Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Automating Regression Testing Using Web-based Application Similarities

Kinga Dobolyi, Elizabeth Soechting and Westley Weimer

University of Virginia

Received: date / Revised version: date

Abstract. Web-based applications are one of the most widely used types of software, and have become the backbone of many e-commerce and communications businesses. These applications are often mission-critical for many organizations, motivating their precise validation. Although regression testing has been widely used to gain confidence in the reliability of software by providing information about the quality of an application, it has suffered limited use in this domain due to the frequent nature of updates to websites and the difficulty of automatically comparing test case output.

We present techniques to address these challenges in regression testing web-based applications. Without precise comparators, test cases that fail due to benign program evolutions must be manually inspected. Our approach harnesses the inherent similarities between unrelated web-based applications to provide fully automated solutions to reduce the number of such false positives, while simultaneously returning true faults. By applying a model derived from regression testing other programs, our approach can predict which test cases merit human inspection. Our method is 2.5 to 50 times as accurate as current industrial practice, but requires no user annotations.

1 Introduction

Web-based applications have become an integral part of the global economy, with Internet-based e-commerce projected to reach over one trillion dollars by 2010 [45]. Despite their pervasiveness, most web-based applications are not developed according to a formal process model [34]. Web-based applications are subject to high levels of complexity and pressure to change, manifesting in short delivery times, emerging user needs, and frequent developer turnover, among other challenges [37]. Under such extreme circumstances, systems are delivered without being tested [37], potentially resulting in functionality losses on the order of millions of dollars per hour [33,48,51]. Such breakdowns are not isolated incidents; user-visible failures are endemic to about 70% of top-performing web-based applications, a majority of which could have been prevented through earlier detection [42]. Such monetary losses can be avoided by designing web-based applicationsto meet high reliability, usability, security, and availability requirements [32], which translates into well-designed and well-tested software.

Regression testing is an established approach for gaining confidence that a program continues to implement its specification in the face of recurring updates, and is a major part of software maintenance [36]. Maintenance activities consume 70% [13] to 90% [41] of the total life cycle cost of software, summing to over $70 billion per year in the United States [38, 49], with regression testing accounting for as much as half of this cost [23,39]. For software in general, a lack of resources often restricts developers to utilize only a fraction of the available regression tests [20,29,53]. Unfortunately, web-based application testing is often perceived to be lacking a significant payoff [21], making the likelihood of investing in rigorous testing even more remote in this domain.

One major challenge with any regression testing approach is that it is often limited by the effort required to compare results between two versions of program output. Formally, testing can be viewed in terms of an *oracle* mechanism that produces an expected result and a *comparator* that checks the actual result against the expected result [10]. The oracle is commonly taken to be the output of a previous, trusted version of the code on the same input and the comparator is a simple lexical comparison, such as `diff`, of the two outputs. Any difference is inspected by developers, as it is commonly an error in the new version, or may indicate an outdated expected output (for example, when the output legitimately changes due to new functionality). Unfortunately, traditional regression testing is particularly burdensome for web-based applications [46] because using `diff` as the comparator produces

too many false positives [48]. For example, rerunning a test suite of a web-based application may result in otherwise-identical output with a different footer, timestamp, session cookie, or dynamically generated elements. While `diff`-like tools have the strength of not producing any false negatives, a `diff` of older output against newer output will almost always flag a potential error, even if no new defect has been introduced. Each new version of the software may compound the problem when other natural program evolutions take place, by adding more and more changes flagged by `diff` that are not actual errors. Therefore, a regression testing approach for web-based applications would have to reduce the number of false positives associated with a naïve `diff`-like comparator, as well as provide an acceptable level of automation and refrain from missing actual faults. Although automating the replay of existing regression test suites is relatively straightforward in this domain, automating the comparator process remains an active research area.

This paper presents SMART, a tool for reducing the cost of regression testing for web-based applications by providing a highly precise oracle comparator. We propose to combine insights from structural differencing algorithms (e.g., [8]) in addition to semantic features (e.g., [46]) into a distance metric for test case output [1]. This distance metric forms the center of a highly precise oracle comparator which indicates test case outputs need human inspection when its distance from the oracle output exceeds a certain cutoff.

We first present our technique in a context where a small fraction of the regression testing output is manually inspected as normal, and that information is used to train a comparator based on our features; the comparator then flags which test cases in the rest of the output require human inspection. We then expand upon this proof-of-concept by demonstrating that the oracle comparator process can be completely automated in this domain, obviating the need for manual training data, because inherent similarities between web-based applications can be used 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. These similarities derive from the common multi-tiered and multi-component structures of these applications, and include behaviors such as corralling errors into user-visible HTML. Both our fully-automated and partially-automated comparators can be considered successful if they reduce false positives (i.e., correctly tell developers not to inspect test cases that have small changes but do not indicate faults) while minimizing or avoiding false negatives (i.e., incorrectly tells developers not to inspect actual defects). In our experiments, a `diff`-based comparator incorrectly labels non-faulty output as requiring inspection 70–90% of the time; our approaches typically have fewer than 1% false positives.

[1] In this paper we use the term *feature* to refer to the set of all arguments used by our distance metric: a feature is an aspect of test case output pairs, such as the ratio of natural language text between two HTML documents.

Existing comparators for web-based applications typically have false positive rates in the 4% range [47], although these false positives resulted from testing on a single version of the software using seeded faults. By contrast, this paper focuses on finding faults between different versions of applications, where the number of false positives reported by such tools increases due to innocuous program evolutions. Additionally, some previous approaches [47] require training on the target application to achieve their false positive values. By contrast, we present an automated oracle comparator that does not require manually training the tool. The main contributions of this paper are:

- A set of structural and syntactic features that are used to determine if web-based application regression test output should be inspected by human developers.
- An experimental evaluation of a proof-of-concept model and distance metric using those features.
- A quantitative and qualitative comparison of the relative power of those features, with a discussion of the possible impact on regression test practices.
- An automated oracle comparator model that meets or exceeds the performance of the proof-of-concept model that requires manually-provided training data.

A portion of these main contributions have been presented previously [15,44]. This paper additionally includes the following:

- A more expansive base of benchmark programs for empirical evaluation showing how an automated oracle comparator model can utilize the tree structured nature of web-based application output and underlying program similarities successfully. In particular, we double the lines of code compared to previous work with the addition of three real-world browser-based applications used by over 24,000–300,000 customers worldwide that rely heavily on non-deterministic output.
- An additional experiment comparing the average feature values for erroneous output compared to correct output.
- An additional experiment evaluating our model's performance when training our comparator on project-specific data only.
- A deeper analysis of relative feature powers used by our automated comparator. We identify four features that were either important in every benchmark, or were highly indicative of actual bugs, and discuss their relation to the types and severities of bugs they tended to reveal.
- An additional experiment evaluating our automated comparator on the new real-world browser-based application benchmarks that includes an analysis of the severity of actual bugs our tool failed to flag. Relying on an automated model of fault severity from [16,17], we are able to quantitatively and qualitatively assign consumer-perceived severity ratings to the faults uncovered and missed by our approach, with the expectation that missed faults will generally be non-severe.

2 Motivating Examples

Testing the functional components of web-based applications is usually achieved by some form of capture-replay (as opposed to non-functional validation such as HTML validation, link testing, or load testing), where tester input sequences are recorded and replayed [18]. For these kinds of tests the oracle output is often HTML output of a previous, trusted version of the application, and `diff` is used to compare the oracle output with the actual test output. In situations where the two outputs differ, manual inspection is required. Unfortunately, the human interpretation of test results is a "tedious and error-prone task" [48] and is potentially more burdensome for web-based applications due to the frequent false positives associated with a `diff`-like comparator.

Consider the following example from a `diff` of two GCC-XML test case outputs [2] (the text above the dashed line was generated by the older application, while the rest is output from the newer version).

```
1  <   <Namespace id="_2" name="std" context="_1"
       members=""/>
2  <   <Function id="_3" name="foo" returns="_4"
       context="_1" location="f0:8">
3  ---
4  >   <Namespace id="_2" name="std" context="_1"
       members="" mangled="_Z3std"/>
5  >   <Function id="_3" name="foo" returns="_4"
       context="_1" mangled="_Z3fooii" location="
       f0:8" file="f0" line="8" endline="15">
```

In both versions the same **<Namespace>** and **<Function>** elements are being defined. 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. A `diff`-like comparison for regression testing these two outputs would lead to a false positive in this instance. Furthermore, we hypothesize that web-based applications often 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. A more precise oracle-comparator should be able to avoid flagging the situation above as an error, and such a comparator may be completely automated by recognizing that many web-based applications evolve in this predictable way with the addition of new element attributes.

