Harnessing Web-based Application Similarities to Aid in Regression Testing

Kinga Dobolyi and Westley Weimer

Department of Computer Science
University of Virginia, Charlottesville, VA. 22904
Email: {dobolyi,weimer}@virginia.edu

Abstract

Web-based applications are growing in complexity and criticality, increasing the need for their precise validation. Regression testing is an established approach for providing information about the quality of an application in the face of recurring updates that dominate the web. We present techniques to address a key challenge of the automated regression testing of web-based applications. Innocuous program evolutions often appear to fail tests and must be manually inspected. We rely on inherent similarities between independent web-based applications to provide fully automated solutions for reducing the number of false positives associated with regression testing such applications, simultaneously focusing on returning all true positives.

Our approach predicts which test cases merit human inspection by applying a model derived from regression testing other programs. We are 2.5 to 50 times as accurate as current industrial practice, but require no user annotations.

I. Introduction

Despite the ubiquitous use of web applications, most are not developed according to a formal process model [22]. Web applications are subject to high levels of complexity and pressure to change. This manifests in short delivery times, emerging user needs, and frequent developer turnover, among other challenges [24]. Consequently, systems are delivered without being tested [24], potentially resulting in functionality losses on the order of millions of dollars per hour [21], [32], [37]. User-visible failures are endemic to top-performing web applications: about 70% of such sites are subject to user-visible failures, a majority of which could have been prevented through earlier detection [27]. One way to avoid preventable monetary losses is then to design web-based applications to meet high reliability, usability, security, and availability requirements [20], which translates into well-designed and well-tested software that is as free from error as possible.

This research was supported in part by, but may not reflect the positions of, National Science Foundation Grants CNS 0716478 and CNS 0905373, Air Force Office of Scientific Research grant FA9550-07-1-0532, and NASA grant NAS1-02117, and Microsoft Research gifts.

Regression testing is an established approach for increasing the reliability of applications in the face of recurring updates, and testing costs sum to as much as $35 billion annually in the United States [12], [25], [33]. Unfortunately, testing of web applications is often perceived as lacking a significant payoff [11]. A general want of resources [10], [17], [39], combined with the additional complexities of web applications [24], makes automation a necessity if regression testing is to be adopted in the web-based application domain. Although automated replay of existing test suites is relatively straightforward, regression testing is constrained by the effort required for the comparison of test results between two program versions.

In this paper, we propose to harness the inherent similarities between web-based applications to reduce the cost of regression testing them. We hypothesize that errors in web-based applications can be successfully modeled due to the tree-structured nature of XML/HTML output, and that unrelated web-based applications fail in similar ways. We employ these similarities to provide a precise, semantics-based automated oracle comparator for the regression testing of web-based applications.

Our goal is to create an *automated* oracle comparator that relies on the *semantics* of HTML (or XML) output to decide if a pair of test case outputs indicates an error. An *oracle* produces an expected result and a *comparator* validates the actual test case output against the expected result [4]. Existing comparators for web applications typ-

978-0-7695-3878-5/09 $26.00 © 2009 IEEE
DOI 10.1109/ISSRE.2009.18

ically have false positive rates in the 4% range [31][1], but must be trained on the target application to achieve this value, while a `diff`-based approach to predicting faulty output is wrong 70–90% of the time in our experiments. Such training requirements and imprecision preclude the automation of the regression testing process for web-based applications. This is a lamentable consequence in the web-based application world, where testing is often sacrificed due to resource constraints and time-to-market pressures [24]. In addition, while comparators have the potential to significantly reduce the number of false positives over `diff`, they may introduce false negatives.

A *precise comparator* is one that reduces the number of false positives when compared to a naïve approach that reports any difference between two outputs. Our goal is provide a precise comparator that minimizes the number of false positives without failing to report true positives. In previous work we presented such a precise comparator, but it required manual annotation and human training, and was thus not truly automatic [29]. In this paper we propose a technique that dispenses with the need for manual annotation yet performs better overall. We use other web-based application output, unrelated to the application-at-test, to "train" such a comparator to recognize error situations. Though our corpus of training data is not related to the application we are testing, the types of and manifestations of faults in such applications are often similar in nature, allowing us to build a general predictive fault model. By offering this set of training data to our model and other users, we are able to automate the comparator process by not relying on any additional form of manual annotation or manual fault seeding.

Our approach focuses not only on the similar ways unrelated web applications fail, but also on the equally important ways in which they tend to benignly evolve. Ignoring harmless program evolutions is central to reducing the number of false positives associated with a comparator. Conversely, correctly modeling true erroneous output allows our model to minimize false negatives.

To highlight the challenges of using such an approach, consider a new version of a web-based application that produces otherwise-identical HTML output, but with a different footer that may or may not include dynamically generated elements such as a timestamp. While comparators based on standard `diff` or `diff`-like tools will not return any false negatives, a `diff` of the old output and the new output will always report a potential error, even if no new defect has been introduced. The problem is compounded with each new version of the software. We construct an automated oracle comparator that does not report harmless updates to the application, while still flagging actual bugs.

