# Automated Oracle Comparators for Testing Web Applications

Sara Sprenkle*, Lori Pollock*, Holly Esquivel†, Barbara Hazelwood‡, Stacey Ecott§
* University of Delaware, {sprenkle, pollock}@cis.udel.edu
† University of Nebraska at Kearney, esquivelhm@unk.edu
‡ Xavier University, hazelwood@cs.xavier.edu
§ Tufts University, stacey.ecott@tufts.edu

## Abstract

*Software developers need automated techniques to help maintain the correctness of complex, evolving web applications. While there has been success in automating some of the testing process for this domain, there exists little automated support for verifying that the executed test cases produce expected results. We assist in this tedious task by presenting a suite of automated oracle comparators for testing web applications. To effectively identify failures, each comparator is specialized to particular characteristics of the possibly nondeterministic web applications' output in the form of HTML responses. We also describe combinations of comparators designed to achieve both high precision and recall in failure detection and a tool for helping testers to analyze the output of multiple oracles in detail. We describe an evaluation study of the effectiveness and costs of each individual and combination oracle comparator. We include recommendations to testers on automatic oracle comparators based on their application's characteristics.*

## 1 Introduction

With the prevalent use of web applications, even partial functionality loss can cost businesses millions of dollars per hour [5, 19]. The critical need for reliable web applications, coupled with their frequent modification, motivates the development of automatic techniques to enable extensive testing with a quick turnaround time. However, test automation goes beyond automatically creating and executing test cases [11, 23, 24] and reducing test suite size [13, 21]. Test automation should also seek to automate the tedious and error-prone task of interpreting the test results, that is, automating the test oracle's evaluation of the test case's actual results as pass or no pass. In this paper, we focus on the *test oracle comparator problem*: given a test case and expected results, *automatically determine whether the web application produces correct output*.

A test *oracle* produces an expected result for a test input and then uses a *comparator* to check the actual results against the expected results [4]. Automated software testing oracles have traditionally been difficult and expensive to develop [3, 4, 20, 26]. Human oracles often evaluate the correctness of the actual output. Some approaches to automated or partially automated oracles involve developing an oracle from specifications, from simulations of the program under test, or using a trusted implementation [4].

The major challenges to developing oracles for web applications are the difficulty of (1) accurately modeling web applications and (2) observing all their outputs. First, while some researchers propose using a model-based approach to oracles [1, 6, 18, 25], accurate models of web applications must address the ability to navigate web pages through a browser (e.g., a user accesses pages through back and forward buttons or by typing the URL directly); the inability to statically determine control flow due to user input and location; and the use of numerous technologies and languages. Second, web applications typically respond with dynamically generated documents written in HTML (Hypertext Markup Language)—the standard language for publishing web pages, regardless of the programming language(s) used to implement the web application—but may have additional outputs, such as generated email messages, internal server state, data state, or calls to external web services. These additional outputs are difficult to monitor, faults often manifest themselves in HTML output, and the user—often a customer or client—sees the HTML response [19]. Thus, we target our oracle comparators to the HTML output from web applications. Other web application testing approaches [9, 11] have also employed oracles that use the HTML responses as the output of an executed test case.

In HTML, authors specify both the content and presentation of their web pages [15], as shown in Figure 1. Authors use HTML *elements* to markup portions of the document's text in terms of the document's structure (e.g., the text's purpose), presentation (e.g., the text's appearance), or hypertext (e.g., links to other documents or parts of the doc-

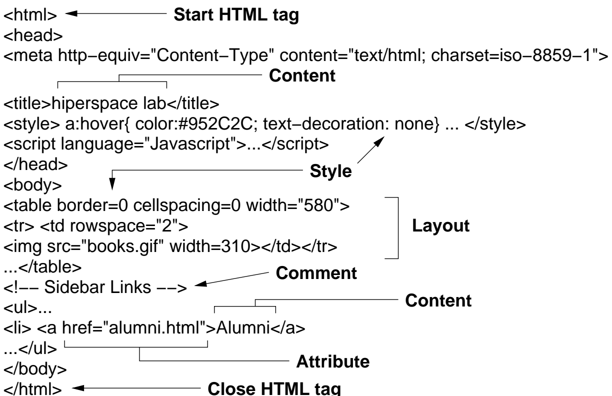


**Figure 1. Simplified Excerpt of HTML File**

ument). HTML elements usually have a start tag with zero or more attributes (name/value pairs) and may have an end tag and content enclosed between tags.

