DoDOM: Leveraging DOM Invariants for Web 2.0 Application Reliability

Karthik Pattabiraman and Benjamin Zorn,
Microsoft Research (Redmond)

Contact: Karthik.Pattabiraman@gmail.com, zorn@microsoft.com

Technical Report MSR-TR-2009-176

Microsoft Research
One Microsoft Way Redmond, WA 98052, USA
http://research.microsoft.com

DoDOM: Leveraging DOM Invariants for Web 2.0 Application Reliability

Karthik Pattabiraman and Benjamin Zorn, Microsoft Research (Redmond)

Abstract

Web 2.0 applications are increasing in popularity and are being widely adopted. However, they are prone to errors due to their non-deterministic behavior and the lack of error-detection mechanisms on the client side. This paper presents DoDOM*, an automated system for detecting errors in a Web 2.0 application using dynamic analysis. DoDOM repeatedly executes the application under a given sequence of user actions and observes its behavior. 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 demonstrate the use of the invariants in detecting errors in web applications due to failures of events and the unavailability of domains.*

1 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 technologies such as JavaScript 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 [2, 28]. They are being rapidly adopted by popular web sites [33] such as Gmail, Bing and Facebook.

Unfortunately, Web 2.0 applications suffer from a number of potential reliability issues. First, as in any distributed application, the division of the application's logic between the server and client makes it difficult to understand their behavior. Further, web applications are non-deterministic and asynchronous, which makes them difficult to debug.

Second, typical Web 2.0 applications aggregate information from multiple domains, some of which may be untrusted or unreliable. The domains interact by calling each other's code or by modifying a shared representation of the page called the Document Object Model(DOM). Therefore, an error in any one domain can propagate to other domains.

Finally, web applications suffer from a lack of visibility into the behavior of the application at the client. Although tools such as AjaxScope [22] provide visibility into the execution of client side code, they do not provide visibility into the behavior of the DOM. However, users ultimately see and interact with the web application through its DOM. Further, the client-side code of an application interacts with the DOM and hence the DOM plays a key part in the end-to-end correctness of an application.

This paper introduces an approach to detect errors in Web 2.0 applications using their DOM structures. The approach analyzes multiple executions of a Web 2.0 application (also called the training set) corresponding to a given sequence of user actions and extracts dynamic invariants based on their DOM structures.

Prior work has shown that dynamic invariants can be used in general-purpose programs for software understanding, testing and error detection [7, 14, 18]. We show that dynamic invariants (1) exist in web applications, (2) can be learned automatically and (3) are useful in detecting errors. *To the best of our knowledge, our work is the first to derive dynamic invariants based on the DOM and apply them for error detection in Web 2.0 applications*.

The main drawback of using dynamic invariants for error detection is the potential for false positives, i.e., it is possible that executions of the application that do not conform to the invariants are deviant executions rather than erroneous ones. However, as we show, false positives can be minimized by using a representative set of executions for training. The main contributions of the paper are as follows:

1. We show that DOM structures can be used for checking the correctness of a web application.

2. We show that Web 2.0 applications exhibit invariants over their DOM structures and that the invariants can be learned dynamically.

3. We build a tool DoDOM, to repeatedly iterate over the sequence of events in an application to obtain its invariants (and hence the name).

4. 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 ten executions of each application.

5. We further show (for Slashdot) that the derived invari-

ants provide close to 100% coverage for failures of event handlers and domains that impact the DOM.

2 Overview

In this section, we outline the potential reliability issues with Web 2.0 applications and present our proposed solution of dynamically extracting invariants. In this paper, we apply the invariants for error detection. However, the invariants we extract can be applied in a wide variety of circumstances which we outline in Section 6.

2.1 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 that must be exchanged with the server and makes the application more interactive to the user. Henceforth, when we say web applications, we mean Web 2.0 applications unless we specify otherwise.

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 using JavaScript. The event-handlers in turn 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 code (scripts).

A web page is represented internally by the browser as its Document Object Model (DOM) [23]. The DOM consists of the page's elements organized in a hierarchical format. Each DOM node represents an element of the page. 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 three scripts as its child nodes.

JavaScript code executing in the browser can read and write to the DOM through special APIs. Any changes made to the DOM are reflected in the web page rendered by the browser. User actions are converted into events by the browser and sent to the nodes of the DOM on which they are triggerred.

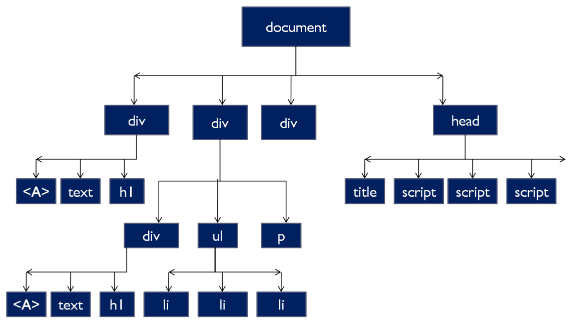

Figure 1. Example of a DOM tree for a web application

Browsers allow the separation of scripts into inline frames and provide isolation between scripts that do not belong to the same frame/domain (also called the same-origin-policy [36]). However, this separation severely curtails the ability of scripts to interact with each other and hence many web applications include scripts into a page using the *script* tag. This allows scripts to interact with each other through the DOM or by directly calling each other's functions. Unfortunately, this practice also means that failures of one domain can affect other domains.

Finally, failures of web applications are handled differently from failures of traditional applications. When a traditional application throws an exception or behaves unexpectedly, it is terminated by the operating system (in most cases). Browsers however allow web applications to continue executing even if an event handler fails or if a domain is unavailable (this behavior is necessary for backward compatibility reasons and is also mandated by the standard). This can lead to semantic violations of the application's behavior that are difficult to detect.

2.2 Fault Model

This section discusses the faults of web applications considered in this study and their potential causes. This is not an exhaustive list of faults in web applications, but only a first cut at characterizing potential errors in these applications (we are not aware of any previous work in this space).

Event drops: These correspond to events being dropped in the client-side code of the application. We consider three kinds of event drops and their causes.

(A) *User-event drops*: Caused by exceptions in event handlers, or the browser terminating the handler because it runs for too long (both Internet Explorer and Firefox do this). It is also possible that a DOM node fails to propagate an event to its parent nodes as required to do so by the DOM [23] specification.

(B) *Server message drops*: Caused by many factors, including (1) network losing the message, (2) server dropping the request due to overload, (3) a message handler throwing an exception or not being invoked in the first place.

(C) *Timeout drops*: This can happen due to (1) the timeout handler throwing an exception, (2) timeout handler not being set correctly by the timer initialization routing, or (3) a browser bug that prevents timeouts from being triggered.

Domain failures: These correspond to cases where a domain included by the application is unavailable. This can happen due to a network failure or the servers of the domains being taken offline. It can also occur due to client-side plugins such as NoScript [24] or administrative proxies which may block scripts from certain domains.

2.3 Scenario

Web applications can be subject to different kinds of faults as shown in Section 2.2. Consider an application developer who wants to test the resilience of the application to faults. She would execute the application on the client, inject a fault and check if the application behaves as expected after the injection. 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 (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). Finally and most importantly, web applications exhibit variations in their behavior from one execution to another as we show later in this paper. Such variations occur as a result of (1) systematic changes introduced by the server (e.g., in showing different advertisements on the page), (2) asynchronous behaviour at the client (e.g., in delivering user actions as events), and (3) non-determinism in network packet delivery (e.g., messages received out of order). 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 a user-interaction sequence and not necessarily at application load time, which makes it challenging to reason about the effects of faults during testing.