[1]These false positive results were obtained when testing on a single version of the software using seeded faults. In this paper we focus on regression testing to find faults between different versions of applications. In such settings the number of false positives reported by such tools increases due to natural, innocuous program evolutions.

We present and evaluate an automated oracle comparator for applications with XML/HTML output. Our comparator relies on web page similarities between unrelated web-based applications and needs no manual training by developers. We distinguish between *web applications* and *web-based applications* in that the latter, in addition to producing HTML, may also produce XML output. Consequently our work focuses on web-based applications, as we are able to handle both XML and HTML output.

The structure of this paper is as follows. In Section II we present some examples of test case output where `diff` would produce too many false positives. Section III presents related work, and Section IV gives an overview of our approach. Our evaluational setup is described in Section V, with experimental results following in Section VI. Finally, Section VII presents conclusions and directions for future work.

II. Motivating Examples

Modern functional testing of web applications (as opposed to HTML validation, link testing, or load testing) relies primarily on capture-replay infrastructure where tester input sequences are recorded and replayed [8]. In such situations, the oracle is often HTML output of a previous, trusted version of the application, and comparison is accomplished through a `diff` of two outputs. Unfortunately, the human interpretation of test results is a "tedious and error-prone task" [32] and is decidedly onerous for web applications because of the frequent false positives generated by a `diff`-based comparator.

Fortunately, updates to web-based applications often happen in similar and predictable ways. Consider the following example from a `diff` of two GCC-XML test case outputs [36] (the text above the dashed line was generated by the older application, while the rest is output from the newer version).

```
<   <Namespace id="_2" name="std" context="_1
    " members=""/>
<   <Function id="_3" name="foo" returns="_4"
     context="_1" location="f0:8">
---
>   <Namespace id="_2" name="std" context="_1
    " members="" mangled="_Z3std"/>
>   <Function id="_3" name="foo" returns="_4"
     context="_1" mangled="_Z3fooii" location
    ="f0:8" file="f0" line="8" endline="15">
```

Notice that in both versions the same `<Namespace>` and `<Function>` elements are being defined. Each attribute that exists in the older version also appears in the newer version, and with the same value. The main difference is that the newer version of the application contains new functionality in the form of additional attributes, such as `mangled="_Z3fooii"` on line 5. Relying on a `diff`-like comparator for regression testing these two outputs would lead to a false positive. Web-based applications frequently evolve through the addition of new attributes to existing elements, as one example of a typical change that should generally not indicate an error in regression testing.

Another example is an update to HTML code that does not change the appearance or functionality experienced by the end user. Consider the `diff` output from two TXT2HTML test case versions [34] (the newer output is below the dashed line):

```
< <P>The same table could be indented.
< <TABLE border="1">
---
> <p>The same table could be indented.</p>
> <table border="1" summary="">
```

As in the previous example, the `<table>` element contains a new attribute (`summary`). The newer output also matches the paragraph tag `<p>` with a closing tag, presumably because future versions of HTML will not support unmatched tags, although most browsers will display these two bits of code equivalently. Again, a `diff` comparison of these two outputs would yield a false positive due to the additional `summary` attribute, the closing paragraph tag, and the difference in case between the two `table` strings.

Using `diff`-like comparators for the above examples, as well as for other outputs involving small natural language changes, formatting updates, or changes in the order of elements or attributes, would yield a high number of false positives because the human-level interpretation of the result remains unchanged. Simply ignoring certain types of updates to websites, however, raises the possibility of missing actual bugs. Consequently, we present an automated technique that reduces the number of false positives in test output comparison while minimizing the number of false negatives by learning characteristics of faults and non-faults.

III. Related Work

Oracle comparators are frequently used for web testing, where human intervention is required in the presence of discrepancies [8], [16], [24], [32]. Lucca *et al.* mention a comparator that automatically compares the actual results against the expected values of the test execution [16]. Our approach can be considered a working instantiation of their outline, extending the concept of syntactic differences to semantic ones.

Providing a precise comparator for web-based applications remains an open research area. Sprenkle *et al.* and Sampath *et al.* have focused on oracle comparators for testing web applications [30], [31], [32]. They investigate features derived from `diff`, content, and structure, and refine these features into oracle comparators [32] based on HTML tags, unordered links, tag names, attributes, forms, the document, and content. Applying decision tree learning allows them to identify the best combination of oracle comparators for a specific application in [31]. Our approach also combines machine learning and automated oracle comparators, though our features and benchmarks are not always HTML-specific and can be more generally applied.

A more important difference between our approaches is that they suggest developers hand-seed faults from bug reports to create faulty versions of code, from which outputs can be collected and used as training data [31]. Our method does not require manual seeding of faults and uses training data from other applications to automate this process. Additionally, rather than introducing multiple oracles targeted at different hypothetical types of web applications [31], [32], our model uses features that we claim are closer to tree-based differences and human judgments in a holistic manner to train one generic oracle-comparator that can be tailored to the application at test automatically. Finally, their approach is validated by measuring their oracles' abilities to reveal seeded faults in one version of an application (i.e., measuring differences between the clean application and a fault-seeded one). By contrast, our experiments train and test on data between *different versions* of the same application. Our approach contends directly with common and benign program evolutions, in contrast to the setting of Sprenkle *et al.* [31], where a `diff` comparator would have no false positives for a deterministic application.