An oracle comparator that naively detects every difference between the actual and expected HTML output [7] could mistakenly report a fault when the difference lies in real-time, dynamic information. Reporting such *false positives* can lead to wasted developer effort tracking down nonexistent bugs. In contrast, an oracle comparator that focuses on specific internal details of behavior may miss reporting faults (*false negatives*), resulting in heavy penalties such as loss of consumer confidence and business revenues.

Validating an HTML response by comparing it with the expected HTML response is particularly difficult because acceptable differences between expected and actual results are not easily generalized and may depend on the application or specific response. Further complicating the process is that important tags may not be structurally tied to their content (e.g., the label for an input field cannot be easily mapped to the `input` tag), changes in whitespace do not matter with respect to what the user views, and some components are case-insensitive while others are not. Finally, real-time or nondeterministic behavior in the web application can occur, for example, when the behavior depends on the current time or when items are pulled randomly from a database and displayed in the web page.

Our previous experimental studies compared replay techniques using four basic oracle comparators [23] and confirmed that some oracles provide high *precision* (i.e., their reported differences were indeed due to faults) for certain kinds of faults but undesirable *recall* (i.e., had some false negatives and did not report all faulty behavior). Other oracles revealed more failures (high recall) but suffered in precision by reporting output differences that were not due to faults. These prior results led us to the work reported in this paper, investigating (1) how the various components of an HTML document should be handled by an oracle comparator that seeks to have both high precision and high recall, (2) what changes to a document are important for revealing faults, (3) when these changes are important, and (4) how oracles targeted to specific changes can be combined to create oracles that yield both high precision and recall. This paper's main contributions beyond the state of the art and our previous work in testing web applications are

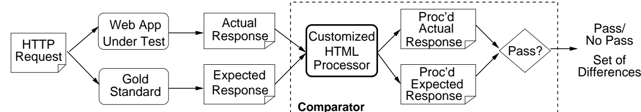


**Figure 2. Oracle Process**

- a suite of 22 automated oracle comparators specialized to particular characteristics of the possibly nondeterministic web applications' HTML output and carefully selected oracle combinations that improve upon the individual comparators' effectiveness,
- results and analysis from two experimental studies of the effectiveness and costs of individual and combination oracle comparators, and
- recommendations to testers on choosing test oracle comparators for testing their web application.

The remainder of our paper is organized as follows: Section 2 presents our suite of oracle comparators, while Section 3 describes the comparators' implementation and integration into a tool that helps testers analyze the output of multiple comparators in detail. In Section 4, we describe our evaluation methodology and present and analyze our results in Section 5. We present related work in Section 6 and conclude with future work in Section 7.

## 2 HTML-based Oracle Comparators

We developed a suite of automated oracle comparators that validate HTML responses, as shown in Figure 2. The comparators take as input the actual HTML response (document) generated by the web application upon an HTTP request[1] and the expected HTML response from a previous, working version of the web application (before a code update) with the same HTTP request. Each comparator performs customized processing on the HTML responses and a comparison of the processed HTML responses. If there are no differences between the processed responses, the comparator says the request *passes*. If the request did not pass, the comparator outputs a set of differences used to reveal

[1] An HTTP request includes the type of request (typically GET or POST), the path to the requested resource, and optional data, such as name-value pairs for a form.