Similarly, HTML code is often updated in ways that do not change the appearance or functionality experienced by the consumer. Consider the `diff` output from two TXT2HTML test case versions [50] (the newer output is below the dashed line):

```
1  < <P>The same table could be indented.
2  < <TABLE border="1">
3  ---
4  > <p>The same table could be indented.</p>
5  > <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, probably because newer versions of HTML will not support unmatched tags, although most browsers will display these two bits of code equivalently. As in the previous example, a `diff` of these two outputs would yield a false positive due to any of the reasons mentioned above.

Using `diff`-like comparators for the above examples, in addition to outputs that involve small natural language changes, reformatting, or nondeterministic output that changes with every run of the application, would yield a high number of false positives because humans would not consider these changes errors. Simply ignoring certain classes of website updates, however, raises the possibility of missing actual bugs. This paper shows that it is possible to provide a highly precise oracle comparator that reduces the false positives in test output comparison that occur with more naïve approaches, while minimizing the number of false negatives.

3 Modeling Test Case Output Differences

The goal of our approach is to save developers effort during regression testing web-based applications, primarily by reducing the number of false positives obtained with `diff`-like tools. We specifically focus on web-based applications due to the tree-structured nature of their output; trees are well-formed objects with a directed edge relationship. Although recent work has explored using semantic graph differencing [35] and abstract syntax tree matching [31] for analyzing source code evolution, such approaches are not helpful in comparing XML and HTML text outputs. Not only do these approaches depend on the presence of source code constructs such as variables and functions, which are absent in generic HTML, to make judgments, but they are meant to summarize changes rather than determine error instances. By contrast, SMART relies on tree representations to decide the relative importance of changes between two versions of output with respect to their structural significance. Our oracle comparator, SMART, is a model, composed of a distance metric and a cutoff, that labels test case output pairs. We use a distance metric to quantify the difference between a pair of program outputs that is based on the weighted sum of individual features. If the distance exceeds a specified cutoff, the test case output pair is labeled as requiring human inspection. Rather than hard-wiring any knowledge of program semantics, SMART empirically learns the weights and cutoffs on a per-project basis via least-squares linear regression. This model was chosen over other techniques because it is straightforward to analyze; Section 4.2 presents the results of an analysis of variance of the relative contribution of various features towards an accurate classification of test case output.

Our approach classifies test case output based on structural and syntactic features of tree-structured documents. Most features are relatively simple, such as counting the number of inserted elements when converting one tree into the other. Each feature is assigned a numeric weight that measures its relative importance. A pair of test cases outputs is labeled as requiring inspection whenever the weighted sum of all its

feature values exceeds a certain cutoff. Both the weights and cutoff are learned empirically by training the model; we return to this issue when discussing our experimental setup (see Section 4.1.2).

3.1 Tree Alignment

In order to recognize such features, the input trees must first be aligned by matching up nodes with similar elements. An alignment is a partial mapping between the nodes of one tree and the nodes of the other. To see why this alignment is necessary, consider these two HTML fragments:

```
1  <u><b>textA</u></b> <i><u><b>textB</b></u></i>
2  <i><b><u>textB</u></b></i>
```

We must know how the fragments align before we can count features: if we align the `textB` subtree of #2 with the `textA` subtree of #1, we can count an inversion between the `<u>` and `<b>` tags. However, if we align the `textB` subtree of #2 with the `textB` subtree of #1, we can count inversions between the `<u>`, `<b>` and `<i>` tags.

This insight motivates us to find an alignment based on the minimal number of changes that enumerate the difference between two test case outputs. We adapt the DIFFX [8] algorithm for calculating structural differences between XML documents to compute alignments on general tree-structured data. Aligning the newer and older output trees allows SMART to identify both local features derived from element pairs, as well as global features such as the addition of natural language text spattered across multiple locations in the newer document.

Our technique employs features that fall into two loose categories: identifying differences in the tree structure of the output, and emulating human judgment of interesting differences between two XML or HTML pages, detailed below. Most of the features are derived from the DIFFX mapping; the remaining atomic features are indicated with an asterisk in Figure 1.

3.2 Tree-based differences

Unlike flat text files, the nested tree structure of XML/HTML output allows for the potential to classify many differences as either faults or non-faults through an analysis of the tree structure. Features may be positively or negatively correlated with test output errors, depending on the application being examined.

The DIFFX Algorithm. Three of our features are taken from a variant of the DIFFX [8] algorithm that we adapted to work on HTML and XML files. The algorithm computes the number of moves, inserts and deletes required to transform the first input into the second. It does this via bottom-up exact tree matching combined with top-down isolated tree fragment mapping; this amalgamated approach provides a high quality characterization of the relationship between the two input trees.

We hypothesize that moves, inserts, and especially deletes frequently correlate with bugs, and that the size of the change indicates the severity of the error. For example, a failure that results in a stack trace being printed will involve a deletion of a large amount of data and an insertion of the trace itself. Considering moves instead of delete-insert pairs reduces the number of changes between two trees.

Inversions. Inverted elements are unlikely to indicate errors in web-based application output. To calculate inversions, we perform a pre-order traversal of all nodes in both of the document trees, removing text nodes as we are interested in structural inversions. We then sort each list and calculate the longest common subsequence between them. All nodes not in the common subsequence are removed and the lists are unsorted, returning the remaining nodes to their original relative orders. We finally walk the lists in tandem, comparing element pairs and counting each difference; each difference is a structural inversion.

Grouped Changes. We note when a set of elements that form a contiguous subtree are changed as a group, measuring the size of the grouped change in terms of the number of elements involved. We hypothesize that clustered edits are more likely to be deserve inspection, often because they contain missing components or lengthy exception reports, as opposed to small changes scattered throughout the document. We also record a boolean feature that notes the presence or absence of grouped changes.

Depth of Changes. The relative depth of any edit operation within a tree is measured under the hypothesis that changes closer to the root are more likely to signal large semantic differences and thus more likely to merit human inspection.

Changes to Only Text Nodes. We record with a boolean feature whether the changes between two documents are restricted only to the natural language text in the trees. We expect that documents with such text-only differences are unlikely to contain semantic errors and thus should not be inspected. This feature in particular allows our approach to outperform `diff`-like comparators, through the ability to ignore natural-language text changes.

Order of Children. We record cases where two matched nodes are otherwise the same but the ordering of their children has changed. We hypothesize that changes in the order of child nodes do not indicate high-level semantic errors and thus should not be inspected.

3.3 Human-Judgment differences

The rest of our features attempt to emulate judgments that a human would make on two test case outputs. These features are specific to HTML and aim to model how a human would view differences.

Text and Multimedia Ratios. Natural language and graphics significantly influence human interpretation of a webpage. We measure the ratio of displayed text between two versions as well as the ratio of displayed text to multimedia objects.

Small changes, such as replacing a textual link with a button are unlikely to warrant human inspection. Replacing large amount of text, however, may be more likely to associate with errors such as stack traces.

Error Keywords. Web-based applications often exhibit similar failure modes. Beyond the standard error messages displayed by web servers (such as 404 errors), many other violations are tied to the underlying languages, and can be reasonably predicted by a textual search of the document for error keywords, such as "exception". Searching for natural language text to signal page errors has been previously explored in [9]. Output pairs containing error keywords in the newer version, but not in the older, are likely to merit human inspection.

Changes to Input Elements. Input elements, such as buttons and forms, represent the primary interface between consumers and the application. We note the changes to these input elements under the hypothesis that a missing button or form indicates a significant loss of functionality and should be examined.

Changed or Missing Attribute Values. Finally, we note when two aligned elements contain the same attribute but have a different attribute value. Consider the following `diff` output:

```
1  < <Type id="_8" name="int"/>
2  ---
3  > <Type id="_8" name="unsigned int"/>
```

If the two `<Type>` elements on lines 1 and 3 are aligned then the change from `"int"` to `"unsigned int"` represents a meaningful change. Note that this is different than an instance where the second `<Type>` has a new attribute that the original does not. Changed attributes may or may not be significant; consider the semantic difference between an update to `height` attribute of an image as opposed to the mistyping of a `action` attribute of a form element. We hypothesize that removing attributes, however, will generally be likely to merit human inspection.

3.4 Feature Validation

In this subsection, we validate some of the assumptions that underlie our approach. We determine whether it is reasonable to use tree-structured features to detect test case outputs that merit human inspection. We establish that our features generally take on different values for differences that merit human inspection than for differences that do not.

Figure 1 shows the average normalized features on 919 test cases used in our experiments (see Figure 2). For example, the normalized value of our feature that measures whether changes were to text only is 0.9946 for test case output that need not be inspected and 0.0179 for test case output that should be inspected. We return to this issue in Section 4.2 when we present an analysis of variance to inspect feature importance; the normalized values in Figure 1 are provided simply as a means of viewing the data in this preliminary experiment: our actual model does not normalize feature values.

Feature	Average – No Inspect	Average – Inspect
Text Only	0.9946	0.0179
DIFFX-move	0.0004	0.0507
DIFFX-delete	0.0007	0.1203
Grouped Boolean	0.0007	0.9739
DIFFX-insert	0.0041	0.0109
Error Keywords*	0.0000	0.0096
Input Elements*	0.0001	0.0031
Depth	0.0007	0.0172
Missing Attribute*	0.0047	0.1580
Children Order	0.0010	0.1769
Grouped Change	0.0002	0.1301
Text/Multimedia*	0.8548	0.9933
Inversions*	0.0010	0.0016
Text Ratios	0.7996	0.9636

Fig. 1. The average values of our features for test cases flagged by `diff` that (1) do not merit manual inspection and (2) do merit manual inspection, as determined by our human annotators. Each feature is individually normalized to 1.0. Features with an asterisk are atomic features not dependent on the DIFFX mapping between two output pairs.

Extracting our features from the output of one test case took 2 seconds on average, and never more than 30 seconds, on a 3 GHz Intel Xeon computer.

3.5 Model Evaluation

Having established that it should be possible to classify test case output based on the features discussed above, we must next define a way to evaluate the performance of such a model. The key task of SMART is, given the oracle output for a test case and the current output for that same test case, to indicate whether a potential error should be flagged and the situation evaluated by a human. Consequently, it is possible that we flag non-faults as requiring inspection; these false positives yield wasted developer effort as humans fruitlessly inspect correct output. Whenever our tool fails to flag actual faults, these false negatives may result in consumer-visible bugs after deployment and revenue losses. We use precision and recall, metrics from the domain of information retrieval [40], 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 which are actual bugs and our model also labels as such (true positives). *Returned* are all the test cases flagged as potential errors by our model, which is composed of both the actual faults correctly flagged by SMART, as well as the non-faults we flag as requiring inspection. *Recall* is defined as the ratio of desired error test cases our model returns over the total number of desired error values (in other words, 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 actual error test cases our model flags 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 we have too many false positives and would waste significant amounts of developer effort. 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 [14].

4 Model Validation

We first evaluate SMART on ten open-source benchmarks that produce either XML or HTML output, totaling 473,000 lines of code, summarized in Figure 2. In order to provide a realistic regression testing simulation, we chose benchmarks for which multiple versions were available, to capture as many natural program evolutions as possible. We supplemented our three large benchmarks with seven smaller ones to vary the domains of the applications examined in this proof-of-concept experiment. Because these additional benchmarks have significantly fewer test cases than the primary three applications, Section 7 supplements our dataset with over 6000 additional test cases from three real-world, PHP browser-based applications.

For each benchmark, we manually inspected the test case output generated by the two versions of the benchmark indicated on the same benchmark-specific test suite. Our manual inspection marked the output as "definitely not a bug" or "possibly a bug, merits human inspection". In an effort to not rule out actual bugs, we conservatively erred on the side of requiring human inspection. Each test case was annotated twice, re-examining situations where the two annotations did not initially agree, to maintain consistency. Our initial experiments involve 7154 pairs of test case output, of which 919 were labeled as requiring inspection.

4.1 Experiment 1 – Model Selection

In this experiment, the feasibility of our approach is evaluated when phrased as an information retrieval task. We create a linear regression model based on the features from Section 3 and select an optimal cutoff to form a binary classifier.

4.1.1 Cross Validation

In order to validate our approach, our initial experiment involves testing and training on the same data. One potential problem with such an approach is biased from over fitting on our dataset. To mitigate this threat, we used 10-fold cross validation [26]. We randomly assigned test cases from all benchmark programs into ten equally-sized groups. Each group is reserved once for testing, and the remaining nine groups are used to train the model; we thus never train and test on the same data. The cross validation results are then averaged across each batch and compared to the results when we tested and trained on the entire dataset. If the two outcomes are not significantly different, we can conclude that we were not subject to bias.

Comparator	F_1-score	Precision	Recall
SMART	0.9931	0.9972	0.9890
SMART w/ cross-validation	0.9935	0.9951	0.9920
`diff`	0.3004	0.1767	1.0000
`xmldiff`	0.2406	0.1368	1.0000
fair coin toss	0.2045	0.1286	0.4984
biased coin toss	0.2268	0.1300	0.8868

Fig. 3. The F_1-score, precision, and recall values for SMART on our entire dataset. Results for `diff`, `xmldiff`, and random approaches are given as baselines; `diff` represents current industrial practice.

4.1.2 Experimental Procedure

SMART classifies pairs of tree-structured outputs as either requiring human inspection or not. In this experiment:

1. We first perform the cross-validation steps (Section 4.1.1). On each iteration, we train a linear model as if the response variable (i.e., our boolean human annotation of whether a human should inspect that output or not) were continuous in the range [0,1].
2. The real-valued model outputs are turned into a binary classifier by comparing against a cutoff by performing a linear search to find a model cutoff. SMART will flag the test case as needing inspection or not depending on how the response variable compares to the cutoff. We choose the cutoff and comparison that yield the highest F_1-score for each validation step.
3. After cross-validation, the model is trained and tested on our entire dataset. We again find the best model cutoff and comparison to maximize the F_1-score.

4.1.3 Results

Figure 3 shows our precision, recall, and F_1-score values for our dataset. As a point of comparison, we also computed the predictive power of other comparator approaches, namely, `diff`, `xmldiff` [4][2], coin toss, and biased coin toss as baseline values. The fair coin returns "no" with even probability. The biased coin returns "no" with probability equal to the actual underlying distribution for this dataset: $(7154 - 919)/7154$ (which is generally not known in advance). SMART has clear advantages in predictive power over all of our baselines, with our approach yielding three times `diff`'s F_1-score.

[2] `xmldiff`, an off-the-shelf `diff`-like tool for XML and HTML [4], was a worse comparator than than basic `diff` because it was unable to process some benignly ill-formatted output.

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. 2. The benchmarks used in our experiments. The "Test cases" column gives the number of regression tests we used for that project; the "Test cases to Inspect" column counts those tests for which manual inspection indicated a possible bug.

Feature	Coefficient	F	p
Text Only	- 0.288	168970	< 0.001
DIFFX-move	+ 0.002	150840	< 0.001
DIFFX-delete	+ 0.029	46062	< 0.001
Grouped Boolean	+ 0.714	7804	< 0.001
DIFFX-insert	+ 0.029	4761	< 0.001
Grouped Change	- 0.012	465	< 0.001
Children Order	- 0.002	317	< 0.001
Inversions	+ 0.001	246	0.020
Missing Attribute	- 0.048	121	< 0.001
Error Keywords	+ 0.174	115	< 0.001
Depth	- 0.000	21	< 0.001
Text Ratios	- 0.007	18	< 0.001
Input Elements	- 0.019	5	0.030

Fig. 4. Analysis of variance of our model. A + in the "Coefficient" column means high values of that feature correlate with test cases outputs that should be inspected (both + and - indicate useful features). The higher the value in the "F" column, the more the feature affects the model. The "p" column gives the significance level of F; features with no significant main effect ($p \geq 0.05$) are not shown.

As shown in Figure 3, little to no bias was revealed by cross-validation. The absolute difference in F_1-score between the model and its corresponding averaged cross validation steps was 0.0004.