Although recent work has explored using abstract syntax tree matching [19] and semantic graph differencing [23] for analyzing source code evolution, such approaches are not helpful when comparing XML and HTML text outputs. Not only do they depend on the presence of source code constructs such as variables and functions (which are not present in generic HTML or XML) to make distinctions, but they are meant to summarize changes, rather than to decide if a pair of test case outputs indicate an error. There is currently no industry standard for comparing pairs of XML/HTML documents beyond that of `diff` used in capture-replay contexts and user-session based testing [13]. Developers have the option of customizing `diff`-like comparators for their target applications, such as by using regular expressions to filter out conflicting dates, but these tools must be manually configured for each application and potentially each test case, and may not be robust as the website evolves.

Binkley [5], [6] as well as Vokolos and Frankl [38] approach regression testing by characterizing the semantic differences between two versions of program *source code* using program slicing. By doing so, only program differences need to be tested and the total number of test cases that need to be executed between versions are reduced. Our approach focuses on the semantic different between two versions of the program *output*; our technique is orthogonal to theirs, and our tool can be used in a retest-all framework or in conjunction with theirs in a setting in which some regression tests have been skipped due to source code similarity.

Sneed explores a case study on testing a web application system for the Austrian Chamber of Commerce [28]. A capture-replay tool was used to record the dialog tests, and XML documents produced by the server were compared at the element level: if the elements did not match, the test failed. Our approach also compares XML documents, but does not necessarily rely on exact element matching, and thus reports fewer false positives.

Capture-replay scripts suffer from the "fragile test problem", where a robot user fails for trivial reasons [18]. Meszaros outlines the two parts of this problem: interface sensitivity, where "seemingly minor changes to the interface can cause tests to fail even though a human user would say the test should still pass", and context sensitivity, such as to the date of the given test suite [18]. Our approach prevents the fragile test problem in many instances, reducing the number of false positives and allowing for older test case outputs to be reused when comparing to newer versions.

IV. Modeling Application Similarities

The main hypothesis of our work is that similarities between unrelated web-based applications can be used to reduce the burden on developers during regression testing within the software life cycle. In the following section, we summarize our approach towards modeling differences between pairs of XML/HTML documents [29].

A. Modeling HTML Differences

Due to the tree-structured nature of XML/HTML, using a `diff`-like comparator for test case outputs has the potential for a high number of false positives. Unlike flat text files, trees are well-formed objects with a directed edge relationship. Certain XML/HTML characteristics, such as the different ordering of element attributes, are recognized as changes by `diff`, but are semantically unimportant when the XML or HTML is later interpreted. By examining the tree representations of two versions of output, our tool can make decisions about the relative importance of changes in terms of semantic significance.

We claim that comparator judgments about XML/HTML test case output should be made by inspecting combinations of tree-structure features, rather than by using textual `diff`. We represent the two output documents as trees, and then find an *alignment* between them such that a minimal number of changes is required to transform one tree into the other. To do so, we rely on an algorithm to calculate structural differences between XML documents. One such algorithm is DIFFX [3], which computes alignments on general tree-structured data by matching pairs of elements between the newer and older trees. Once pairs of tree elements are aligned, features can be calculated. A feature is a measure of the similarity of a characteristic between two XML/HTML outputs. Our goal is to delineate features that have a significant semantic meaning — in other words, to find visible characteristics of XML/HTML output that are likely to indicate errors between the two documents. Some of our features were conceived during the manual examination of test case output pairs, while others were obtained from first principles. Our features fall into two main categories: those that measure differences between the tree-like structure of the two outputs, and those that directly characterize human-judged differences between two documents. Features may be correlated positively or negatively with test output errors, depending on the application being examined. Most of our features are relatively simple, and we summarize them below:

- The number of inserts, deletes, and moves required to transform one tree into the other, as dictated by the DIFFX algorithm. These features are likely to be correlated with errors such as stack traces or missing page elements due to the large number of shifted or replaced elements between the two trees.
- The number of element inversions (e.g., `<b><u>The</u></b>` to `<u><b>The</b></u>`). We hypothesize that inversions do not indicate high-level errors. Such changes are not likely to be rendered differently by the browser.
- Grouped changes to a set of contiguous elements in the tree. Such changes are likely to represent errors, such as a missing component.
- The maximum tree depth of changes. We hypothesize that changes closer to the root of the tree are likely to merit inspection.
- Changes to only text nodes. We hypothesize that text changes are not likely to represent faults. This is one of the features that allows our approach to outperform `diff`-like comparators, as the ability to ignore natural-language text changes is integral to reducing the false positives associated with more naïve approaches.
- Changes to child ordering. Child reorderings are not predicted to indicate errors, as they most likely do not

change the semantics of the rendered webpage.