The problem we address in this paper is a concrete way to test the resilience of a web application to faults. We assume that the web developer has one or more user interaction sequences under which she wants to exercise the application. Our solution involves extracting a invariant characteristic of the web application's DOM from multiple executions of the application. We characterize the expected

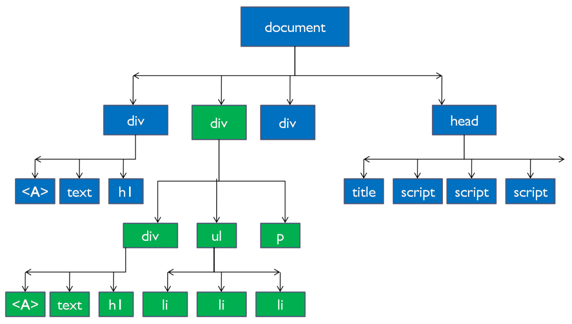

Figure 2. Proposed solution in the context of the example DOM

behavior of the application and consider significant deviations from this behavior as an error.

2.4 Dynamic Invariant Extraction

This section illustrates our proposed solution for the problem illustrated in Section 2.3. 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 portion of the application's DOM across multiple executions (each of which corresponds to the user-interaction sequence), 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 invariants. A deviation between the trees indicates an error.

Figure 2 shows the invariant portion of the DOM for the example considered in Figure 1. In the figure, the blue(dark) nodes represent the invariant portions of the tree (also called the web page's backbone) while the green(light) 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 'A' element in the invariant DOM but fails to do so due to an error (e.g., the handler throws an exception). This error will be detected as the resulting DOM would deviate from the invariant DOM[1] for the event (mouse-click event).

Second, assume that the web application imports a faulty script (from a different domain) using the 'script' tag. The script has access to the entire web page's DOM. However, it is not expected to modify any portion of the invariant DOM. Assume that the faulty script modifies the center div element in the tree. This element is not a part of the invariant

[1]The invariant DOM will have the modification while the resulting DOM will not because of the error.

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_{1_I}	...	T_{n_I}

Table 1. Invariant DOMs

DOM and hence the modification is not considered an error. On the other hand, assume that the script attempts to modify the div element in the far left branch of the tree. This element is part of the invariant DOM and hence the modification will be detected as an error.

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 various events. In reality, the technique will capture the entire sequence of invariant DOMs and use the invariant sequence to check for deviations. We now give a more precise definition of the invariant DOM.

An invariant DOM is a sub-tree of the web application's DOM that is shared by multiple executions. We derive an invariant DOM for each event in the application corresponding to a given event seqeunce. Table 1 shows the DOM trees obtained during multiple executions of the web application for each event in the sequence. The rows of the table indicate different executions of the web application, while the columns indicate the events in the sequence. The DOMs are indicated by T_{i_j}, where i is the event after which it is obtained and j is the corresponding execution. The invariant DOMs T_{i_I} are derived from the DOM trees of individual executions T_{i_j} corresponding to the event i.

2.5 Challenges

There are three major challenges in extracting the invariant DOMs for error detection. First, web applications exhibit small changes from one execution to another due to variations at the server and at the client (as explained in Section 2.3). The invariant DOM must not include such changes as otherwise it will incur false positives.

Second, the invariants extracted should retain as much of the original DOM as possible. This is to ensure high detection coverage of the invariants for reliability and security violations. We show that our approach achieves both high detection coverage and low false-positive rates (Section 5).

Finally, the invariant-extraction system should be compatible with different browsers and platforms, i.e., it should not require adding new functionalty to existing browsers. Further, it should not require any modifications to the server-side code of web applications[2].

[2]The last constraint may not be as important if the server-side code is under our control, but this is often not the case

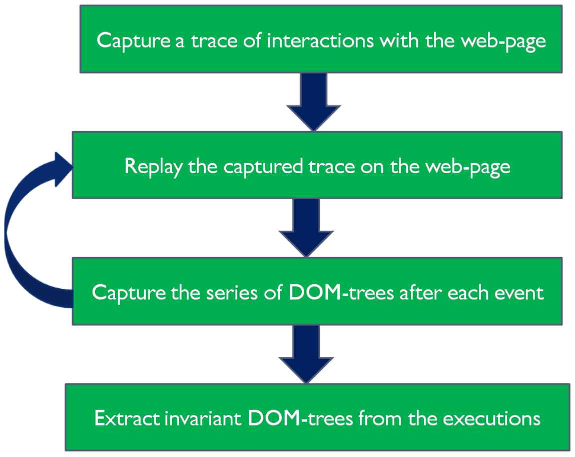
Capture a trace of interactions with the web-page
Replay the captured trace on the web-page
Capture the series of DOM-trees after each event
Extract invariant DOM-trees from the executions

Figure 3. Steps in learning DOM invariants

3 Approach

The overall approach for extracting and learning invariants over the DOMs of web applications is as follows (shown in Figure 3). First, we record a sequence of user interactions and events on a page (the sequence of events is called a trace). We then replay this trace over multiple executions and capture the sequence of DOMs generated after each event in the trace. We then extract invariants over the set of all DOM sequences using an offline learning process. The recording, replay and capture process is performed using a tool we built called DoDOM. In this section, we describe the architecture of DoDOM and its operation. We also discuss the design choices and the trade-offs made in DoDOM.

3.1 DoDOM Architecture

Figure 4 shows the architecture of the *DoDOM* system. In the figure, the gray rectangles represent existing code bases and tools and the green rectangles are the components we added for implementing DoDOM. They are as follows.

The **proxy** is written as a plugin in the Fiddler web application testing framework [9] using the .Net environment. Its 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 write them to the file system. Further, the proxy intercepts messages sent between the client and the server and records them. The JS logger communicates with the proxy by appending a special suffix to its messages and sending it through AJAX. The proxy identifies these special messages and takes appropriate action without forwarding them to the server. Hence, the server-side code does not need to be modified to deploy DoDOM.

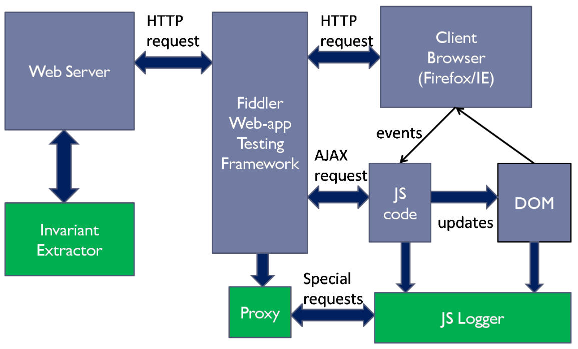

Figure 4. Architecture of DoDOM system

The **JS logger** is a piece of JS code that is executed on the client's browser. A separate instantiation of the JS Logger is inserted for each inline frame 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). Finally, it can intercept all communications between the webpage's JS code and the *window* object, e.g., timeout events. The JS logger performs the core operation of DoDOM and runs within the browser. Since it is written using JS, it is highly portable.

The goal of the **invariant extractor** is to perform offline analysis of multiple executions recorded by the proxy and to extract the invariant DOMs. It runs outside the browser.

