs. Of the two domains, *mediaplex.com* differs from the invariant DOM in only 2 executions out of over 80 executions. Hence, these two executions are likely false-positives. On the other hand, Fsdn.com exhibits mismatches in 27 of 27 executions. Therefore, this domain likely has an influence on the DOM, and its failure is detected by the derived invariants. Hence, the invariants detect failures of all domains that affect the DOM.

[12]The number of executions is different for different blocked domains, because we capped the total time for each experiment to 30 minutes.

Other applications: We run DoDOM on two other web applications, namely *CNN* and *Java PetStore*, to test its invariant extraction capabilities. As before, we measure the convergence of the invariant DOMs for the two applications as a function of the training set size. Table VII shows the results for both applications. The table focuses on the final events in the applications' traces. As can be seen in the table, the invariant DOMs stabilize with a training set size of 6 for both applications. Similar results were obtained for the other events, but are not presented due to space constraints.

Training Set	**Java PetStore**				**CNN**			
Size	*NumNodes*	*NumChildren*	*MaxLevels*	*Descendants*	*NumNodes*	*NumChildren*	*MaxLevels*	*Descendants*
2	397	3.04	2.81	44.2	2413	2.47	2.63	48.4
4	397	3.04	2.94	44.2	2408	2.47	2.63	48.5
6	387	3.10	2.94	45.5	2407	2.47	2.63	48.5
8	387	3.10	2.94	45.5	2407	2.47	2.63	48.5
10	387	3.10	2.94	45.5	2407	2.47	2.63	48.5

TABLE VII
RESULTS FOR JAVA PETSTORE AND CNN APPLICATIONS.

Threats to Validity: An internal threat to validity is the limited number of applications we examined in the study. We chose popular Web 2.0 applications without apriori knowledge of their behavior. However, it is possible that there are web applications that do not exhibit any invariants over their DOMs (see Section VI). An external threat to validity is that our fault-injection is limited to event errors and domain failures. While DOM invariants are effective at detecting the injected errors, it is possible that they may not detect more subtle bugs. Future work will consider more extensive fault models.

Performance overhead: We also evaluate the performance overhead of the DoDOM tool. The JS logger consists of about 1500 lines of JavaScript code. When compressed, this code occupies 16.5 Kilobytes, which constitutes less than 10% of the code loaded by typical Web 2.0 applications [13]. Further, the JS logger incurs an overhead of 3.5 seconds to traverse the entire DOM and install event handlers after the page is loaded. The time taken for Slashdot to finish loading is approximately 10 seconds, so DoDOM adds an overhead of 35% to the initial load time. However, once the page has been loaded, DoDOM incurs negligible overhead in capturing and replaying the events in the trace. Finally, the invariant extraction procedure is performed offline and hence does not contribute to the performance overhead of DoDOM.

VI. DISCUSSION

In this paper, we consider a single kind of DOM invariants, namely those that are specific to a given user interaction sequence. We showed that such invariants are highly effective at detecting errors in the application and can be used in robustness testing of the application. However, it may be possible to generalize the invariants across multiple user-interaction sequences, in order to capture the most typical behavior of the web application under common usage scenarios. Such invariants may have broad uses beyond error detection. We consider some of the use cases below.

Dependability Benchmarking: One of the main challenges in benchmarking the dependability of web applications is in ensuring the repeatability of the benchmarking experiments [14]. This is because web applications exhibit a high degree of variation from one execution to another, and it is challenging to obtain a characterization of the common aspects of the application across different executions. Such a characterization can be provided by the invariants.

Security Enforcement: The invariants can also be used to check if a web page has been tampered with either during transmission or rendering at the client. This is similar to the Web Tripwire project [15], with the difference that we can apply it to arbitrary web applications that execute client-side code. Further, the invariants can also help identify if a web page has been permanently defaced or its contents have been significantly modified (for example, through a Type 2 XSS attack [16]).