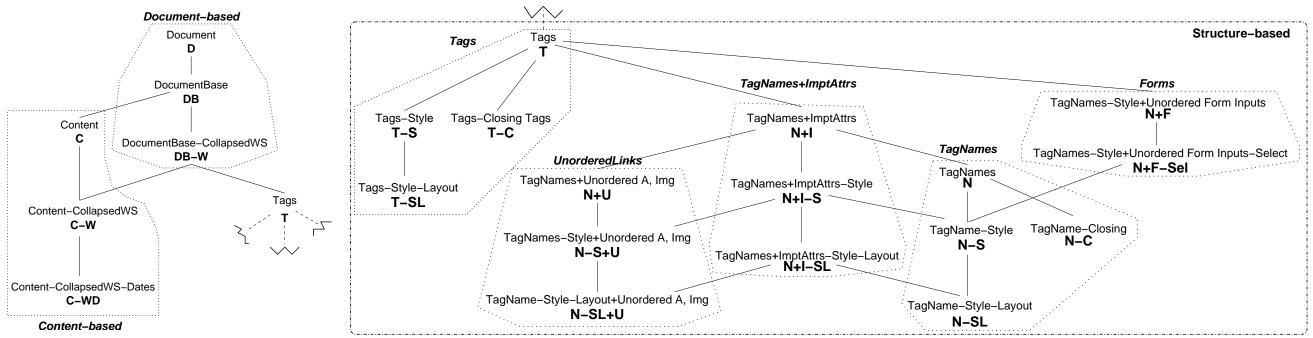


**Figure 3. Partial Ordering of Implemented Oracle Comparators**

potential faults in the current version. Execution of an oracle comparator is repeated for each HTTP request in the test case. If any request in the test case fails, the test case fails. Since the comparators use the output of a previous, working version of the web application as the *gold standard* [4], they are *consistent* oracles [14], which are fast but will not reveal faults that exist in the previous version.

The full suite of comparators that we implemented is presented in Figure 3 as a partial ordering based on the information from the HTML document that each oracle uses. Oracle *A* is a child of oracle *B* in the partial ordering if oracle *A* uses a subset of the information in the HTML document used by oracle *B*. Many different oracle comparators could be created based on various HTML characteristics; we have focused on oracles that, in our experience, have the most potential to provide high precision and recall in reporting differences that indicate faults.

A naive automatic comparator, which we call the **Document** comparator, identifies all the differences between the actual and expected HTML documents as a whole [21], essentially using the utility `diff`. **Document** will reveal any failure that appears in the HTML. However, **Document** is susceptible to false positives from changes in either the document's *content* (dynamic behavior, such as changes in the date or time) or its *structure* (i.e., display or appearance of the document).

The family of **Document**-based oracles examines the full document but ignores elements in which differences do not greatly affect the user or are best handled by other testing techniques. First, we can ignore HTML comments. Second, since `meta` elements contain information about the document, such as keywords for search engines, that do not affect the user's experience with the web page, we can safely ignore meta elements. Furthermore, since we do not focus on techniques for testing client-side scripting, we can ignore programming elements, such as `script`, `noscript`, `applet`, and `object`. Elements, such as `noframes`, are special directives for older browsers that cannot handle that functionality. We call the comparator that uses the entire document but ignores comments, meta, programming, and special-directive elements the **DocumentBase** comparator because it is the ancestor of all other implemented oracle comparators. Our final document-based comparator **DocumentBase-Collapsed WhiteSpace** collapses whitespace because browsers collapse whitespace, and differences in whitespace are not meaningful differences to the user.

Beyond the **Document**-based oracles, there are two main classes of oracle comparators: *content-based* and *structure-based*. We developed three **Content**-based comparators that ignore structure, i.e., the HTML tags. The **Content** oracle [23] does not falsely indicate failures for web pages with simple changes to the UI, e.g., changes in the page layout (tables versus CSS) or display (font color), which do not affect the application's behavior. However, **Content** will falsely suggest application failures upon changes in the generated dynamic output, such as changes in the current date. Besides missing faults not manifested in the HTML response, **Content** will not reveal faults in the HTML tags, such as omitted form input tags. The two refinements to the **Content** comparator are **Content-CollapsedWhitespace** because a user cannot see whitespace changes in the browser and **Content-CollapsedWS-Dates** because displayed dates may indicate the date/time that some event occurred, and the displayed dates will not match when the test cases to produce the expected and actual results were not executed at the same time.

The **Structure**-based oracles target only the HTML tags of the actual and expected output [23]. Faulty behavior may manifest itself in tag-based content (such as forms) or may affect the structure of a page (e.g., missing rows in a table). This comparator class will incorrectly suggest failures in pages that have slight changes in the UI that do not affect the behavior of the application. For example, the comparator will flag changes in presentation from paragraph (`p`) to line breaks (`br`) tags. The comparator will miss faults that

manifest themselves in the response's content.

Because HTML is not well-structured and browsers are robust, there is variability in the tag changes that indicate a fault. The following is a list of observations about tag properties (as shown in Figure 1) that a tester may want an oracle comparator to ignore:

P1. *closing tags:* Since the browser closes some tags implicitly when other tags start, a page that contains a difference in closing tags still may be rendered the same.