3.2 Operation

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 which is used to identify the script in interactions with the proxy. 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) Traverses the DOM of the web page, creates a compact representation of the DOM, and sends it to the proxy. (2) Installs a new replacement handler for all DOM elements that have event handlers installed 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 sub-tree rooted at the element.

The **JS logger** operates in two modes: record and replay. We first describe the operation of the system in the record mode and summarize the main differences during 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 (one at a time). The JS logger then invokes original handers corresponding to the event stored in the element. The proxy also writes the events to an event log before sending it to the client.

During *replay*, the above sequence of steps is repeated, but with three differences. First, instead of waiting for events from the JS logger, the proxy reads in the list of events from the event log and populates the global queue. When the JS logger polls for events, the proxy retrieves the events from the queue one at a time and sends them to the client with the corresponding time-delay. Second, the web page may have undergone small changes between the recording session and the replay session, and the DOM node that triggered the event during the recording session may be in a different position during replay. DoDOM uses the contents of a node and its pre-order traversal index to match the node during replay. In the case of a mismatch, it searches the DOM for the node with the closest match and replays the event at the node [3]. Third, only those timeouts that expired during recording are allowed to expire during replay and only if the timeout handlers match. This ensures that the effects of a timeout expiring during replay are faithful to the effects observed during recording.

The proxy records the changes made to the web page's DOM tree after every event (i.e., user action, timeout or received messages). The **invariant extractor** post processes these traces to obtain a sequence of invariant DOM trees for each event. This sequence represents the invariant DOM shared by each of the executions (modulo small differences due to variations among replays). Each tree in the invariant sequence is learned independent of the other trees based on the corresponding trees in the individual executions.

The **algorithm** for learning the invariant DOM trees from a set of execution traces is given below (the pseudocode for the complete algorithm can be found in 3.2).

We set the initial invariant tree to the tree obtained from the first execution trace. For each execution trace, we compare its tree with the corresponding invariant tree recursively starting from the root nodes and traversing the tree in post order. A node is removed from the invariant tree due

[3]The closest match is the node(s) in which the highest number of fields and values match with the original.

to any of the following three conditions. (1) the contents of a node of the invariant tree does 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.

The match threshold thus determines how much of the invariant tree is pruned away due to 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.

3.3 Discussion

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 as follows.

- The ordering of events in the JavaScript Virtual Machine (VM) cannot be observed or controlled by DoDOM. These may impact the behavior of some applications. However, a robustly designed page must not depend on the ordering of events by a specific VM, and hence this is not a significant limitation.

- DoDOM traps events by hooking into the event-handlers of DOM nodes and replacing the existing functions with a wrapper function. This requires that the web page be written using the DOM 1.0 event-model. 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 of the web pages we observed were written using the DOM 1.0 model for comaptibility reasons. Further, the DOM 3.0 specification remedies thi situation, but is not yet implemented by most browsers.

- DoDOM assumes that the communications between the JS logger script and the client-side proxy is First-In-First-Out (FIFO), i.e. messages sent by the proxy are delivered in-order to the JS logger code. If this assumption is violated, it may not be possible to isolate the effects of an event. However, we found that this assumption was satisfied on our platform, namely Firefox on Windows.

```
int MatchScore(HashCode H1, HashCode H2)
{
  int score = 0, total = 0;
  for (Field F, Value V) in H1 do:
    if H2.hasField(F) and (H2.getValue(F)==V)
       score = score + 1;
    total = total + 1;
  return (score / total);
}

Set CheckMatch(TreeNode N1, TreeNode N2)
{
   Set diffSet = empty;
   HashCode H1 = getHash(N1);
   HashCode H2 = getHash(N2);
   if (MatchScore(H1, H2) < matchThreshold)
     diffSet.add( N1, N2, "content" )
     Children C1 = getChildren(N1);
     Children C2 = getChildren(N2);
     numChildren = min( C1.length(), C2.length() )
     for (i = 0; i<numChildren; ++i) {
        deltaSet = CheckMatch( C1, C2 );
        diffSet = diffSet Union deltaSet;
    }
    if (C1.length > numChildren) {
       for (i=numChildren; i < C1.length; ++i)
          diffSet.add( N1, N2, "remove" )
    } else if (C2.length > numChildren) {
         for (i=numChildren; i < C2.length; ++i)
            diffSet.add( N1, N2, "append" )
    }
    return diffSet;
}

Tree pruneInvariants(Tree InvTree,
   Tree TraceTree)
{
    Set diffSet = CheckMatch(InvTree.root,
                           TraceTree.root);
    for (element in diffSet) do {
       (node1, node2, diffType) = element
       if (diffType == 'remove')
          removeFromTree(InvTree, node1);
       else if (diffType == 'content')
          removeFromHash(InvTree, node1);
       }
       return InvTree;
}
```

Figure 5. Pseudo-code for extracting invariants from DOMs

- DoDOM cannot capture the state of the JS heap as it is implemented entirely in JS. While it is possible to route all object allocations on the JS heap through DoDOM, such a mechanism would incur high overheads. Hence, we do not consider the JS heap in DoDOM. This is not a significant limitation as changes to the JS heap that impact the page will be reflected in the DOM, server messages or timeouts and would hence be captured by DoDOM. As for the other changes, they do not matter from the point of view of the page and are hence not captured.

- If for any reason, the JavaScript VM crashes or hangs, then so does DoDOM. This is not a significant limitation as most JS VM's are reasonably robust[20]. We do however provide a heartbeat service to detect hangs of the JS logger and reload the page if it hangs.

4 Experimental Setup

This section describes the experiments performed and the benchmarks used in our evaluation. The experiments were performed using Firefox on a Core-2-Duo Intel processor machine (running at 3 GigaHertz each) with 4 GigaBytes RAM and running Windows (Vista).

We summarize the main research questions answered by the experiments.

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

2. How effective are the invariants in detecting errors due to dropped events?

3. How effective are the invariants in detecting failures of domains in the web application?

4. How much variation do the invariants exhibit from one day to another ?

4.1 Invariant Extraction

The goal of this experiment is to answer research question 1. From the set of all executions (replay sequences), we randomly choose a training set, which is a subset of executions used to learn the invariants. In these experiments, we vary the training set size in order to understand how quickly the invariants converge to a stable value. We also vary the match threshold described in Section 3.2 to understand how much content similarity is present among the executions.

We measure the following characteristics of the DOM tree in order to measure its convergence. (1) total number of nodes, (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 (not only its immediate children).

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

Table 2. Faults injected and their characterization

We also compare the invariant DOM sequence with the DOM sequences from all executions (even if they are not in the training set). 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.

4.2 Event Drops

The goal of this experiment is to answer research question 2. We measure the error-detection coverage of the invariant sequences for event drops corresponding to those in Section 2.2. We observe their effects on the web page's DOM. Table 2 shows the types of faults introduced and the injection method. The fault injector is implemented as an enhancement of the replay mechanism in the DoDOM system and can be configured through an external file. Each run injects at most one fault to ensure that its 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 its sequence exhibits a mismatch with the corresponding DOM in the invariant sequence.

4.3 Domain Failures

The goal of this experiment is to answer research question 3. It emulates the effect of domain failures as described in Section 2.2 using the NoScript plugin in Firefox [24]. First, the invariant DOM sequence is obtained from multiple executions as shown in section 4.1. 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 then compared to the invariant DOM sequence. A mismatch indicates that the domain's failure is detected by the invariants.

4.4 Diurnal Variations

The goal of this experiment is to answer research question 4.