Better Domain Filtering: Section V shows that failures of the majority of domains do not impact the invariant DOM for Slashdot. We believe this is also likely to be so for many web applications that include multiple domains. We could filter such domains at the client and prevent them from being loaded in the first place, for advertisement blocking or performance optimization. The NoScript plugin already allows domain filtering but leaves it to the user to decide which domains to block. With an approach such as DoDOM, we can automate the decision making process based on which domains impact the invariant DOM.

Criteria for choosing Applications: An interesting question to ask is "What kinds of web applications are likely to exhibit invariants across multiple user-interaction sequences?" We believe any application that has a fixed static structure (i.e., its backbone) in addition to dynamically generated content will fall into this category. Examples of such applications are news websites, e-commerce sites and online forums. However, certain applications such as office applications or productivity tasks may not satisfy this requirement because their content is highly dependent on the user and the specific task performed and hence may not exhibit generalizable invariants.

Standard frameworks such as Dojo and AJAX.Net are being increasingly used to construct web applications [17]. We hypothesize that applications written using these frameworks are more likely to exhibit invariants over their DOM structures by virtue of following programming patterns that are specific to the framework. We will explore this hypothesis in future work.

VII. RELATED WORK

A number of approaches have been developed to test web applications that execute primarily at the server, i.e., Web 1.0 applications [18], [19]. An example of this approach is Veriweb [18], which systematically explores a web site by navigating to each of its pages. However, Veriweb cannot be applied to Web 2.0 applications which often execute within a single page. Marchetto et al. [20] propose an approach to test Web 2.0 applications using an abstract state machine model provided by the developer. Mesbah and Deursen [7] extend this work to infer the state machine model automatically by finding clickable elements in the application and emulating clicks on them using an automated tool called ATUSA. Similar to our work, they use invariants on the DOM tree to check the validity of a state. Unlike DoDOM however, the invariants used in ATUSA correspond to generic invariants on the validity of the DOM, and are not specific to the web application being tested. Further, while ATUSA allows the programmer to specify other invariants, it leaves open the question of how to derive them.

There has been substantial work on regression testing of web applications [2], [21], [22]. These papers also capture a user's interaction with the application, replay their executions, and automatically characterize the invariant properties of its output. However, they differ from DoDOM in two ways. First, they consider only the server-side state of the application during replay, and hence do not apply to Web 2.0 applications. Secondly, the techniques assume that the web page does not change once it is loaded, which does not hold for Web 2.0 applications that continue to change even after they are loaded.

Concurrent to our work, Roest et al. [23] propose a method for regression testing of Web 2.0 applications by specifying oracle comparators. This work requires developers to manually specify the comparators based on generic templates. In contrast, our approach is fully automatic.

Finally, Swaddler [24] derives dynamic invariants on web applications written using the PHP language and uses the inferred invariants to detect security attacks that attempt to bypass the application's workflow and force the application into an inconsistent state. However, Swaddler's analysis and enforcement is implemented on the server side and hence cannot be used for Web 2.0 applications.

VIII. Conclusion

This paper presents an automated approach to test the robustness of Web 2.0 applications and compare their outputs using DOM-based invariants. The approach automatically derives invariants on web pages' DOMs through dynamic execution and uses the invariants to detect errors. We present *DoDOM*, an automated tool to extract DOM invariants over multiple executions of the application. We show that DOM invariants (1) exist in real web applications, (2) can be learned using DoDOM within a small number of executions (six in our experiments), and (3) can be used to detect event errors and domain failures that affect the DOM with high accuracy.

As future work, we plan to (1) consider more realistic fault models, (2) extract invariants across multiple user-interaction sequences, and (3) implement DoDOM in the web browser.

Acknowledgements

We thank Nikhil Swamy, Benjamin Livshits, Emre Kiciman and Trishul Chilimbi for their insightful comments about this work. We also thank Suzanne Zorn for help with editing this paper and the anonymous reviewers for helpful feedback.

References

[1] T. OReilly, "What is Web 2.0: Design patterns and business models for the next generation of software," 2005.