P2. *layout tags (e.g., div, span, table) or attributes (e.g., align, width):* Since layout mostly affects appearance and the style directives may be in a separate CSS file, differences in layout are less severe failures that a human may need to verify in multiple browsers.

P3. *style in tags (e.g., b, pre) or attributes (e.g., bgcolor):* Style mostly affects appearance, which is a less severe failure, and humans will need to verify the page's appearance in multiple browsers.

P4. *script references in attributes:* Since we do not verify the scripts, changes to references are not meaningful for an oracle comparator.

P5. *ordering of images and links:* When applications dynamically generate pages containing images or links, changes to the links or images or their order may be insignificant—only which images or the number of links may matter.

P6. *ordering of form input:* Dynamically generated forms may change the order of input tags or options, but if the form contains all the inputs and options, it is correct.

P7. *selection in forms:* A change to the default selected or checked option in a form may not indicate a fault, as long as the user can select the appropriate option.

Based on these observations, we created several classes of Structure-based comparators: **Tags**, **Tag Names**, **TagNames+ImptAttrs**, **UnorderedLinks**, and **Forms**. **Tags** examines all HTML tags, **Tag Names** addresses P2-P7 and examines only the names of the HTML tags, and **TagNames+ImptAttrs** (P2-P4, P7) focuses on the names and the *important* attributes of each HTML tag. For the important attributes, we selected the *required* attributes from the HTML specification [15] and used our intuition to add other important attributes, such as the name, type, and value for input tags. When appropriate, we assigned a missing attribute its default value, such as "text" for the input's type. We designed specialized comparators that handle links/images and forms. Both specialized comparators first examine the documents' tag names for differences to identify major failures. The **UnorderedLinks** class of comparators (P5) compares an alphabetized list of anchor (links) and image tags with their important attributes (href, name, and target and src, respectively). The **Forms** class of comparators (P6) compares an alphabetized list of input tags and option tags (grouped by the select tags) with their important attributes (type, name, and value and name, respectively). The **Forms** comparator includes the `selected` and `checked` attributes, while **Forms-Select** (P7) ignores them. For each class, we can include or ignore closing (P1), layout-based (P2), or style-based (P3) tags.

***Expected Tradeoffs.*** We expect that, in terms of false positives and false negatives, the oracle comparators will follow the partial ordering: *comparators at the top of the hierarchy will have higher recall, reporting the fewest false negatives, but with lower precision, reporting the most false positives*. The amount of information that an oracle examines is a double-edged sword: the more information, the more potential to reveal faults or mistake irrelevant differences as failures. In general, a comparator's cost is a lesser concern than its effectiveness. The proposed comparators are essentially equivalent in terms of space costs: they all require the same actual and expected HTML responses. Oracles, such as **Forms** and **UnorderedLinks**, require more processing of the responses, which will increase the execution cost.

## 3 Implementation and Tool Support

We implemented our oracle comparators primarily using Perl's HTML Parser class and the utility `diff`. We integrated our comparators into a tool we developed that helps testers evaluate test suite results in detail [22]. Our tool provides a means for managing and analyzing large test suites for web applications, with the goal of identifying the test cases and corresponding results associated with potential faults. When a user navigates to a request, the tool automatically executes each oracle on the responses and identifies which comparators report a failure. The tool is especially useful for comparing results from multiple comparators—to evaluate them for false positives, false negatives, and their limitations. Except for **Forms** and **UnorderedLinks**, which required minor tweaking, our comparators required no changes to integrate them into the tool.

## 4 Experimental Methodology

We designed our experimental study to answer the fundamental question: **How effective is each oracle comparator at detecting failures?**[2] To thoroughly answer this question, we sought to answer the following research questions:

1. How many failures does each oracle correctly identify?
2. How many failures does it miss (false negatives)?
3. How many responses are incorrectly identified as failures (false positives)?

[2] We do not evaluate the detection of unmet time/space-constraint failures.

| Apps | Classes | Methods | Statements | NCLOC |
|---|---|---|---|---|
| Masplas | 9 | 42 | 441 | 999 |
| Book | 11 | 385 | 5250 | 7791 |
| CPM | 75 | 172 | 6966 | 8947 |
| DSpace | 274 | 1453 | 27136 | 49513 |

**Table 1. Subject Application Characteristics**

4. What types of faults does each oracle reveal?
5. What are the costs of each oracle comparator in terms of its requirements for time and space?