Because web pages may exhibit day-to-day changes, we gather executions of the application over multiple days to study the effect of diurnal variations in the effects of the blocked domains on the web page. The invariants obtained on each day are compared with the executions obtained by blocking a domain (on a particular day) to determine whether the blocked domain impacts the backbone of the web page on a given day. This allows us to understand whether the effects of a domain are local to a given day or whether they span multiple days.

4.5 Benchmarks

Table 3 summarizes the characteristics of the three web applications. It shows the number of domains in the application, the total number of lines of JS code (obtained with Firefox's Phoenix plugin [29]) and the number of events in the recorded trace for the application. We use the Slashdot web site (*http://slashdot.org*) as the primary source of measurements in this paper.

Slashdot aggregates technology-related news from different websites and allows users to comment on a news story. Slashdot also allows users to navigate among different comments and view/expand comments on demand. We interact with a Slashdot news story normally and replay the interactions using DoDOM. The results reported are for a specific news story on Slashdot[4] with close to 300 comments. We perform a total of 58 replays for the story (over the course of 1 hour). We also repeated the experiments with a different story with about 50 events, but the results were similar and are hence not reported.

We also evaluate the invariant extraction capabilities of DoDOM on two other web applications, namely CNN and Java Petstore. Java Petstore is a freely available Web 2.0 application that mimics an e-commerce website for buying pets [10][5]. It allows the user to browse through pet listings and choose a pet corresponding to the user's preferences. We interact with the website by moving over and clicking on different elements of the page. CNN is a widely read news website that delivers customized content to its readers using JavaScript. We study the main page of CNN, which is highly dynamic. Our interaction with CNN also consists of moving the mouse over various elements of the page and clicking on news stories of interest.

Table 3 shows that Java Petstore is the least complex application in terms of the number of lines of JS code while CNN is the most complex. We choose to focus on Slashdot for our experiments as it presents a middle ground among the three applications.

[4]The story is "http://hardware.slashdot.org/story/09/07/30/2147246/ARM-Hopes-To-Lure-Microsoft-Away-From-Intel"

[5]Java PetStore is written using the Dojo programming framework, which registers event handlers for almost all events in the page, and hence the large number of events in this application

Website	Lines of JS	No. domains	No. events
Java Petstore	499	1	211
Slashdot	9647	5	13
CNN	15603	9	9

Table 3. Characteristics of the web applications

5 Results

This section presents the results of the experiments described in Section 4. We focus on the Slashdot application for the bulk of the experiments and summarize the results for the CNN and Java PetStore applications at the end. We also measure the performance overhead of DoDOM.

We summarize the main results first.

Convergence of invariants (Section 5.1): We show that the invariant DOM converges with a training set size of 6 executions, which corresponds to 10% of the total executions.

Coverage for event drops (Section 5.2): The invariants detect 100% of the drops of events whose handlers affect the DOM.

Fault impact (Section 5.3): We find that most faults have little or no impact on future events. However, faults that do affect future events are more likely to impact events closer to their origin in the execution trace.

Coverage for domain failures (Section 5.5): We also find that failures of 4 of the 5 domains included by Slashdot have no effect on the DOM. Hence, they are not detected by the invariants. However, the invariants detect 100% of the failures of the one domain that affects the DOM.

Impact of day-to-day variations (Section 5.6): Finally, we find considerable variations among the invariants obtained on a day-to-day basis for Slashdot. However, DoDOM is able to learn a consistent set of invariants for the application across multiple days with no false positives on any day.

Other applications(Section 5.7): Finally, we find that the other two applications, namely Java PetStore and CNN, also exhibit invariants that can be learned within six executions (similar to Slashdot).

5.1 Invariant Extraction

In this experiment, we vary the training set size 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 4 are shown in Figure 6 (a) through (d).

In the figures, the X-axis represents the event number and the Y-axis represents a metric corresponding to the

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 4. False positives versus training set size

event. The lines represent the invariants obtained with a training set of a specific size. The figures show that the number of nodes monotonically increases with the event number, while the maximum number of levels monotonically decrease. The other two metrics, namely the number of children and the number of descendants show no consistent trend. This shows that the events are progressively adding nodes to the tree although the number of descendants does not change much because the number of leaf nodes in the tree correspondingly increase.

Figure 6 (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, it stops decreasing once the training set reaches a size of 6 (roughly 10% of the executions). Nonetheless, the number of nodes converges to a value that is within 1% of its original value (i.e., the number of DOM nodes for any given execution). Similarly, the the average number of descendants steadily increase with increasing the training set size, but also stabilizes at a training set size of 6. This shows that the invariant DOMs can be learned with a relatively modest training set of 6 executions.

To further confirm the convergence of the invariants, we measure the false-positive rate for the executions. Table 4 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 3.2). In the figure, the X-axis represents the training set size and the Y-axis represents the number of false positives. As can be seen in the figure, 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 (see below). This confirms the earlier observation that a training set size of 6 is the point at which the invariant DOMs stabilize.

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 (maximum in this experiment). This suggests that one of executions exhibits significant differences from the invariant DOM. Nonetheless, the false positives drop to 0 when the match threshold is decreased to 0.05, suggesting that the deviation is due to content differences among the DOMs rather than structural differences. We are investigating this case further.

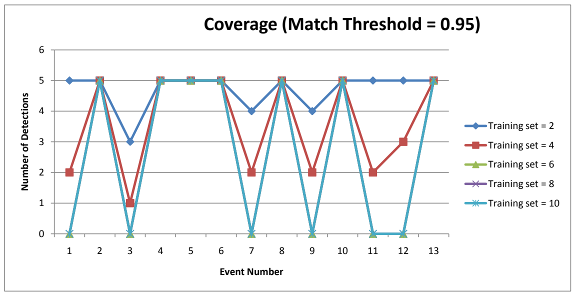

Figure 7. Error-detection coverage of the DOM invariants

5.2 Coverage for Event Drops

We measure the error-detection coverage of the invariants for event drops through fault-injection experiments shown in Table 2. For each of the 13 faults (each corresponding to an event in the trace), we perform 5 fault-injection runs and compare the resulting DOM sequence with the invariant DOM sequence. A mismatch among the sequences indicates that the fault was successfully detected. Figure 7 shows the error-detection coverage as a function of the fault (event-number) that was injected. Based on the previous results for false positives (Section 5.1), we focus on the invariants derived with training sets of 6 or more.

Figure 7 shows that for invariants with training sets of 6 or more, the detection rate is either 0% (0 detections) or 100% (5 detections) depending on the injected fault. The reason for the differences 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 these events only update the global JavaScript heap and not the DOM[6]. The other events are mouse-clicks and message-handling events and the corresponding handlers add or remove nodes from the DOM. Thus, the invariants detect all event drops that affect the DOM.

The match threshold was set to 0.95 for these experiments. However, the above results are not affected by varying the match threshold from 0.95 to 0.05 (not shown in figure). This suggests that the coverage is due to structural differences among the DOMs rather than content differences.

Further, a match threshold of 0.05 has identical coverage to a match threshold of 0.95, although it has a lower false-positive rate (Section 5.1). Therefore, a match threshold of

[6]We confirmed this observation by reading their JavaScript code.