4.2 Experiment 2 – Feature Analysis

Having established that SMART is able to successfully flag both faults and non-faults, we evaluate relative feature importance and determine which features correlate with output that should be inspected. Figure 4 shows the results of a per-feature analysis of variance on the model using the entire dataset, listing only those features with a significant main effect. F denotes the F-ratio, which is close to 1 if the feature does not affect the model; conceptually F represents the square root of variance explained by that feature over variance not explained. The p column denotes the significance level of F (i.e., the probability that the feature does not affect the model).

Our most significant feature was whether or not the difference between a pair of test cases output could be quarantined to only low-level text. We found that text-only changes have a strong negative effect on human inspection, and relying on such a feature is one of the key reasons we are able to outperform `diff`, because many web-based applications may update natural language text as part of normal program evolutions.

Our second most important feature was DIFFX-move, which, by contrast, was frequently correlated with test case errors. Rather than relying on only insertions or deletions, moves are able to capture both of these types of changes as a move always occurs in conjunction with any of the other two edits. Consequently, despite the high F-ratio of the DIFFX-move feature, its model coefficient was an order of magnitude smaller than those of insert or delete. Therefore, although moves were most frequently associated with actual bugs, other features also had to be present in order for the test case output to merit human inspection. Our boolean feature that indicated the presence of clustered changes was also highly correlated with errors. Notably, grouped changes were more important than DIFFX-inserts, which may have been scattered across the output.

Some of our features were less powerful than we originally hypothesized. For example, the presence of error keywords did not effect our model as much as the features listed above. We return to the issue of error keywords in Section 7.2.3. Despite the moderate F-ratio of this feature, its coefficient was the highest across all features, indicating that it was strongly predictive of error instances when present.

Additionally, SMART depends heavily on the DIFFX algorithm to calculate structural changes between pairs of web-based application output. DIFFX is an approximation, and any errors in mapping one output tree to another may reduce the performance of our approach. To investigate the impact of DIFFX's potential false positives and false negatives on SMART's performance, we conducted a pilot study where we manually perturbed the number of DIFFX-reported insertions, deletions, and moves reported by between zero and at one standard deviation for each of these three features. We then examined changes in SMART's performance when these error-seeded DIFFX feature values were randomly applied to between 0 and 100% of the test case output pairs. Overall, even in the worst case, where every output pair's feature val-

Benchmark	Comparator	F_1	Prec	Reca
HTMLTIDY	SMART	1.000	1.000	1.000
	`diff`	0.048	0.025	1.000
	`xmldiff`	0.021	0.010	1.000
GCC-XML	SMART	0.999	1.000	0.999
	`diff`	0.352	0.213	1.000
	`xmldiff`	0.352	0.213	1.000
All ten (global)	SMART	0.993	0.997	0.989
	`diff`	0.300	0.177	1.000
	`xmldiff`	0.241	0.138	1.000

Fig. 5. F_1-score, precision (Prec), and recall (Reca) when trained and tested on individual projects, as well as all ten of our benchmarks. Results for `diff` are presented as a baseline.

ues were perturbed by an entire standard deviation, the $F1$-score for our dataset decreased by at most 0.001, suggesting that our overall model is robust to errors in the underlying structural mapping.

To gain additional insight into relative feature power and number of required features beyond that provided by the F-values in Figure 4, we also conducted a "leave-one-out" experiment. We cumulatively removed one feature at a time, starting with the feature with the lowest F-value. When removing up to four of the lowest-performing four features (Input Elements, Text Ratios, Depth, and Error Keywords), the F_1 score varied by less than 0.001. F_1-scores dropped from 0.99 to to 0.98 when only three features remained, and dropped to 0.26 when only two features remained. Although some of the changes in F_1 score seem small, subsequent experiments (e.g., Figure 6) demonstrate that even a 1% increase in false positives has a severe impact on the real-world usability of the technique.

4.3 Experiment 3 – Project-specific Models

Having established the feasibility of training such a general comparator, we investigated whether it was possible to improve our model's performance when training a tailored comparator for specific projects. We used the same experimental setup for our per-project classifiers described in Section 4.1.2. We restricted attention to the 6513 test case output pairs for GCC-XML and HTMLTIDY, because those two benchmarks were large enough to be feasible for individual study. Figure 5 shows SMART's average F_1-score, precision and recall values when trained and tested on each program separately.

For HTMLTIDY we obtain perfect performance, with no false positives or false negatives. Our precision is thus an order of magnitude better than that of `diff`: our technique presents only 25 test case outputs to developers compared to the 960 flagged by `diff`.

For GCC-XML we obtain near-perfect recall (0.999) and perfect precision; we present 874 test cases for human inspection compared to `diff`'s 4100, but we fail to flag one test case that did merit human inspection.

Project-specific feature weights contributed to our strong per-project performance. For example, the DIFFX-delete and DIFFX-insert features were equally important for the HTMLTIDY project, but not across all benchmarks. As another example, DIFFX-insert and error keywords were significantly associated with errors in HTMLTIDY but not at all in GCC-XML. Intuitively, using our technique for either of the projects alone is always advantageous. When the same global model is applied to all ten benchmarks, however, recall suffers: we fail to present some actual regression test errors for human inspection.

5 Training the Model with Human Annotations

Although SMART has near-perfect precision and recall in our experiments so far, it is unlikely that humans will be willing to annotate ninety percent of their test case outputs as cross-validation training data for the comparator. In this section, we explore a more realistic scenario where developers are responsible for training SMART on twenty percent of the output from each run of the test suite, which they manually annotate, and testing on the remaining eighty percent. We include multiple releases of the same product to characterize the performance in this longitudinal manner. Subsequent releases of the same project retain training information from previous releases, as well as incorporating the false positive or true positive results of any test case that our tool deemed to require manual inspection.

Our goal is to estimate the effort we save developers when using SMART, by defining a cost of looking (*LookCost*) at a test case and a cost of missing (*MissCost*) for each test case that should have been flagged but was not. We consider SMART a useful investment when the cost of using it:

$$\begin{aligned} &\mathit{LookCost} \times (\mathit{TruePos} + \mathit{FalsePos} + \mathit{Annot}) \\ &+ \mathit{MissCost} \times \mathit{FalseNeg} \end{aligned}$$

is less than the cost of $|\texttt{diff}| \times \mathit{LookCost}$. Here Annot denotes the number of test cases that are manually annotated for training (20% of the total). Therefore, SMART saves effort when the cost of looking at the test cases flagged by `diff` but not by our model exceeds the cost of missing any relevant test cases we fail to report. We thus express the condition under which SMART is profitable:

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

We assume $\mathit{LookCost} \ll \mathit{MissCost}$ [54], so we would like this ratio to be as small as possible.

5.1 Results

Figure 6 shows our results, as well as the number of test cases that a developer would need to examine when using `diff` as a baseline. For example, when applying our technique to the last release of HTMLTIDY, if the cost of missing a potentially useful regression test report is less than or equal to 1000 times the cost of triaging and inspecting a test case, we save

developer effort. A ratio of 0 indicates no false negatives, and thus a net win compared to `diff`, regardless of *LookCost* or *MissCost*.

SMART's performance generally improves on subsequent releases, and it totally avoids missing any faults in one instance for both HTMLTIDY and GCC-XML. In situations where there is a large increase in the relative number of regression test errors, such as a rushed release of a product, such as the fourth release of HTMLTIDY, SMART performs the worst. When the manual annotation of 20% of the training test case outputs shows a historically unreasonable number of regression test errors, a developer would not be advised to use our approach.

Previous work on bug report triage has used a *LookCost* to *MissCost* ratio of 0.023 as a metric for success for an analysis that required 30 days to operate [22]. The typical performance of our technique, which includes the cost of the 20% manual annotation burden and would take 1.3 hours on average per release, is 0.0183 — a 20% improvement over that figure. By excluding the single HTMLTIDY outlier mentioned above, our ratio improves to 0.0015, exceeding the utility of previous tools by an order of magnitude and requiring an order of magnitude less time.