### 4.1 Independent and Dependent Variables

The **independent variable** is the HTML oracle comparator. We evaluated all 22 comparators described in Section 2. The **dependent variables** are the comparators' accuracy (measured in terms of precision, recall, false positives, and false negatives) and their time and space costs. *Precision* is the ratio of the number of correctly identified HTML responses that exhibit failures to the total number of reported responses, while *recall* is the ratio of the number of correctly identified HTML responses containing failures to the expected number of responses containing failures. Higher precision means fewer false positives; higher recall means fewer false negatives.

### 4.2 Subject Applications and Test Suites

We used four subject applications of varying sizes (1K-50K non-commented lines of code), technologies, and representative web application activities: a conference website (Masplas); an e-commerce bookstore (Book) [12]; a course project manager (CPM); and a customized digital library (DSpace) [10]. Table 1 summarizes the applications' code characteristics. We generated test suites by deploying each subject application, collecting user accesses to each application, and then converting the user accesses into our test suite of user sessions [21].

**Masplas** is a web application in which users can register for a workshop, upload abstracts and papers, and view a schedule, proceedings, and other related information. Masplas is implemented in Java and JSP with a MySQL database. We collected user accesses during the submission and registration periods for MASPLAS 2005.

**Book** allows users to register, login, browse for books, search for books by keyword, rate books, add books to a shopping cart, modify personal information, and logout. Book uses JSPs and a MySQL database. To collect user accesses for Book, we sent email to local newsgroups and posted advertisements in the University's classifieds web page asking for volunteer users.

With **CPM**, course instructors and teaching assistants can manage group work in a course, including recording grades and coordinating demonstration times. CPM is written using Java and JSPs that access a filestore backend. We collected user accesses from instructors, teaching assistants, and students using CPM during five academic semesters from 2004-05 at the University of Delaware.

**DSpace** is a customized digital publications library based on an open-source digital repository system [10]. DSpace automatically generates sorted publications pages from a database that research group members maintain through a web application interface. A user can create dynamic views of publications by searching with various criteria and can download publications in various formats. DSpace is written in Java and JSP and uses a PostgreSQL database. We collected user accesses after publicizing our digital library in August 2005 through May 2006.

Table 2 shows characteristics of the test suites for each application. The number of requests is the total number of *valid* requests, including requests to statically and dynamically generated resources and excluding requests for images, JavaScript, or style sheets. We show both the number of requests and the number of HTML responses because, in DSpace, the response is sometimes the publication itself in various formats, and a simple `diff` will suffice. The test suites do not completely cover the application code because users do not access some error and administrative code and code for alternative configurations.

### 4.3 Methodology

We performed two experiments—one to analyze the effect of nondeterministic application behavior on the oracle comparators and one to analyze the effectiveness of the oracle comparators in the presence of seeded faults in the subject applications—within the experimental framework described in our previous work [23]. The results of the first study allowed easier identification of false positives in the second experiment. During each experiment, we also measured each comparator's execution time.

**Nondeterminism/Real-Time Behavior Effect.** This experiment's goal was to identify the applications' nondeterministic and real-time behavior and analyze its effect on the comparators' effectiveness. To examine possible nondeterminism between different runs of the same test suite, we executed each test suite nine times over several months on the same clean version of the application. Since application behavior may depend on the current date or time, we wanted to expose those differences—as well as the applications' nondeterministic behavior—by executing the test suite on different days, months, and even years (2006, 2007). We then ran each comparator pairwise on the responses from the nine executions of each test suite, totaling 36 outputs for each comparator per application[3]. Because the application

[3] We chose nine executions and 36 suite-level outputs by the comparator

| Apps | Test Input # Test Cases | # Requests | % Stmt Coverage | Test Output # HTML Pages | Avg HTML Page Size |
|---|---|---|---|---|---|
| Masplas | 169 | 1103 | 90.5% | 1103 | 3.8 KB |
| Book | 125 | 3640 | 56.8% | 3564 | 10.7 KB |
| CPM | 890 | 12352 | 78.4% | 12352 | 2.0 KB |
| DSpace (Exp 1/Exp 2) | 1800/75 | 22129/3183 | 65.6%/51.6% | 17892/3023 | 8.6 KB/12.5 KB |