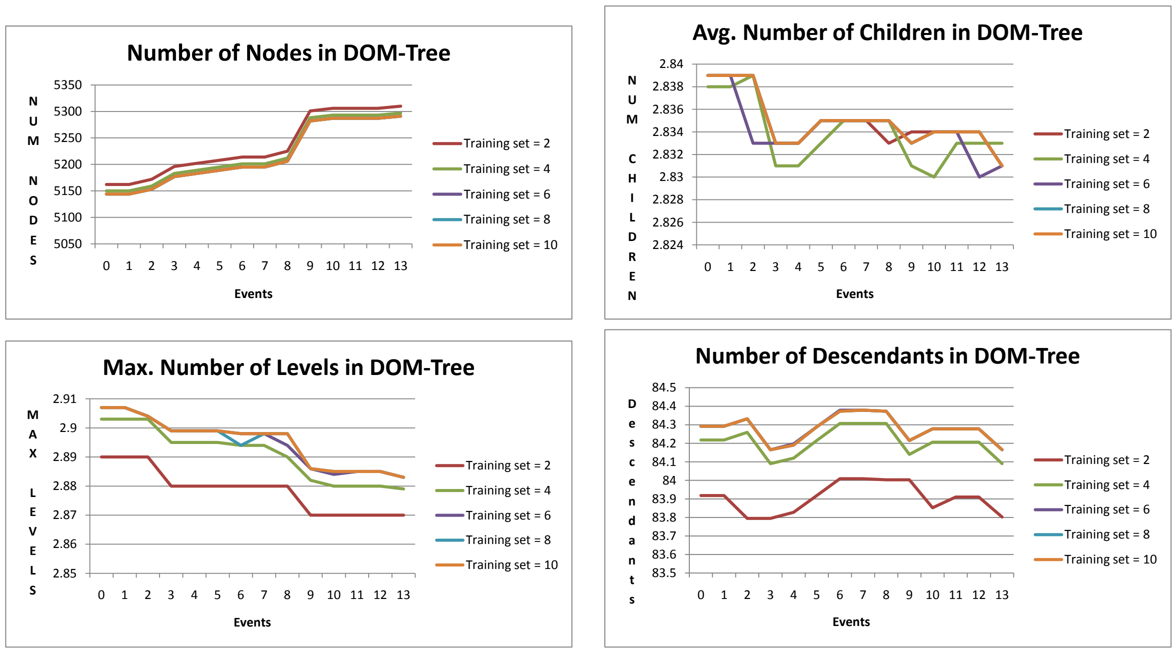

Figure 6. Invariant Characteristics of the DOM: (a) Total number of nodes, (b) Average number of children per node, (c) Maximum number of levels in each node and (d) Average number of descendants per node

0.05 represents an optimal tradeoff between detection coverage and false-positive rate.

5.3 Fault Impact

In this study, we measure the impact of a fault by comparing the DOM sequence from the fault-injected executions with the invariant DOM sequence. The training set size for the invariants used in this study was fixed at 6 and the match threshold set to 0.95. For each fault injected execution, Figure 8 shows the number of nodes in the DOM that are different from the corresponding invariant DOM sequence as a function of the event in which they differ. Each line in the graph represents the average of five injection runs for a particular fault (identified based on the event number that is dropped). Only the faults that were detected in Figure 7 are shown in Figure 8.

The following observations may be made from Figure 8. First, the total number of DOM nodes that differ from the invariant DOMs is typically less than 25 for each event and fault. This is relatively small compared to the total number of nodes in the tree (over 5000), suggesting that the impact of a fault is confined to a small fraction of nodes. Second, for all faults except fault 2 and fault 8, the number of DOM nodes that differ from the invariant tree is a constant independent of the event number. This suggests that the portion of the DOM affected by the corresponding events is not influenced by future events. Finally, for faults 2 and 8, the number of DOM nodes affected by the fault first increases and then decreases before converging to a constant value. This suggests that the events corresponding to the faults influence the behavior of the events that immediately follow them in the trace. However, the impact decreases with increasing distance from the fault's origin and tapers to a constant value (after about three events in each case).

5.4 Aggregate Invariants

In this paper, we have derived invariants based on both the structure and content of DOM trees as shown in Section 3. In this section, we consider only the aggregate properties of the invariant DOMs and evaluate their error detection capabilities. The aggregate measures used correspond to those in Section 5.1. We also measured the false-positive rate for the aggregate invariants. The results for coverage and false-positive rates are shown in Figure 9 and Figure 10. Each bar in the figures represents the aggregate invariants derived using training set sizes ranging from 2 to 10. In the false-positive graph, the total number of executions on the Y-axis was 58, while in the coverage graph, the total number of executions on the Y-axis was 65 (13 faults, 5 runs each).