6 Automating the Model

The previous sections demonstrated that designing a highly precise oracle comparator that relies on surface feature values between pairs of regression test case output is a feasible and reasonable approach for reducing the cost of regression testing web-based applications. Our comparator, SMART, significantly outperforms `diff` and other baselines, and is able to successfully save developer effort. Given the resource constraints web-based applications are developed under [37], and the perceived low return on investment in this domain [21], we believe that in order for any approach to be adopted in industry, automation is essential. The remainder of this paper explores *automated* approaches for oracle comparators for web-based applications, that do not require manual annotation of training data.

Although Section 4.3 presented results where a comparator customized to a benchmark application outperforms that trained on a global dataset, we believe that inherent underlying similarities between unrelated web-based applications can be exploited to automate the oracle comparator process.

6.1 Experimental Setup

To test our hypothesis that underlying similarities from unrelated web-based applications are sufficient for training a highly precise oracle comparator, we designed an experiment which relied on the same training dataset from Section 4 (see Figure 2), and tested on applications not in the training set. Optionally, developers may choose to supplement the training dataset with their own annotations of data from the application-at-test, but we do not require them.

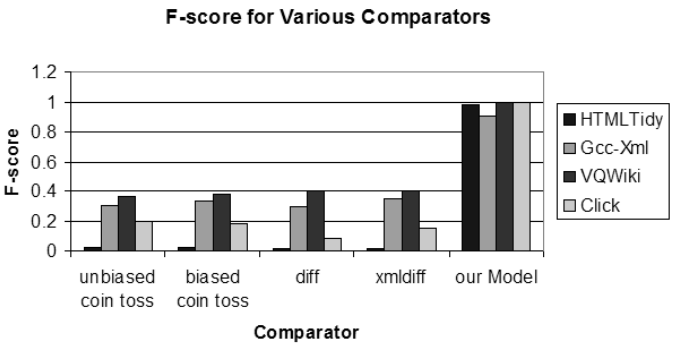


Fig. 8. F_1-score on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK) using our Model, and other baseline comparators. 1.0 is a perfect score: no false positives or false negatives.

We selected four benchmarks, shown in Figure 7, to serve as our test data. HTMLTIDY and GCC-XML were the only benchmarks from our previous experiments that had enough output pairs labeled as faults (given by the "Test Cases to Inspect" column) to serve as *testing* (as opposed to training) subjects. We also relied upon two open source browser-based applications (CLICK and VQWIKI) to supplement our test benchmarks in a "worst-case scenario" fashion: none of the training benchmarks are browser-based applications, and successful performance on them further supports our claims about inherent web-based application similarities.

VQWIKI [3] is wiki server software that can be used out-of-the-box as a browser-based application. CLICK [1] is a Java Enterprise Edition web-based application framework that ships with a sample browser-based application demonstrating the framework's features. HTMLTIDY and GCC-XML are two open-source HTML- and XML-based applications that are also a part of our training benchmarks; we therefore 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. In total we tested our model on 6728 test case pairs, 941 of which were labeled as errors by manual inspection (see Figure 7).

6.2 Experimental Results

Figure 8 shows our model's F_1-score values for each test benchmark, as well as the baselines in Section 4.1.3 (see Figure 3). Figure 9 and Figure 10 present the recall and precision values, respectively, from which the F_1-scores were calculated. SMART 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`. Both browser-based applications (CLICK and VQWIKI) obtain perfect results, and the F_1-score of 0.98 for our second largest HTML benchmark, HTMLTIDY is near optimal. Overall, relying on underlying web-based application similarities to train an oracle comparator to test an unrelated web-based application is a successful approach. For example, an analysis of variance

Benchmark	Release	Test Cases	Should Inspect	True Positive		False Positives		False Negatives		Ratio
				SMART	diff	SMART	diff	SMART	diff	
HTMLTIDY	2nd	2402	12	5	12	78	781	7	0	0.0099
	3rd	2402	48	48	48	0	782	0	0	0
	4th	2402	254	109	254	1	574	145	0	0.2019
	5th	2402	48	48	48	0	775	0	0	0
	6th	2402	20	19	20	1	774	1	0	0.0013
GCC-XML	2nd	4111	662	658	662	16	2258	4	0	0.0018
	3rd	4111	544	544	544	0	2577	0	0	0
Total		20232	1588	1431	1588	96	8521	157	0	0.0183

Fig. 6. Simulated performance of our technique on 20232 test case executions from multiple releases of two projects. The "Test Cases" column gives the total number of regression tests per release. The "Should Inspect" column counts the number of those tests that our manual annotation indicated should be inspected (i.e., might indicate a bug). The "Inspected" column gives the number of tests that our technique and `diff` flag for inspection. The "False Positives" and "False Negatives" columns measure accuracy, and the "Ratio" column indicates the value of *LookCost*/*MissCost* above which our technique becomes profitable (lower values are better).

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. 7. 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.

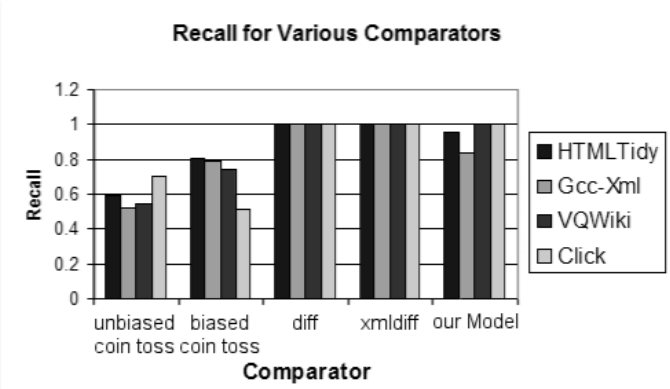


Fig. 9. Recall on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK) using our Model, and other baseline comparators.

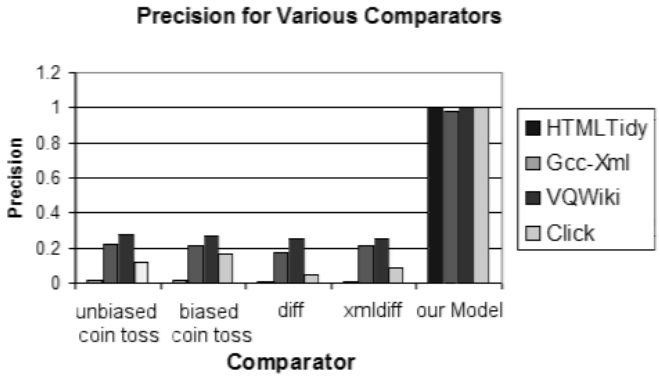


Fig. 10. Precision on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK) using our Model, and other baseline comparators.

revealed that text-only changes were strongly negatively associated with bugs across all benchmarks, and similar to the conclusions presented in Section 4.2. By utilizing our readily available training data set and companion comparator, developers would be able to significantly reduce the number of false positives associated with testing web-based applications while minimizing or eliminating false negatives.

While our F_1-score for our 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 faults. Unlike our other test applications, GCC-XML outputs XML rather than HTML code, and may not be able to make full use of our human-emulating semantic features described in Section 3.3, or may exhibit errors differently than in the HTML-emitting applications (we return to this notion in Section 7.2). Instead of suggesting developers continue to use `diff`-like comparators, or rely on extensive manual annotation, we propose that they continue to apply our approach

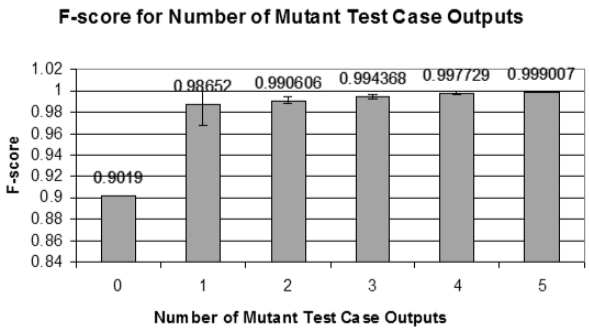
with one small modification: by extending the training data set with seeded faults obtained from automated fault injection, developers can customize SMART to a specific application while still avoiding relying on manual annotations. We explore this concept in the following subsection.

6.3 Training Data from Defect Seeding

Our relatively low recall value for GCC-XML suggests that this application-at-test exhibits errors in a way that differs from our other training benchmarks. By relying on automated defect seeding, we claim that it is possible to improve the results of our automated comparator to levels comparable with a manual approach.

Seeding the source code of an application with defects has been previously explored [25,27,47]. It can be assumed that any output from a fault-seeded version for a deterministic test case that differs from the expected output can be attributed to the injected fault, and thus that output pair should be added to the training data set with the label "should inspect". While automatically generating, compiling and running mutants can be CPU-intensive, it does not require manual intervention. In this experiment, a subset of mutation operators from Ellims *et al.* [19] were injected into a single version of the source code on a single random line. 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`. Each mutant version of the source code, containing a single mutation, was then re-run on the entire test suite, and all differing outputs were collected; we were able to obtain 11,000 usable erroneous output pairs within 90 minutes on a 3 GHz Intel Xeon computer.

Figure 11 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) over 1000 trials. Adding any single mutant is always better than using none, although the large margin of error implies that best performance is obtained when selecting the most appropriate mutant. After adding 5 mutants our F_1-score of 0.999 approaches the optimal value of 1. We hypothesize that it is possible to dramatically affect SMART's predictive power by adding a small number of mutants 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. Other benchmarks may require more mutants to obtain similar levels of increased performance, although we have demonstrated that it is quite simple to automatically generate these defects. We conclude that very little application-specific training data (5 labeled output pairs) is needed to bring even our-worst performing benchmark up to near-optimal performance, and even such application-specific data can be easily obtained automatically.



[t]

Fig. 11. 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.

7 Case Study: Popular Browser-based Applications

Section 5 and Section 6 explored annotation-based and automated approaches using SMART to reduce the number of false positives associated with regression testing web-based applications, while minimizing or eliminating false negatives. This section presents the results of SMART's performance on three popular, open-source PHP browser-based applications. Although Section 6.2 evaluated our model's performance on two browser-based applications, CLICK and VQWIKI, our goal is to investigate SMART's performance on additional and potentially more typical and challenging benchmarks. For this purpose we evaluated our tool's performance on three open-source popular browser-based applications summarized in Figure 12.