- The ratio of displayed text, and the ratio of displayed text to multimedia, between two versions. A sudden increase in the amount of natural language text may indicate a stack trace or other error.
- The number of programming-language based error keywords (i.e. "exception" or "error") that occur in the newer version but not the older. We hypothesize that such keywords are highly likely to signal a defect. Because web applications are built using similar programming languages, and these languages contain analogous natural language for exceptional situations, searching for common error keywords in the newer output is a reasonable way to predict errors.
- Changes to functional elements, such as buttons and forms. Such changes may signal errors, in the case of a missing button, or may be a part of the natural evolution of websites as new functionality is added.
- Changed attribute values of an element. We claim that changed attribute values are unlikely to be associated with errors in a web application, as opposed to missing attributes, which we hypothesize can be linked to defects.

We assign each feature a numeric weight that measures its relative importance. Our model indicates that a pair of test case outputs should be manually inspected whenever the weighted sum of all feature values for that pair of test case outputs exceeds a certain cutoff value. The weights and cutoff value can be learned empirically; we return to this issue when discussing our experimental setup (see Section V-A).

The key task of such a model is: given the oracle output for a test case and the current output for that same test case, indicate whether an error should be flagged and the situation evaluated by a human. False positives yield wasted developer effort as humans fruitlessly inspect correct output, while false negatives cause the testing process to miss bugs. We use precision and recall, metrics from the domain of information retrieval [26], to measure our model's success at this task:

$$\text{recall} = |Desired \cap Returned| \div |Desired|$$

$$\text{precision} = |Desired \cap Returned| \div |Returned|$$

Here *Desired* refers to the test cases our model labeled as errors that were actually errors — in other words, the desired true positives associated with our model. *Returned* represents all the test cases flagged as errors by our model, which includes both true positive test cases as well as unwanted false positive test cases that were wrongly flagged. *Recall* is the ratio of desired error cases our models returns to the total number of desired error values; that is, how close are we to finding all the desired error cases. A low recall value indicates that our model is missing too many actual errors (i.e., has too many false negatives). *Precision* refers to the number of true positives our model returns as a fraction of the total number of values returned — in other words, what fraction of our model's output is correct. A low precision value implies that our model cannot successfully distinguish between erroneous and non-erroneous output (i.e., has too many false positives). Because precision can be trivially maximized by returning only a single error, and recall can be similarly maximized by returning all test case pairs as errors, we combine precision and recall by taking their harmonic mean: $2pr \div (p + r)$. The result is called the F_1-*score*, and gives equal weight to the two variables [7].

V. Experimental Setup

We experimentally tested our hypothesis that web site similarities can be exploited to aid in the automation of various aspects of testing web-based applications. This section introduces our experimental setup; Section VI presents the results. We measure the predictive power of our precise oracle-comparator with respect to finding faults in test case output pairs (Section V-A). Our hypothesis is that we can train a model on one set of data, and test on a separate, unrelated application, using the underlying similarities between these two sets of output.

A. Experimental Setup: Modeling Meaningful Differences

Since our approach for modeling output differences involves supervised learning, we must first train our model before we can test it. Recall that our model takes a weighted sum of feature values for a pair of test case outputs and indicates that they should be inspected if the sum exceeds a certain cutoff; we must determine the weights and the cutoff. In our experiments, the feature weights and cutoff are learned on a per-project basis using linear regression. The data used to train the linear regression model is a corpus of previously-annotated test output pairs for other, unrelated web-based applications. We achieve savings over other approaches because we provide this annotated data-set to the user of the model, rather than intending them to produce their own annotations.

Our goal is to use our model as a classifier that partitions the set of test case output pairs into "should inspect" and "can ignore" groups. To do so, we train the linear model as if the response variable were within the continuous range of [0,1], where a cutoff represents the transformation of the real-valued model into a binary classifier. The cutoff is calculated through a linear search that finds the highest F_1-score.

Training our model requires a set of test case output pairs with known labels (i.e., annotations indicating whether the pair should be inspected or not). One option is to annotate a subset of test cases from the application-at-test to be used as training data, but this has the disadvantage of requiring human effort. Instead, we use pairs of test case output from *unrelated*, publicly-available applications as training data for our model. This has the advantage of not requiring new manual annotations of test case output, with the potential drawback of not being as effective as training data tailored to the application-at-test. Our experimental results in Section VI-A show, however, that we are able to achieve very high levels of accuracy using this approach, due to the underlying similarities between web-based applications and their failure modes.

Figure 1 describes the XML/HTML applications from which we obtain a corpus of test case output pairs to train our model. Our benchmarks were selected based on the availability of multiple versions and test cases, with the goal of application diversity in mind. In total we exercised over 473,000 lines of code to produce 7154 pairs of test case output as our training data set.

As an option, a developer using our approach may also add their own test cases from previous projects to the set of training data, assuming that they have already annotated those output pairs, but in general we do not require any developer annotations.