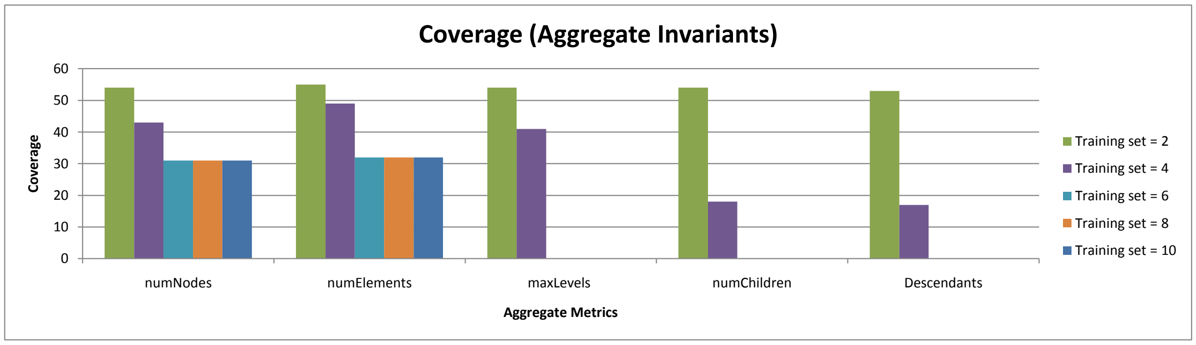

Figure 9. Error-detection coverage of aggregate invariants

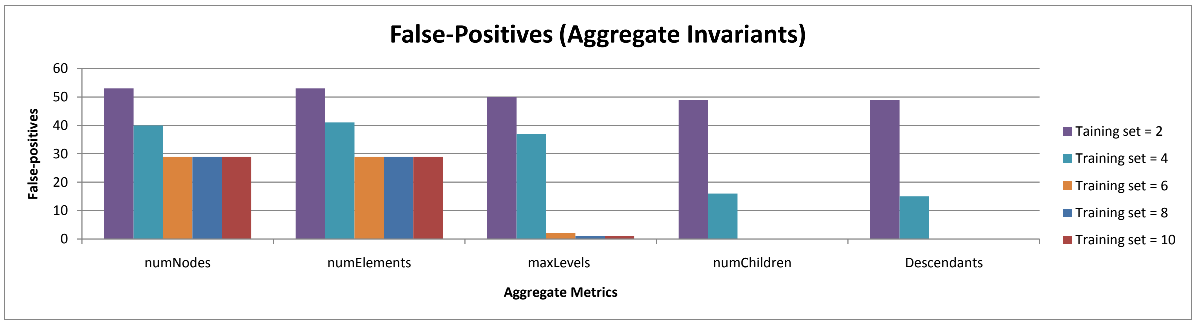

Figure 10. False-positive rates of aggregate invariants

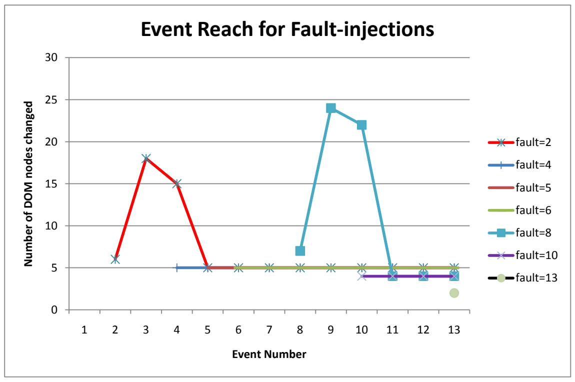

Figure 8. Impact of injected faults on the DOM

The main results from Figures 9 and 10 are summarized here. First, we see that both the false-positive rate and the error-detection coverage decrease as the training set increases from 2 to 6 after which they more or less stabilize (with the exception of false positives incurred by the maxLevels metric). This is in line with the results reported in Section 5.2. Therefore, we consider only training sets of 6 or more in this discussion. Further, both the number of nodes (numNodes) and number of elements (numNodes) have high error detection coverage, but also exhibit a high false-positive rate. Therefore, they are unsuitable as detectors because they exhibit high variations from one execution of the web application to another. However, the other three metrics, namely, the maximum number of levels (maxLevels), average number of children (numChildren) and number of descendants (numDescendants) have a low false-positive rate, but also have almost zero error-detection coverage. Hence, they are ineffective at detecting errors as the information they capture is too coarse-grained to detect small changes in the DOM due to errors (as we showed in Section 5.2 the injected faults resulted in changes in relatively few DOM nodes). Therefore, we do not consider aggregate metrics in this paper, although we do not rule out the possibility that there may exist aggregate metrics other

Domain	Total executions	No. of mismatches
doubleClick.net	16	0
Fsdn.com	27	27
Google Syndication	31	0
mediaplex.com	81	2
2mdn.com	25	0
No domain	90	0

Table 6. Domain failures for Slashdot

than the ones we have considered that may be effective at detecting errors. This is a direction for future investigation.

5.5 Domain Failures

The goal of this study is to measure the error-detection coverage of the invariant DOMs for failures of the domains in Slashdot. The Slashdot application imports scripts from a number of different domains as shown in Table 5. Table 5 shows that some of the scripts are obfuscated and use unsafe practices such as the introduction of new code into the page. The results of the study are shown in Table 6 and summarized below.

First, the number of executions is different for different blocked domains. This is because we capped the total time for running each experiment to 30 minutes for each blocked domain. Blocking different domains has different effects on the time taken for each execution. For example, domains containing a lot of complex JS code take longer to execute and blocking these domains substantially speeds up the execution of the web application.

Second, among 90 executions in which no domain was blocked, none affected the invariants for the page, showing that the false-positive rate for the experiment was 0. Hence, the invariants were stable for the training set (as in previous experiments, the match threshold of 0.95).

Finally, each domain either impacts all the executions in which it is blocked, or it does not impact any execution[7]. Of the five domains, only fsdn.com affects the DOM tree and when it is blocked, its failure is detected by the invariants. The failures of other domains are not detected as they do not affect the DOM. Hence, the invariants detect 100% of the failures of the domain(s) that impact the DOM.

5.6 Diurnal Variations

In this section, we attempt to understand the day-to-day variations of the invariants obtained in section 5.1. To this end, we collected execution traces from the application by replaying a set of recorded events over three different days (we did not block domains for this experiment).

We collected a total of 90 executions, of which each day contributed 30 executions. We then derive four sets of invariants from the traces. Three sets of invariants are derived from a training set of 10 executions each drawn from a specific day, and one set of invariants is derived by combining together the training sets of three days to form a training set of 30 executions. We then compare each set of invariant DOMs with the DOMs of the executions and measure the false-positive rate with respect to each set of invariants (since no faults are injected, all deviations are false positives). The results are shown in Figure 11. The X-axis shows the executions categorized by the day on which they were obtained while the Y-axis shows the percentage of false positives. Each bar in the graph corresponds to the false positives with respect to the one of the four invariant sets described above. The match threshold used was 0.95.

The following results may be observed from Figure 11. First, the invariants obtained across all three days do not exhibit false positives for any of the executions. Second, the invariants for days 2 and 3 do not exhibit false positives for the executions obtained during those days. This suggests that the web application did not change significantly during those two days. However, the invariants for day 1 exhibit false-positive rates of 20% and 50% for the executions obtained during days 2 and 3 respectively. Likewise, the invariants for days 2 and 3 exhibit false-positive rates of about 80% each for the executions obtained during day 1. This suggests that the web application underwent significant changes between days 1 and 2. This result is not surprising, as web applications are in constant flux and may change on a day-to-day basis. However, even with the changes, DoDOM was able to extract a stable set of invariants for the application.

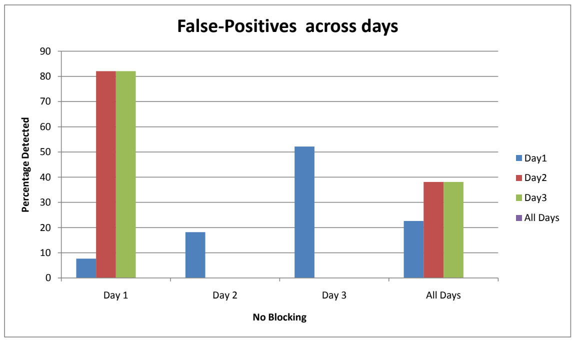

Figure 11. Results of day-to-day variations for Slashdot

[7]Mediaplex.com is an exception - it differs in 2 of 81 executions from the invariant DOM. We believe these executions are false positives.

Domain Name	Lines of JS code	Comments
doubleclick.net	555	1. Uses document.write to insert new scripts and function calls into the document, 2. Writes into global objects on the JavaScript heap shared by all scripts on the page.
c.fsdn.com	7460	1. Complicated code that is obfuscated and compressed using JSCrunch, 2. Adds and removes DOM tree nodes based on user's input
Google syndication	1451	Shows advertisements on the page using the Google advertisement service
s.fsdn.com	181	Storage of media files and scripts for Slashdot
Variable domains	Unknown	These domains vary for different news stories or even when the same story is reloaded. Examples are eyewonder.com, mediaplex.com, 2mdn.com

Table 5. Domains from which Slashdot imports scripts

5.7 Other applications

In this section, we summarize aggregate results from running DoDOM on two other web applications, namely *CNN* and *Java PetStore*. We measure the convergence of the invariant DOMs for the two applications as a function of the training set size. Table 7 shows the results for both applications. The table focuses on the final events in the respective applications' traces. As can be seen in the table, the invariant DOMs stabilize with a training set size of 6 for both applications.

5.8 DoDOM Performance Overhead

In this section, we measure the performance overheads introduced by the DoDOM tool. The overhead consists of three main components.

Page load: The load time of a page depends on the amount of data that is downloaded by it. This includes both HTML and JavaScript code. DoDOM consists of about 1500 lines of uncompressed JS code, which corresponds to 16.5 KB in compressed form. The amount of JS code loaded by Web 2.0 sites ranges from 200 KB to 1.9 MB [31] and hence DoDOM adds less than 10% to the code size.