[2] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock, "Automated replay and failure detection for web applications," in *Intl. Conference of Automated Software Engineering (ASE)*, 2005, pp. 253–262.

[3] A. Guha, S. Krishnamurthi, and T. Jim, "Using static analysis for AJAX intrusion detection," in *Intl. conference on World Wide Web (WWW)*, 2009, pp. 561–570.

[4] T. Chilimbi and V. Ganapathy, "HeapMD: Identifying heap-based bugs using anomaly detection," pp. 219–228, 2006.

[5] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin, "Quickly detecting relevant program invariants," in *International Conference on Software Engineering (ICSE)*, 2000, pp. 449–458.

[6] S. Hangal and M. Lam, "Tracking down software bugs using automatic anomaly detection," in *International Conference on Software Engineering (ICSE)*, vol. 24, 2002, pp. 291–301.

[7] A. Mesbah and A. van Deursen, "Invariant-based automatic testing of AJAX user interfaces," in *International Conference on Software Engineering (ICSE)*, 2009, pp. 210–220.

[8] A. e. a. Le Hors, "Document Object Model (DOM) level 3 core specification," *W3C Recommendation*, 2004.

[9] S. Pertet and P. Narasimhan, "Causes of failure in web applications," *Carnegie Mellon University Tech Report, CMU-PDL-05-109*, 2005.

[10] Microsoft, "Fiddler: Web debugging proxy." [Online]. Available: http://www.fiddler2.com/Fiddler/help/WebTest.asp

[11] K. Pattabiraman and B. Zorn, "DoDOM: Leveraging DOM invariants for Web 2.0 applications' reliability," *Microsoft Research Technical Report (MSR-TR-2009-176)*, December 2009.

[12] Sun-Microsystems. Java Petstore 2.0. [Online]. Available: http://java.sun.com/developer/technicalArticles/J2EE/petstore/

[13] B. Livshits and E. Kiciman, "Doloto: code splitting for network-bound Web 2.0 applications," in *Intl. Symposium on Foundations of Software Engineering (FSE)*, 2008, pp. 350–360.

[14] J. Durães, M. Vieira, and H. Madeira, "Dependability benchmarking of web-servers," *Lecture notes in computer science*, pp. 297–310, 2004.

[15] C. Reis, S. Gribble, T. Kohno, and N. Weaver, "Detecting in-flight page changes with web tripwires," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2008, pp. 31–44.

[16] S. Di Paola and G. Fedon, "Subverting AJAX," in *23rd Chaos Communication Congress*, 2006.

[17] B. Livshits and U. Erlingsson, "Using web application construction frameworks to protect against code injection attacks," in *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2007, pp. 95–104.

[18] M. Benedikt, J. Freire, and P. Godefroid, "VeriWeb: Automatically testing dynamic web sites," in *Proceedings of 11th International World Wide Web Conference (WWW)*, 2002.

[19] F. Ricca and P. Tonella, "Analysis and testing of web applications," in *Intl. Conference on Software Engineering (ICSE)*, 2001, pp. 25–36.

[20] A. Marchetto, P. Tonella, and F. Ricca, "State-based testing of AJAX web applications," in *Intl. Conference on Software Testing Verification and Validation (ICST)*, 2008, pp. 121–130.

[21] K. Dobolyi and W. Weimer, "Harnessing web-based application similarities to aid in regression testing," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2009, pp. 71–80.

[22] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated oracle comparators for testing web applications," *In the International Symposium on Software Reliability Engineering (ISSRE)*, pp. 117–126, 2007.

[23] D. Roest, A. Mesbah, and A. van Duersen, "Regression Testing AJAX Applications: Coping with Dynamism," in *Intl. Conference on Software Testing, Verification and Validation (ICST)*, 2010, pp. 127–136.

[24] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna, "Swaddler: An approach for the anomaly-based detection of state violations in web applications," *Lecture Notes in Computer Science (LNCS)*, vol. 4637, p. 63, 2007.