We manually inspected the test case output pairs for the two versions of each benchmark, arriving at 919 pairs that were labeled as errors. In this case, an error is defined as "possibly a bug, merits human inspection", as opposed to a passed test case which translates into "definitely not a bug". Intuitively, we flagged situations where the acceptability of the output was in question, based on either a functional or a visual difference for the user. Our annotations erred on the side of conservatism: we only passed test cases when we were highly certain they did *not* signal an error. Consequently, it is possible we labeled non-errors as errors, which reduces our opportunity to outperform a `diff`-like baseline comparator, but does not impact the correctness of our approach.

We selected four benchmarks, shown in Figure 2, to serve as our test data, and used the applications in Figure 1 as our training data. Although we used ten benchmarks as our training corpus, only two of them (HTMLTIDY and GCC-XML) had enough test case output pairs that were labeled as faults (given by the "Test Cases to Inspect" column) to serve as *testing* (as opposed to training) subjects. We also chose two open source web applications (CLICK and VQWIKI) to supplement our test benchmarks in a "worst-case scenario" fashion: none of the training benchmarks are web applications, so successful performance on them further supports our claim about inherent application similarities.

VQWIKI [2] is wiki server software that can be used out-of-the-box as a web application. CLICK [1] is a Java Enterprise Edition web application framework that ships with a sample web application demonstrating the framework's features. Our other two testing benchmarks, HTMLTIDY and GCC-XML, are open-source XML-based applications that are also a part of our training benchmarks. For these two applications, we removed each benchmark's respective test case outputs from the corpus of training data, so that we never tested and trained on the same data. Therefore, the training data for CLICK and VQWIKI were the test case output pairs from the ten benchmarks in Figure 1, while GCC-XML and HTMLTIDY were trained with the nine remaining benchmarks from Figure 1, when the test cases for each respective application were removed from the training data set — in no case did we test and train on the same data. In total we tested our model on 6728 test case pairs, 941 of which were labeled as errors by manual inspection (see Figure 2).

VI. Experimental Results

This section presents empirical measurements of our model's predictive power at detecting faults between test case output pairs (see Section VI-A).

A. Experiment 1: Results

Figure 3 shows our model's F_1-score values for each test benchmark. An score of 1 indicates perfect performance. We also include F_1-score values for unbiased and biased coin toss, standard `diff`, and `xmldiff` [35], an off-the-shelf `diff`-like comparator for XML and HTML. `xmldiff` is able to ignore features such as whitespace, namespaces, and case in text elements when comparing two XML/HTML files. The unbiased coin toss returns "inspect" with a probability of 0.5, while the biased coin toss returns "inspect" with the dataset's actual underlying ratio: $(6728 - 941)/6728$ (note that it is not possible to know this ratio *a priori* in the field). We chose `diff` and `xmldiff` instead of the *Struct* comparator of Sprenkle *et al.* [30] to avoid false negatives.

Our tool is anywhere from over 2.5 to almost 50 times as good as `diff` at correctly labeling test case outputs, with similar improvements over `xmldiff`. For the two web applications, we achieve perfect precision and recall — an optimal result. Our scores for our large XML benchmark, HTMLTIDY, are also close to perfect (an F_1-score of 0.98). Overall, we judge that using test case output pairs from unrelated web-based applications

Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
LIBXML2	v2.3.5 v2.3.10	84K	XML parser	441	0
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
CODE2WEB	v1.0 v1.1	23K	pretty printer	3	3
DOCBOOK	v1.72 v1.74	182K	document creation	7	5
FREEMARKER	v2.3.11 v2.3.13	69K	template engine	42	1
JSPPP	v0.5a v0.5.1a	10K	pretty printer	25	0
TEXT2HTML	v2.23 v2.51	6K	text converter	23	6
TXT2TAGS	v2.3 v2.4	26K	text converter	94	4
UMT	v0.8 v0.98	15K	UML transformations	6	0
Total		473K		7154	919

Fig. 1. The benchmarks used as **training data** for Experiment 1. The "Test cases" column gives the number of regression tests used; the "Test cases to Inspect" column counts those tests for which our manual inspection indicated a possible bug.

Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
VQWIKI	2.8-beta 2.8-RC1	39K	wiki web application	135	34
CLICK	1.5-RC2 1.5-RC3	11K	JEE web application	80	7
Total		108K		6728	941

Fig. 2. The benchmarks used as **test data** for Experiment 1. The "Test cases" column gives the number of regression tests used; the "Test cases to Inspect" column counts those tests for which our manual inspection indicated a possible bug. When testing on HTMLTIDY or GCC-XML, we remove it from the training set.

to train a model to predict errors in the application-at-test is a successful approach. We claim that the underlying similarities between web-based applications in general make this possible. We performed an analysis of variance; features associated with text-only changes were strongly negatively associated with errors in most benchmarks. By employing an available model and training set such as ours, developers would be able to significantly reduce the number of false positive test case output pairs they must inspect, without requiring annotations or additional human effort to train the model.

Figure 4 shows our model's precision scores for each benchmark, as well as our baseline comparators, highlighting our predictive power over `diff`-like comparators. Figure 5 presents our model's recall scores, where we are challenged by `diff` in that the latter will always be able to return all true positive errors. For the two web applications (VQWIKI and CLICK), we are equally as good as `diff` at returning error cases, while for HTMLTIDY our score is competitive.