Record Mode: After the page has loaded, DoDOM performs a complete traversal of the web page's DOM and sends it to the proxy (in a compact form). This operation takes approximately 3.5 seconds for the Slashdot page (we use the Firebug profiler for the measurements). The time taken to run the initial scripts on the page without DoDOM is approximately 9.5 seconds. Therefore, the initialization overhead of DoDOM for the Slashdot application is 36%. DoDOM also introduces delays in capturing user interactions as they must be sent to the proxy. However, we did not observe any noticeable delay when using DoDOM for the applications considered in the paper.

We currently implement DoDOM using JavaScript, which is not optimized for performance. A more efficient implementation as a browser plugin would substantially speed up its operation and is a direction for future work.

Invariant Extraction: The invariant extraction process is done offline and is not in the critical path of the application. The overhead is on the order of a few minutes.

5.9 Threats to Validity

In this section, we discuss some of the threats to the validity of the results obtained in the study. First, the key assumption made in this study is that the invariant DOM does not change substantially from one execution to another. However, this assumption may be violated if either the server-side code undergoes upgrades, or if the content of the web page undergoes substantial changes. If this happend, then the invariants need to be relearned.

The second threat to validity is the assumption of consistency in the behavior of web applications under failures. For example, we assume that a significant deviation from the invariant DOM is due to an event being dropped (Section 5.2) or due to a domain being blocked (Section 5.5). However, it is possible that factors such as load on the server or measurement noise at the client result in spurious deviations. Future work will focus on eliminating spurious deviations.

6 Discussion

Although not the focus of this paper, the invariants extracted by DoDOM have uses beyond error detection. We outline some of the uses in this section.

Web Applications Testing: One of the challenges in testing Web 2.0 applications is in determining the correct or acceptable behavior of the application under different test inputs. ATUSA [26] uses programmer-specified invariants on the DOM to exhaustively test all paths in a Web 2.0 application and ensure their conformance. However, writing invariants is time and effort intensive. DoDOM can be used to automate the extraction of invariants for a tool such as ATUSA to check during its exploration.

Security Enforcement: The invariants learned by DoDOM can also be used to check if a web page has been tampered with either during transmission or rendering at the

Training	**Java PetStore**				**CNN**			
Size	*NumNodes*	*NumChildren*	*MaxLevels*	*Descendants*	*NumNodes*	*NumChildren*	*MaxLevels*	*Descendants*
2	397	3.04	2.81	44.29	2413	2.470	2.632	48.485
4	397	3.04	2.94	44.29	2408	2.475	2.636	48.548
6	387	3.10	2.94	45.57	2407	2.475	2.636	48.548
8	387	3.10	2.94	45.57	2407	2.475	2.636	48.548
10	387	3.10	2.94	45.57	2407	2.475	2.636	48.548

Table 7. Results for Java PetStore and CNN applications

client. This is similar to the Web Tripwire project [33], 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 (for example, through a Type 2 XSS attack [13]).

Better Domain Filtering: Section 5.5 shows that failures of the majority of domains do not impact the invariant DOM for Slashdot. We believe this is also likely to be the case for many web applications that include multiple domains. We could filter such domains at the client for advertisement blocking and performance optimization. The NoScript plugin [24] already allows domain filtering but leaves it to the user to decide which domains to block. With DoDOM, we can automate the decision making process based on whether the domain impacts the DOM.

7 Related Work

Dynamic invariant derivation: DAIKON [14] and DIDUCE [18] derive program invariants based on dynamic executions. DAIKON derives invariants over multiple test inputs while DIDUCE does the same over multiple stages of a program's execution. These techniques attempt to learn invariants over program structures, and do not apply to Web 2.0 applications as these applications are typically data centric [27]. Hence, web applications require techniques that infer data-centric invariants.

HeapMD [7] infers invariants over data-structures on the heap and identifies significant violations of the properties as potential bugs. DoDOM also falls under the category of data-invariant detection techniques. However, it infers invariants over the web page's DOM, which are different from data-structure invariants in three aspects. First, data structure invariants typically encode the connectivity properties of data structures, while DOM invariants encode both the content and structure of the DOM. Second, data structure invariants are predicates over the nodes in the data structure, while DOM invariants are parts of the DOM tree. Finally, DOM invariants evolve with respect to specific events and actions in the application, while data structure invariants are typically fixed for the duration of the program.

Web application testing: A number of approaches have been developed to test web applications that execute primarily at the server, i.e., Web 1.0 applications [3, 34]. An example of this approach is Veriweb [3], 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. [25] propose an approach to test Web 2.0 applications using an abstract state machine model provided by the developer. Mesbah and Deursen [26] extend this work to infer the state machine model automatically by finding clickable elements in the application and emulating clicks on them. Similar to our work, they use invariants on the DOM tree to check the validity of a state explored by their tool called ATUSA. ATUSA differs from DoDOM in two aspects. First, the invariants used in ATUSA correspond to generic invariants on the validity of the DOM (e.g., invalid HTML), and are not specific to the web application being tested[8]. Second, ATUSA uses heuristics to find clickable elements on the page, and explores paths corresponding to the clicks. However, this exploration may not model a real user-interaction sequence.

Recently, Bezemer et al. [4] present a dynamic approach for security testing of widgets used in Web 2.0 applications. Widgets are small pieces of JavaScript code and/or HTML that co-exist within a single web page. The main differences between this approach and ours are that (1) we can analyze the entire web application rather than only widgets and (2) we do not require an explicit security policy specification.

AjaxScope [22] is a tool for remotely monitoring web 2.0 applications and changing their behavior on the fly. Similar to our approach, AjaxScope interposes on the web application using a proxy server and rewrites the application's JavaScript code. AjaxScope can be used for performing drill-down performance analysis or for debugging of complex bugs such as memory leaks. However, the intrumentation is at the level of JavaScript code and cannot be easily extended for DOM elements. Furthermore, AjaxScope does not in and of itself derive invariants and it would be non-trivial to extend it to do so.

Static Analysis: Static analysis approaches can also

[8]ATUSA allows the programmer to specify application-specific invariants, but does not derive the invariants.

be used for detecting errors in web applications. However, Web 2.0 applications are written in languages such as JavaScript, which are difficult for static analysis approaches to handle. This is because the JavaScript language has features such as dynamic/loose types, on the fly code creation and lack of separation between code and data [12]. Static analysis approaches for analyzing JavaScript [8, 16, 17] ignore these features in the interest of accuracy. However, many real web sites extensively use these unsafe features [39] and hence cannot be statically analyzed.

Real World Studies: Kalyanakrishnan et al. [21] study the availability of popular websites from an end-user perspective. Their study focused on the network connectivity between the client and the server. Further, they focus on the availability of a web site rather than its reliability. Chen et al. [6] present Pinpoint, a tool to automatically determine the root causes of failures in large internet service infrastructures deployed on the J2EE platform. Pinpoint targets the backend of web services. Pertet and Narasimhan [30] study downtime incidents of web services to understand their root causes. The study was confined to server failures.

Fault injection into web applications: Reinecke et al. [32] use fault injection to evaluate the resilience of reliable messaging standards in web services. Vieira et al. [38] use fault injection to evaluate the robustness of web-services' infrastructure. These approaches target errors in communication protocols and server code respectively.

Huang et al. [19] evaluate the security of web applications using fault-injection experiments. The injected faults correspond to SQL injection attacks and cross-site scripting (XSS) attacks. Fonseca et al. [15] use vulnerability and attack injection to test the resilience of web applications to attacks. However, these approaches operate on the application's server-side code and cannot target client-side code. Further, they are not concerned with non-malicious that result in the application deviating from its correct behavior.