Our first benchmark, PRESTASHOP, is an e-commerce application with over 24,000 companies employing their own instances of the product worldwide [6]. Besides featuring authentication and database properties, PRESTASHOP is interesting from a regression testing perspective because it makes use of a non-deterministic "featured product" which changes between runs of the application test suite. OPENREALTY is an on-line real estate listing management application with over ten thousand registered members in their development forum [5]. VANILLA is a standards-compliant, multi-lingual, theme-able, pluggable discussion forum for the web with "over 300,000 businesses, brands, and fans" [7]. All three PHP applications make use of session cookies that result in additional non-deterministic output that would be flagged by a naïve comparator such as `diff`.

To provide a precise analysis of SMART's fault revealing properties for these popular browser-based applications, we used manual seeding of source code faults for each benchmark, rather than running the test suite on two different versions as in previous sections. Although this no longer provides us an opportunity to ignore natural program evolutions, we chose to conduct this experiment to know with certainty

Benchmark	Versions	LOC	Description	Training Cases		Testing Cases	
				Faults	Non-Faults	Faults	Non-Faults
PRESTASHOP	v1.1.0.55	155K	e-commerce (shopping cart)	0	83	431	83
OPENREALTY	v2.5.6	185K	real estate listing management	4506	33	62	33
VANILLA	v1.1.5a	35K	web forum	895	48	462	48
Total		375K		5401	164	955	164

Fig. 12. Additional web-based application benchmarks. PRESTASHOP, OPENREALTY, and VANILLA are all popular, open-source PHP applications that make heavy use of non-deterministic output. The "Training Faults" test cases were obtained by automatic fault seeding (see Section 6.3). The "Testing Faults" were manually seeded. The "Non-Faults" columns represent clean runs (and may included non-deterministic data, such as "featured product" information). Ideally, all of the "Testing Cases" instances should be correctly categorized by a precise comparator.

whether or not the faults that we should flag are actual errors. This is in contrast to the *potential* faults flagged in previous sections. Furthermore, we believe that the heavy use of non-deterministic output in these benchmarks provides a challenge equal to that of ignoring harmless program evolutions in our previous experiments.

7.1 Severity of Missed Faults

We follow the setup of Section 6.3 for this experiment; a comparator is trained for each individual benchmark using the pre-existing training data from the benchmarks in Figure 2, as well as additional automatically injected faults obtained using the methods in Section 6.3. Figure 12 indicates the number of automatically injected faults used as training data for each benchmark in the "Training Faults" column. Because of the heavy reliance on non-deterministic output for these PHP benchmarks, we also supplemented the training data set with clean runs of the test suite where no faults were injected but where non-deterministic output still existed, these are labeled as "Training Non-Faults". The training data for OPENREALTY and VANILLA were augmented with such injected-fault information; for comparison, PRESTASHOP was not. The number of training faults and non-faults used in the remaining two benchmarks were natural artifacts of the number of automatically injected faults that happened to be exercised by a test case, and the size of the test suite, respectively — there was no conscious effort to present any particular ratio of faults to non-fault for the training dataset. We hypothesize that it may be more difficult to reduce false positives in web-based applications which make the heaviest use of non-deterministic output.

The "Testing Faults" column of Figure 12 gives the number of manually injected faults SMART should detect. Because these are known faults, rather than potential faults as flagged by a human annotator, we expect the experiments in this section to have more false negatives. Therefore, we also seek to characterize the *consumer perceived severity* of correctly flagged and missed faults. We claim that if the severity of missed faults is not high, using SMART under the automated approach is still a useful investment to developers, even if some bugs are missed, given the resource constraints of development in this domain. Finally, the "Testing Non-Faults" column gives normal test case output that should be correctly classified — different from the training data in terms of non-deterministic output.

We follow previous work [16, 17] in dividing fault severity (i.e., consumer perceptions of fault importance) into four levels: severe, medium, low and very low. These severity ratings correspond to varying levels of human actions based upon a fault seen; for example, severe faults occur when the user would either file a complaint or probably not return to the website. A rating of very low indicates that no fault was noticed by the consumer. The severity model used here is based on a human study. We involve fault severity because SMART may fail to report some defects, but not all defects are equally important to users.

Figure 13 presents the results of a severity analysis of the faults missed by our approach, broken down into severe and non-severe (medium, low, and very low) categories. The "Weighted Found" column refers to the percent of total faults correctly identified by SMART when assigning weights ranging from 1–4 for severity in increasing order. For PRESTASHOP, OPENREALTY, and VANILLA, our technique missed 1%, 9%, and 0% of the severe faults in their testing data respectively, indicating that overall the percentage of severe faults missed is extremely low. Out of 532 severe faults considered, SMART missed only 7.

The last column of Figure 13 shows the number of non-faults that were classified correctly by SMART. Using the terminology of Section 5, this corresponds to the amount of effort saved compared to `diff` (i.e., the factor multiplied by *LookCost*). For example, for VANILLA, our approach reduces developer inspection costs by 97% without missing any severe faults. For PRESTASHOP, our approach reduces inspection costs by 47% while missing only 1% of severe faults.

7.2 Feature Analysis

Our final experimental goal is to explicitly evaluate the hypotheses about feature associations with faults and non-faults presented in Section 3. Figure 14 presents the results of analysis of variance experiments conducted on all of our test applications. Each value represents the coefficient of the feature in the model — higher values are more associated with errors, for features whose p-value was less than 0.05. F values

Benchmark	Goal Severe Faults	Goal Medium Faults	Goal Low Faults	Goal Very Low Faults	Miss Severe Faults	Miss Medium Faults	Miss Low Faults	Miss Very Low Faults	Weighted % Found Faults	% Effort Saved
PRESTASHOP	302	45	57	27	3	32	17	26	98%	47%
OPENREALTY	44	1	1	16	4	1	0	10	85%	100%
VANILLA	186	183	10	83	0	5	8	0	89%	97%

Fig. 13. Breakdown of missed faults across our three PHP benchmarks. Human-perceived fault severity is emphasized, and ranges over "very low", "low", "medium" and "severe".

are not shown, and for the training dataset, only those features are shown which overlap with at least one significant feature from one of the testing benchmarks. We now examine the results qualitatively in detail, tying in feature importance to concrete software engineering features. In this section we conduct a deeper analysis of the delete, move, text-only, and error keywords features, beginning with their significance in our representative PHP benchmarks, and extending our analysis to all test applications we have visited in our work.

7.2.1 Moves and Deletes

Of all the features SMART employs, DIFFX-move and DIFFX-delete were the only two that correlated with judgments in a statistically significant manner across every test benchmark, as seen in Figure 14. In addition, the F-value for at least one of these features was always either the highest or second-highest feature for each specific benchmark model, indicating that these features significantly affected the respective model.

Given the importance of DIFFX-move and DIFFX-delete in our models, we sought to characterize the nature of the faults they tended to predict, as they were generally indicative of faults across most benchmarks. To do so, we manually examined the HTML output for test cases in our dataset which were labeled as having high DIFFX-move and DIFFX-delete values, as well as those with low DIFFX-move and DIFFX-delete values. For our three PHP browser-based applications, high DIFFX-delete values indicated large chunks of expected output were missing. For example, in VANILLA an output pair with a high DIFFX-delete feature value typically corresponds to output in which all forum comments are gone, or in which a search returns no results. In OPENREALTY, high DIFFX-delete values were associated with missing a large loan calculator form, while in PRESTASHOP, they indicated blank pages that contained a single error message such as "Error: install directory is missing". Similarly, high DIFFX-delete values in VQWIKI and CLICK, our other two browser-based applications, were instances of pages not being found or missing the entire body of data. By contrast, low DIFFX-delete values were associated with less severe errors such as missing links, small parts of pages, or small bits of functionality such as a Javascript calendar.

Like DIFFX-delete, high DIFFX-move values were also generally associated with faults. For our three PHP benchmarks, high DIFFX-move scores indicated authentication failures, missing entire forms, or other high severity faults with explicit error messages. For our other benchmarks, high DIFFX-move values revealed the same faults as those with DIFFX-delete scores. Low DIFFX-move values were less indicative of lower severity errors than low DIFFX-delete scores. For example, low DIFFX-move scores were found when large parts of the webpage were missing, as well as for small amounts of missing data such as parts of pages or incorrectly calculated data items.