We can estimate the savings using our approach by defining the cost of looking at a test case (*LookCost*) and the cost of missing a bug (*MissCost*). Our approach is advantageous when its associated cost:

$$(\mathit{TruePos}+\mathit{FalsePos})\times \mathit{LookCost}+\mathit{FalseNeg}\times \mathit{MissCost}$$

is less than the cost of current industrial practice of $|\texttt{diff}| \times \mathit{LookCost}$. We are able to save developers effort when the cost of examining false positives flagged by `diff`, but not our technique, is greater than the cost of missing any relevant test cases with our tool:

$$\frac{\mathit{LookCost}}{\mathit{MissCost}} > \frac{-\,\mathit{FalseNeg}}{\mathit{TruePos}+\mathit{FalsePos}-|\texttt{diff}|}$$

We assume *LookCost* $\ll$ *MissCost*, so we aim to minimize this ratio. For our two web applications, our perfect F_1-scores imply we always produce savings with respect to `diff`: 75% and 96% of the test case pairs reported as errors by `diff` were false positives for CLICK and VQWIKI respectively, and we eliminate the need to check any of these while, at the same time, correctly flagging all potential errors. For HTMLTIDY, we achieve savings over `diff` if the ratio of *LookCost* to *MissCost* is at least 0.0004 (in other words, if the cost of missing a bug is no more than 2500 times the cost of looking at a report). This is significantly better than the 0.0015 ratio of previous, non-automatic work [29].

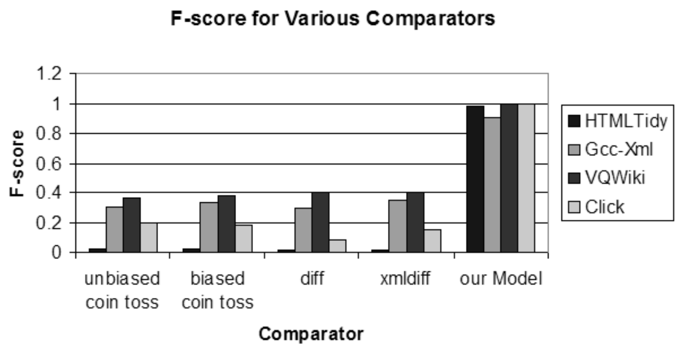

Fig. 3. F_1-score on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK using our Model, and other baseline comparators. 1.0 is a perfect score: no false positives or false negatives.

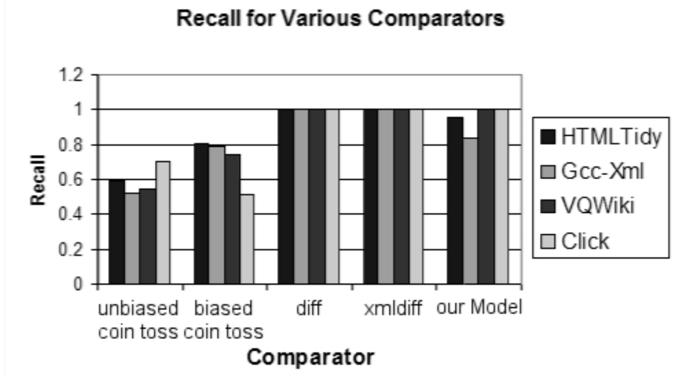

Fig. 4. Recall on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK using our Model, and other baseline comparators.

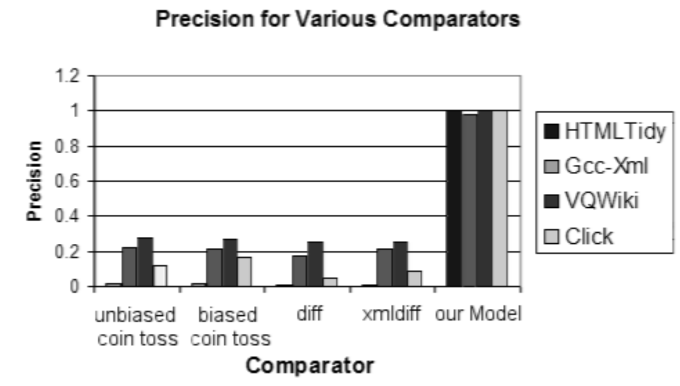

Fig. 5. Precision on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK using our Model, and other baseline comparators.

While our F_1-score for our other XML benchmark, GCC-XML, was three times better than that of `diff`, its recall score of 0.84 implies that we may be missing a significant number of actual errors. Our analysis of variance revealed that GCC-XML relied heavily on deletions being positively associated with errors, while in GCC-XML's training data the opposite was the case. Rather than recommend that developers return to using a `diff`-like comparator to avoid missing bugs, or employ other methods where manual annotation is required, we suggest they continue to apply our approach with one modification: they should extend the training data with test case output pairs between unmodified source code executions and fault-injected source code executions. A cautious development organization might randomly spot-check 10% of the results predicted by our technique. Such a spot-check would still involve less effort than a standard `diff` comparator. If the results are insufficiently accurate, the test suite can be augmented by defect seeding, as described in the next subsection.