Finally, Bagchi et al. [1] and Automatic failure path inference(AFPI) [5] use fault injection to track dependencies in server applications. We use fault injection for resilience testing rather than dependency discovery.

Dynamic Checking: Web applications have also used dynamic invariant approaches for detecting security attacks. Swaddler [11] derives dynamic invariants on the server code of web applications written using the PHP language. Subsequently, Swaddler uses the inferred invariants to detect attacks that attempt to bypass the application's workflow and force the application into an inconsistent state. Swaddler's analysis and enforcement is implemented on the server side and hence cannot be used for Web 2.0 applications.

Blueprint [37] is an approach to enforce the integrity of a web application's DOM at the client in order to prevent script injection (XSS attacks). The server encodes the intended DOM of the application as part of the document sent to the client, which is then compared with the actual DOM generated by the client's HTML parser. A mismatch indicates that the web page has been injected with untrusted content, a classic XSS attack. Robertson and Vigna [35] enforce the integrity of a web application's DOM using static typing at the server end to prevent both XSS attacks and SQL injection attacks. Both approaches require apriori knowledge of which portions of an application's DOM tree are trusted, and this information may not always be known to the developer. Robertson and Vigna's approach provides a way to automate the generation of such specifications, but requires the developer to use their static type system.

Web tripwire [33] is a system to detect unintended modifications of web pages made in transit or at the client. The server inserts JavaScript code into the web page to compare the page's HTML source after it has been loaded at the client with the original page's HTML. A mismatch indicates that the web page has been tampered with, either in transit to the client (e.g., by a malicious router) or by the client's web browser (e.g., by a malicious browser plugin). However, web tripwire requires exact knowledge of the page's source at the client which is hand coded and inserted into the page. Therefore, it cannot be applied to generic web pages. Furthermore, Web tripwire cannot be used to detect malicious modifications to the page after the check has been performed or by dynamic events in Web 2.0 applications.

8 Conclusions

This paper presents a novel approach to enhance the reliability of Web 2.0 applications using DOM-based invariants. The approach dynamically derives invariants on web applications' DOMs and uses them to detect errors. We present *DoDOM*, an automated tool to extract DOM invariants over multiple executions of the application.

We show that (1) DOM invariants exist in real web applications, (2) can be learned using DoDOM within a reasonable number of executions, and (3) are useful in detecting failures of events and domains that impact the DOM.

As future work, we plan to (1) integrate DoDOM as a part of the web browser, (2) extract invariants across different user-interaction sequences, and (3) explore the use of invariants to detect security attacks on web applications.

Acknowledgements

We thank Trishul Chilimbi, Emre Kiciman, Benjamin Livshits, and Nikhil Swamy for their insightful comments on this paper.

References

[1] S. Bagchi, G. Kar, and J. Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *Proc. 12th Intl. Workshop on Distributed Systems: Operations & Management*, 2001.

[2] W.I.C. Be. Rich Internet Applications.

[3] M. Benedikt, J. Freire, and P. Godefroid. VeriWeb: Automatically testing dynamic web sites. In *11th International World Wide Web Conference (WWW)*. ACM, 2002.

[4] Cor-Paul Bezemer, Ali Mesbah, and Arie van Deursen. Automated security testing of web widget interactions. In *Foundations of Software Engineering Symposium (FSE)*, pages 81–90. ACM, 2009.

[5] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *3rd IEEE Workshop on Internet Applications (WIAPP)*, 2003.

[6] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. *International Conference on Dependable Systems and Networks (DSN)*, 0:595, 2002.

[7] T.M. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. *ACM SIGPLAN Notices*, 41(11):228, 2006.

[8] R. Chugh, J.A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62. ACM, 2009.

[9] Microsoft Corporation. Fiddler: Web debugging proxy.

[10] Sun Microsystems Corporation. Java Petstore 2.0.

[11] 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*, 4637:63, 2007.

[12] D. Crockford. *JavaScript: the good parts*. O'Reilly Media, Inc., 2008.

[13] S. Di Paola and G. Fedon. Subverting AJAX. In *23rd Chaos Communication Congress*, 2006.

[14] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *22nd international conference on Software engineering(ICSE)*, pages 449–458, New York, NY, USA, 2000. ACM.

[15] J. Fonseca, M. Vieira, and H. Madeira. Vulnerability and attack injection for web applications. In *IEEE/IFIP International Conference on Dependable Systems and Networks(DSN)*, pages 93–102, 2009.

[16] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Usenix Security Symposium*. Usenix Association, 2009.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for AJAX intrusion detection. In *18th international conference on World wide web*, pages 561–570. ACM, 2009.

[18] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, 2002.

[19] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *13th international conference on World Wide Web(WWW)*, pages 40–52. ACM, 2004.

[20] D. Ingalls. The Lively Kernel: just for fun, let's take JavaScript seriously. In *Proceedings of the 2008 symposium on Dynamic languages*, page 9. ACM, 2008.

[21] M. Kalyanakrishnan, R. K. Iyer, and J. Patel. Reliability of internet hosts - a case study from the end user's perspective. In *6th International Conference on Computer Communications and Networks*, page 418, Washington, DC, USA, 1997. IEEE Computer Society.

[22] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. pages 17–30, 2007.

[23] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) level 3 core specification. *W3C Recommendation*, 2004.

[24] G. Maone. Noscript-whitelist JavaScript blocking for a safer Firefox experienc e.

[25] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of Ajax web applications. In *IEEE International Conference on Software Testing Verification and Validation (ICST)*, 2008.

[26] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *IEEE 31st International Conference on Software Engineering (ICSE)*, pages 210–220. IEEE, 2009.

[27] M. Ohara, P.N.Y. Ueda, and K. Ishizaki. The Data-centricity of Web 2.0 Workloads and its Impact on Server Performance. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 133–142, 2009.

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

[29] pc facile.com. Phoenix 1.6.0.

[30] S. Pertet and P. Narasimhan. Causes of failure in web applications. *Parallel Data Laboratory, Carnegie Mellon University, CMU-PDL-05-109*, 2005.

[31] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn. Jsmeter: Characterizing real-world behavior of JavaScript programs. *Technical Report, MSR-TR-2009-173*, 2009.

[32] P. Reinecke, A.P.A. van Moorsel, and K. Wolter. The fast and the fair: A fault-injection-driven comparison of restart Oracles for reliable Web services. In *QEST*, volume 6, pages 375–384. ACM, 2005.

[33] C. Reis, S.D. Gribble, T. Kohno, and N.C. Weaver. Detecting in-flight page changes with web tripwires. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 31–44. USENIX Association, 2008.

[34] F. Ricca and P. Tonella. Analysis and testing of web applications. In *International Conference on Software Engineering*, volume 23, pages 25–36, 2001.

[35] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Usenix Security Symposium*. Usenix Association, 2009.

[36] J. Ruderman. The Same Origin Policy.

[37] M. Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *30th IEEE Symposium on Security and Privacy (Oakland)*, pages 331–346, 2009.

[38] M. Vieira, N. Laranjeiro, and H. Madeira. Benchmarking the Robustness of Web Services. In *The 13th IEEE Pacific Rim Dependable Computing Conference (PRDC 2007), Melbourne, Victoria, Australia*, 2007.

[39] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *18th international conference on World wide web*, pages 961–970. ACM, 2009.