Overall high DIFFX-delete and DIFFX-move are generally indicative of severe faults in web-based applications, as they correlate with large amounts of missing data. The user-perceived severity of such faults can frequently be predicted by the size of the DIFFX-delete value.

7.2.2 Text-only differences

The third feature we examine in depth is when the difference between two HTML outputs can be qualified as only changes to the natural language text within the documents. More than any other feature, text-only changes significantly impacted the negative performance of SMART in cases of false positives and false negatives when its respective F-value was high (PRESTASHOP, OPENREALTY, VANILLA, GCC-XML, and HTMLTIDY).

For example, text-only changes in PRESTASHOP correlated negatively with faults, but were not helpful at reducing false positives. For PRESTASHOP, a rotating "featured product of the day" display caused even the "clean" (non-fault) output pairs to include text-only changes. Text-only changes in OPENREALTY and VANILLA had the opposite effect on SMART's performance: rather than failing to rule out false positives, in the case of OPENREALTY, text-only changes were responsible for every false negative. For example, faults such as incorrectly calculating the number of comments on a forum appears to our model as simple text-only changes. In VANILLA, all other false negatives were due our tool ignoring changed HTML attribute values, our default behavior to avoid flagging changes to image height and other non-errors. For example, in VANILLA an input field's `name` attribute was mistyped. In future work, we propose to explore SMART's performance when flagging attribute changes for HTML files as well as XML output.

7.2.3 Error Keywords

Finally, we return to the issue of error keywords and their ability to predict faults in web-based application output. Man-

Benchmark	move	ins	del	text only	group	child	missing attr	inv	new text	depth	error keywords	text ratio	group binary	functionality	new text
PRESTASHOP	+0.084	-0.004	+0.009	-0.368		-0.008	-0.515								
OPENREALTY	+0.008		+0.036	-0.388								+0.576			
VANILLA	-0.002	+0.023	+0.024	-0.326	-0.025	-0.003									
HTMLTIDY	+0.001	+0.050	-0.025	-0.919	-0.121			-0.061	+0.010		+0.220	+0.004	+0.553	+0.630	-0.010
GCC-XML	+0.000		+0.005	-0.163	-0.013			+0.000		-0.011			+0.835		
CLICK	+0.000	+0.000	+0.000	+0.000	+0.000	+0.000		+0.000		+0.000	+1.000	+0.000			
VQWIKI	+0.088	+0.000	-0.000												
TRAINING	+0.002	+0.029	+0.029	-0.288	-0.012	-0.002	-0.048	+0.001		-0.000	+0.174	-0.007	+0.714	-0.019	

Fig. 14. Coefficients of feature values across all test benchmarks, plus the generic training dataset.

ual inspection of output pairs rated as severe revealed that they frequently contain error keywords. Despite this, the overall predictive power (F-value) of this feature was generally low, with the exception of CLICK where error keywords were able to perfectly predict actual faults. The main reason is that none of these real-world browser-based applications ever displayed a stack trace directly, and instead wrapped their visible errors in more human-friendly formats; this is in contrast to our experience with web-based application faults in general [17]. From a consumer perspective, it should be possible to design web-based applications that fail elegantly without stack traces or upsetting error messages [17] and yet still rely on a sophisticated tool such as SMART that can nevertheless detect such failures without relying on a search for any particular keywords in the document.

8 Experiment Summary

This paper approaches the problem of providing a highly precise oracle comparator for web-based application regression test output by proposing SMART, an approach that relies on surface features of web-based application output to determine whether or not human should inspect test case output pairs. Section 4 demonstrated that such structural and semantic features can be successfully used to model web-based application output and make judgments about errors. Initially, we tested SMART in instances where manual annotation of test case output is required to train the classifier. On 7154 test cases from 10 projects, SMART obtains a precision of 0.9972, a recall of 0.9890, and an F_1-score of 0.9931, over three times as good as `diff`'s F_1-score of 0.3004. Encouraged by these very strong machine learning results, we further tested our technique in a simulated deployment involving 20232 test case executions of 6513 test cases and multiple releases of two projects. Under such circumstances humans were asked to annotate 20% of their test case output pairs to serve as training data for the classifier; the remaining 80% of the test case output was then offered for classification by our tool. In such a scenario we had 8425 fewer false positives than `diff`, and we save development effort when the ratio of the cost of inspecting a test case to the cost of missing a relevant report is over 0.0183; numbers in that range correspond to industrial practice.

Given the extreme resource constraints under which web-based applications are developed, we extended this proof-of-concept model from Section 5 by replacing it with a completely automated approach. Section 6 demonstrated that using test case output pairs from *unrelated* web-based applications to train a model to predict errors in output in the application-at-test is a viable strategy. SMART was able to achieve perfect recall and precision for our two browser-based application benchmarks, while close to perfect (0.98 and 0.99) F_1-scores for our two web-based applications. To obtain such a near-perfect F_1-score for GCC-XML, our XML-emitting benchmark application, we augmented the training data set with automatically generated outputs obtained via defect seeding. 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, significantly reducing the number of false positives, and thus the human resources required, when compared to this more naive approach.

Finally, we extended our analyses to three real-world, popular, open-source PHP benchmarks to more accurately measure SMART's ability to detect faults in a more natural setting where web-based applications make heavy use of non-deterministic data. When tested on known, manually-injected faults, SMART was able to correctly flag most of the actual bugs, missing only 1% actual severe errors on average. We found out that our ability to accurately identify faults is inversely proportional to our effectiveness at ruling out non-faults. Finally, we analyzed our model's feature importance for these three benchmarks, extending our study to all test items in this paper. We found that DIFFX-move and DIFFX-delete significantly affected all models across all benchmarks. DIFFX-delete in particular was highly predictive of fault severity. We also examined the effect our natural language analyses had on our models. Our text-only feature was, in general, useful for outperforming `diff`-like tools, but was sometimes not helpful in reducing false positives for challenging non-deterministic data such as a rotating featured product on a e-commerce site. Ultimately, we believe that SMART's ability to usefully capture or ignore textual changes in a document relies on the structural context of that text, effectively raising the abstraction level of the text. Finally, we analyzed the

importance of error keywords in our model and discovered an overlap between this feature and DIFFX-delete, which can explain the lower importance of this feature. This implies that SMART is able to detect errors even without the presence of such natural language text, which is beneficial for developers who choose to provide more user-friendly failure modes to their consumers.

9 Threats to Validity

Although this paper presents significant savings in terms of developer effort during regression test comparison and evaluation, it is possible that our results do not generalize to industrial practice. For example, the benchmarks we selected may not be indicative of other applications. To mitigate this threat, we attempted to choose open-source benchmarks rather than toy applications, 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 [48] 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. We also chose to supplement our benchmarks with three widely-used popular open source browser-based applications in Section 7, which we believe to be representative of common websites. These specific benchmarks relied heavily on non-deterministic output and were selected to pose an additional level of difficulty for that reason.

In Section 3.3 and Section 6 it is possible that our human annotations were unable to accurately label potential errors in web-based application regression test output. To avoid missing actual bugs, our annotations were conservative: we only annotated test outputs as not meriting inspection if we were highly certain they did *not* indicate an error. Consequently, we may have mislabeled non-errors as errors, which reduces our ability to outperform `diff`, but does not impact the correctness of our approach. Similarly, because annotators were also partly responsible for suggesting features for the model, it is possible that bias exists in the annotations themselves, although this situation was guarded against by annotating each pair of output at least twice for consistency. In addition, the experiments in Section 7 utilize known faults, rather than potential faults, thereby avoiding this problem entirely.

Many of our experiments make use of automated defect seeding to obtain faults to be used as part of the training data set for our comparator. There may be some classes of web-based applications, however, where defect seeding is inappropriate to generate such mutants. For example, consider a Wiki application where the formatting and content of displayed natural language text *is* important. Fault seeding may be unable to generate instances on which to train our model to recognize potentially meaningful changes to natural language text which may be otherwise ignored in most of our benchmarks.