B. Training Data from Defect Seeding

In this subsection we detail our experience with using defect seeding to generate additional training data for GCC-XML. Defect seeding offers the benefit of annotation-free training data generation, while still tailoring the training data to the current application under test.

Our relatively low recall value for GCC-XML suggests that the application-at-test may exhibit some errors that are different from the instances of errors in the general training data set we provide. Given the semantic difference between GCC-XML and the rest of our training applications, this is not surprising. We claim that developers can automatically tailor the training set to their application as needed using defect seeding.

The basic approach is to seed the source code of the application with defects [14] and run the resulting mutated program on its existing regression test suite. Any difference in the output can be attributed to the injected fault, and that output pair can be added to the training data with the label "should inspect". The process is repeated until a sufficient number of training instances have been generated. Using defect seeding or mutation to simulate errors in test case output for web-based applications has previously been explored [15], [31]. While automatically generating, compiling and running mutants can be CPU-intensive, it does not require manual intervention.

We implemented defect seeding for GCC-XML with a subset of mutation operators described by Ellims *et al.* [9]. Examples of mutation operators include deleting a line of code, replacing a statement with a return, or changing a binary operator, such as swapping `AND` for `OR`. To generate a mutant version of GCC-XML, we randomly choose a single line of source code from all of the source code files for that project and apply a mutation to it. For each mutant version of the program, only a single line was mutated. We compiled each mutant version of the source code separately, and re-ran the test suite, recording as erroneous any cases where the output from the mutant source code differed from that of the original output. The overall process was quite rapid: using single-line seeded faults we were able to obtain 11,000 usable erroneous output pairs within 90 minutes on a 3 GHz Intel Xeon computer.

Figure 6 shows our F_1-scores when adding between 0 and 5 defect-seeded output pairs to the set of training data. Selecting 0 mutants is provided as a baseline. The large margin of error when adding only one mutant output pair implies that performance depends on selecting the most useful mutant outputs to include as a part of the training data set. However, we note that selecting any mutant output is always better than selecting none. We hypothesize that it is possible to dramatically affect our model's predictive power by adding a single mutant because for the case of GCC-XML, there were only 44 errors in the training data set, and adding one more to such a small number can significantly change the results. For training data sets that contained more errors, it is possible that more mutants will be required, although we have demonstrated that it is quite simple to automatically generate these defects.

In addition, no significant performance gains are witnessed beyond adding 5 mutant output pairs, at which point the F_1-score was an essentially-perfect 0.999. We conclude that very little application-specific training data (5 labeled output pairs) is needed to bring even our-worst performing benchmark up to almost-perfect performance, and we demonstrate that even that application-specific data can be obtained automatically.

C. Summary of Experiments

Inherent web site similarities are a promising way to reduce the burden of human effort in regression testing for web-based applications. In Section V-A we demonstrated that using test case output pairs from *unrelated* web applications to train a model to predict errors in output in the application-at-test is a viable strategy, achieving perfect recall and precision for our two web application benchmarks, while close to perfect (0.98 and 0.99) F_1-scores for our two XML-based applications. To obtain the F_1-score of 0.999 for GCC-XML, we augment the training data with five automatically generated outputs obtained via defect seeding. In all cases we outperform `diff`-like comparator by a factor between 2.5 and 50 times, thereby significantly reducing the number of false positives, and thus the developer cost, with respect to `diff`.

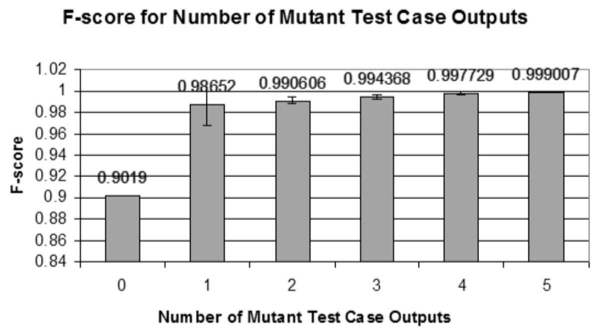

Fig. 6. F_1-score for GCC-XML using our model with different numbers of test case output pairs from original-mutant versions of the source code. The "0" column indicates no mutant test outputs were used as part of the training data. Each bar represents the average of 1000 random trails; error bars indicate the standard deviation.

D. Threats to Validity

Although we show significant savings in the amount of effort required to automate parts of the regression testing process, our results may not generalize to industry practice. It is possible that the benchmarks we selected to test on were not indicative of other applications. To mitigate this threat, we attempted to choose open-source benchmarks rather than toy applications, and select them from a variety of domains. Our combined benchmarks are over seven times larger than the combined benchmarks of the previous work that we are most closely related to [32] in terms of lines of code, and we have over twice as many total test cases. In addition, all of our benchmarks are freely-available open source applications.

In cases where our technique does not work as well as desired, our defect-seeding results suggest that largely-automatic improvement is possible. Adding mutant test case outputs to the set of training data for our precise comparator can help to tailor our model to the application-at-test, and the low number of mutants required implies that it may even be possible to provide a very small (≤ 5) set of manually-generated error instances to tailor our tool to a specific application.

It may also be possible that there are certain web applications for which we do poorly, despite defect seeding, because the specification of the application-at-test has unusual properties. For example, consider a Wiki application where the formatting and content of displayed natural language text *is* important. If fault seeding is unable to provide suitable defects on which to train our model to recognize small changes in natural language text as errors, we will be unable to use our approach. Although we did not explore such unusual cases, in future work we seek to examine such scenarios.

VII. Conclusion

Testing web-based applications is often overlooked due to a lack of time and resources, despite their high reliability requirements. Although automating test suite replay is relatively simple, comparing test results with expected output remains a challenge for this domain. We present a new technique that takes advantage of inherent similarities between web-based applications to automate parts of the regression testing process for this domain. Using a `diff`-like comparator for web-based output yields a significant number of false positives that must be manually inspected: instead, we offer a fully automated precise comparator that is based on a model trained on data from unrelated web-based applications. We evaluated our technique on 6728 test case pairs, and found that our approach outperforms the current industrial practice anywhere from 2.5 to 50 times, achieving perfect precision and recall half the time, and very close to perfect precision and recall otherwise.

References

[1] Apache Click, 2008. http://incubator.apache.org/click/.

[2] Vqwiki open source project, 2008. http://www.vqwiki.org/.

[3] Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[4] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *International Conference on Software Maintenance*, pages 41–50, 1992.

[6] David Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997.

[7] Tsong Yueh Chen, Fei-Ching Kuo, and Robert Merkel. On the statistical properties of the f-measure. In *International Conference on Quality Software*, pages 146–153, 2004.

[8] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *International Conference on Software Engineering*, pages 49–59, 2003.

[9] Michael Ellims, Darrel Ince, and Marian Petre. The csaw c mutation tool: Initial results. pages 185–192, 2007.

[10] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[11] Edward Hieatt and Robert Mee. Going faster: Testing the web application. *IEEE Software*, 19(2):60–65, 2002.

[12] G. M. Kapfhammer. Software testing. In *The Computer Science Handbook*, chapter 105, 2004. CRC Press.

[13] Srikanth Karre. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005.

[14] John C. Knight and Paul Ammann. An experimental evaluation of simple methods for seeding program errors. In *International Conference on Software Engineering*, pages 337–342, 1985.

[15] Suet Chun Lee Lee and Jeff Offutt. Generating test cases for xml-based web component interactions using mutation analysis. In *International Symposium on Software Reliability Engineering*, page 200, 2001.

[16] G. Di Lucca, A. Fasolino, F. Faralli, and U. de Carlini. Testing web applications. *International Conference on Software Maintenance*, page 310, 2002.

[17] Atif Memon, Ishan Banerjee, Nada Hashmi, and Adithya Nagarajan. DART: A framework for regression testing "nightly/daily builds" of GUI applications. In *International Conference on Software Maintenance*, 2003.

[18] Gerard Meszaros. Agile regression testing using record & playback. In *Object-oriented programming, systems, languages, and applications*, pages 353–360, 2003.

[19] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[20] J. Offutt. Quality attributes of web software applications. *Software, IEEE*, 19(2):25–32, Mar/Apr 2002.

[21] S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, December 2005.

[22] R.S. Pressman. What a tangled web we weave [web engineering]. 17(1):18–21, January/February 2000.

[23] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. pages 188–197, 2004.

[24] Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Ann. Softw. Eng.*, 14(1-4):93–114, 2002.

[25] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[26] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.

[27] Luis Moura Silva. Comparing error detection techniques for web applications: An experimental study. In *NCA '08: Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*, pages 144–151, 2008.

[28] Harry M. Sneed. Testing a web application. In *Workshop on Web Site Evolution*, pages 3–10, 2004.

[29] Elizabeth Soechting, Kinga Dobolyi, and Westley Weimer. Syntactic regression testing for tree-structured output. In *International Symposium on Web Systems Evolution*, September 2009.

[30] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Automated Software Engineering*, pages 253–262, 2005.

[31] Sara Sprenkle, Emily Hill, and Lori Pollock. Learning effective oracle comparator combinations for web applications. In *International Conference on Quality Software*, pages 372–379, 2007.

[32] Sara Sprenkle, Lori Pollock, Holly Esquivel, Barbara Hazelwood, and Stacey Ecott. Automated oracle comparators for testing web applications. In *International Symposium on Reliability Engineering*, pages 117–126, 2007.

[33] Jeff Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.

[34] http://txt2html.sourceforge.net/. txt2html - text to HTML converter. Technical report, 2008.

[35] http://www.a7soft.com/jexamxml.html. A7soft jexamxml is a java based command line xml diff tool for comparing and merging xml documents. Technical report, 2008.

[36] http://www.gccxml.org/HTML/Index.html. GCC-XML. Technical report, 2008.

[37] Kenneth R. van Wyk and Gary McGraw. Bridging the gap between software development and information security. 3(5):75–79, September 2005.

[38] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Reliability, quality and safety of software-intensive systems*, pages 3–21, 1997.

[39] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2006.