10 Related Work

Many testing methodologies use oracle comparators that require manual intervention in the presence of discrepancies [18, 28, 37, 48]. For example, user session data can be used as both input and also test cases [18, 46], but there must be a way to compare obtained results with expected results. Lucca *et al.* address web-based application testing with an object-oriented web application model [28]. They outline a comparator which automatically compares the actual results against the expected values of the test execution. Our technique can be thought of as a working instantiation of such a design, and we extend the notion to structural differences.

Sneed explores a case study on testing a web-based application system for the Austrian Chamber of Commerce [43]. 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.

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-based applications [46–48]. They investigate features derived from `diff`, content, and structure, and refine these features into oracle comparators [48] 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 [47]. 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 [47]. 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-based applications [47, 48], our model uses features that we claim are closer to tree-based differences and human judgments in a holistic manner to train one generic 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 in Section 3.3 and Section 6 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.* [47], where a `diff` comparator would have no false positives for a deterministic application.

Although recent work has explored using abstract syntax tree matching [31] and semantic graph differencing [35]

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 [24]. 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. Differencing logical UML models is also an active research area [55], where both the lexical and structural differences between two documents are considered. SMART uses a similar approach to inspect changes between two versions of tree-based artifacts associated with software, but is used to differentiate between errors, rather than just intelligently summarize changes.

Binkley [11, 12] as well as Vokolos and Frankl [52] 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 differences 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.

Capture-replay scripts suffer from the "fragile test problem", where a robot user fails for trivial reasons [30]. 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 [30]. 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.

11 Conclusion

Despite their high reliability requirements and user-centric nature, the testing of web-based applications is often ignored due to a lack of time, resources, and a low perceived return on investment. Consequently, recent work on web-based applications focuses on automating as much of the testing process as possible. This paper presented a new technique that takes advantage of the special structure of XML/HTML output, as well as the underlying similarities between unrelated web-based applications, to provide a fully automated approach for regression test output comparison in this domain. In a study of open-source benchmarks totaling over 848,000 lines of code, our highly precise oracle comparator, SMART, was between 2 to 50 times more capable than `diff` at correctly labeling non-faults, saving developers significant effort. At the same time, SMART is highly effective at finding actual faults, obtaining perfect recall for three of our benchmarks. In an effort to explore the real-world impact using our approach, we tested SMART on three popular open-source PHP web-based applications that made heavy use of non-deterministic data. Overall we found that SMART was able to successfully flag 99% of the severe faults on average in these applications automatically, without relying on any manual annotation of test case output by developers.

Acknowledgments

This research was supported by, but does not reflect the views of, National Science Foundation Grants CCF 0954024, CCF 0916872, CNS 0716478, CNS 0627523 and Air Force Office of Scientific Research grant FA9550-07-1-0532, as well as gifts from Microsoft.

References

1. Apache Click, 2008. http://incubator.apache.org/click/.
2. GCC-XML. `http://www.gccxml.org/HTML/Index.html`, 2008.
3. Vqwiki open source project, 2008. http://www.vqwiki.org/.
4. A7soft jexamxml is a java based command line xml diff tool for comparing and merging xml documents. `http://www.a7soft.com/jexamxml.html`, 2009.
5. Open-realty. `http://www.open-realty.org/`, 2010.
6. Prestashop free open source e-commerce software for web 2.0. `http://www.prestashop.com/`, 2010.
7. Vanilla - free, open-source forum software. `http://vanillaforums.org/`, 2010.
8. 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.
9. Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In *World Wide Web Conference*, May 2002.
10. Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, Inc., 1999.
11. David Binkley. Using semantic differencing to reduce the cost of regression testing. In *International Conference on Software Maintenance*, pages 41–50, 1992.
12. David Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8):498–516, 1997.
13. Barry Boehm and Victor Basili. Software defect reduction. *IEEE Computer Innovative Technology for Computer Professions*, 34(1):135–137, January 2001.
14. 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.
15. Kinga Dobolyi and Westley Weimer. Harnessing web-based application similarities to aid in regression testing. In *20th International Symposium on Software Reliability Engineering*, November 2009.

16. Kinga Dobolyi and Westley Weimer. Modeling consumer-perceived web application fault severities for testing. Technical report, University of Virginia, 2009.
17. Kinga Dobolyi and Westley Weimer. Addressing high severity faults in web application testing. In *The IASTED International Conference on Software Engineering*, February 2010.
18. Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *International Conference on Software Engineering*, 2003.
19. Michael Ellims, Darrel Ince, and Marian Petre. The csaw c mutation tool: Initial results. pages 185–192, 2007.
20. 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.
21. Edward Hieatt and Robert Mee. Going faster: Testing the web application. *IEEE Software*, 19(2):60–65, 2002.
22. Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.
23. G. M. Kapfhammer. Software testing. In *The Computer Science Handbook*, chapter 105, 2004. CRC Press.
24. Srikanth Karre. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005.
25. 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.
26. R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.
27. 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.
28. G. Di Lucca, A. Fasolino, F. Faralli, and U. de Carlini. Testing web applications. *International Conference on Software Maintenance*, page 310, 2002.
29. 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.
30. Gerard Meszaros. Agile regression testing using record & playback. In *Object-oriented programming, systems, languages, and applications*, pages 353–360, 2003.
31. 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.
32. J. Offutt. Quality attributes of web software applications. *Software, IEEE*, 19(2):25–32, Mar/Apr 2002.
33. S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, December 2005.
34. R.S. Pressman. What a tangled web we weave [web engineering]. *IEEE Software*, 17(1):18–21, January/February 2000.
35. Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *International Conference on Software Maintenance*, 2004.
36. C. V. Ramamoothy and W-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.
37. Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Annals of Software Engineering*, 14(1-4):93–114, 2002.
38. Scott Rosenberg. What you don't know can cost you millions, July 2009. `http://www.cxoamerica.com/pastissue/article.asp?art=270091&issue=202`.
39. Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.
40. Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., 1986.
41. Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
42. Luis Moura Silva. Comparing error detection techniques for web applications: An experimental study. In *Network Computing and Applications*, pages 144–151, 2008.
43. Harry M. Sneed. Testing a web application. In *Workshop on Web Site Evolution*, pages 3–10, 2004.
44. Elizabeth Soechting, Kinga Dobolyi, and Westley Weimer. Syntactic regression testing for tree-structured output. In *International Symposium on Web Systems Evolution*, September 2009.
45. Fawzy Soliman and Mohamed A. Youssef. Internet-based e-commerce and its impact on manufacturing and business operations. In *Industrial Management & Data Systems*, 2003.
46. Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Automated Software Engineering*, 2005.
47. Sara Sprenkle, Emily Hill, and Lori Pollock. Learning effective oracle comparator combinations for web applications. In *International Conference on Quality Software*, 2007.
48. 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.
49. Jeff Sutherland. Business objects in corporate information systems. *ACM Comput. Surv.*, 27(2):274–276, 1995.
50. `http://txt2html.sourceforge.net/`. txt2html - text to HTML converter. Technical report, 2008.
51. Kenneth R. van Wyk and Gary McGraw. Bridging the gap between software development and information security. *IEEE Security and Privacy*, 3:75–79, 2005.
52. 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.
53. 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*, 2006.
54. Leigh Williamson. IBM Rational software analyzer: Beyond source code. In *Rational Software Developer Conference*. `http://www-07.ibm.com/in/events/rsdc2008/presentation2.html`, June 2008.
55. Zhenchang Xing and Eleni Stroulia. Differencing logical uml models. *Automated Software Engg.*, 14(2):215–259, 2007.