**Table 2. Test Suites and Responses**

| Apps | Seeded Faults | Exposed Faults | Categories of Exposed Faults: Logic | Data | Form | Appearance | # HTML Pages Exposing Each Fault: Median | Mean | Std Dev |
|---|---|---|---|---|---|---|---|---|---|
| Masplas | 28 | 22 | 17 | 4 | 2 | 2 | 35.5 | 54.9 | 55.9 |
| Book | 39 | 36 | 17 | 11 | 4 | 6 | 595 | 521.2 | 629.3 |
| CPM | 135 | 96 | 75 | 27 | 19 | 6 | 25 | 202.6 | 516 |
| DSpace | 50 | 20 | 19 | 2 | 2 | 1 | 104 | 282.6 | 347.4 |
| **Total** | 252 | 174 | 128 | 44 | 27 | 15 | N/A | N/A | N/A |

**Table 3. Seeded and Exposed Faults**

does not contain faults during this experiment, any failures that an oracle comparator reports are false positives.

**Failure Detection Effectiveness.** We analyzed each oracle comparator's effectiveness at revealing faults. To create the faulty versions of each subject application, graduate and undergraduate students familiar with JSP, Java servlets, and HTML manually seeded faults into each subject application. We also seeded naturally occurring faults that were discovered by users during application deployment into Masplas, CPM, and DSpace. In general, four types of faults were seeded—data store (faults that exercise application code interacting with the data store), logic (application code logic errors in the data and control flow), form (e.g., modifications to name-value pairs and form actions), and appearance (faults that change the way the page appears). Since these categories are not mutually exclusive, a fault can be classified into multiple categories. Table 3 reports the number and types of seeded and exposed faults. Since some seeded faults were not exposed—either the test suite did not execute the seeded faults or the faults did not manifest in the HTML responses, we report the number of seeded and exposed faults separately and only analyze the comparator results for exposed faults.

We replayed the test suite *with_state* [23], where application state is restored before replaying every test case in the suite on the fault-seeded versions of each subject application. By replaying *with_state*, test suite execution on the faulty versions will match the clean execution more closely, and faults that manifest themselves in the application's state will not propagate in the state for subsequent test cases to expose. If we allowed faulty state to propagate, the responses would diverge from the clean responses more, and failures would become trivial to detect. Replaying *with_state* reduces the number of responses that may display a failure, making it more difficult for comparators to detect failures and easier to differentiate between the comparators. We then executed each comparator, comparing the test suite's responses with the clean (the expected results) and fault-seeded versions (actual results) of the application. Using our tool, we manually determined the *expected* pass/no pass results for the ideal oracle by checking if each response exhibited a failure; Table 3 summarizes the pages that the manual oracle determined were no pass. Finally, we compared the expected oracle results with each comparator's results to measure the comparator's correct answers, false positives, and false negatives.

to obtain a good sample size for indicating nondeterminism.

## 4.4 Threats to Validity

Several factors may affect the interpretation of the results of this study. Since we performed our study with four applications, a study with additional applications may be necessary to generalize the results; however, we chose subjects with different technologies, implementers, and application and usage characteristics to help reduce the threat to generalizing our results. We believe that our oracle comparators are general and will handle most web applications and their HTML output; however, we designed our oracle comparators with knowledge of how our subject applications behave. Since HTML is not well-structured, the choice of HTML parser affects the behavior of the comparators; an oracle implemented using a different parser or language may yield slightly different results. Furthermore, since browsers may parse and display HTML differently, a comparator may report a page as containing a failure, but the page may display and function correctly in other browsers. We include faults that emulate some of these problems. Since we used the same HTML parser to implement each oracle, we do not believe any of these implementation differences significantly affect our experiments' validity.

Each application's test suite was generated from user accesses to the deployed application. The test cases do not completely exercise the applications or expose all seeded

faults, non-deterministic behavior, and real-time behavior. However, our test cases do exercise a large portion of each application and expose interesting application behavior.

Manually seeded faults may be more difficult to expose than naturally occurring faults [2]. Although we tried to model the seeded faults as closely as possible to naturally occurring faults—even including naturally occurring faults from previous deployments, some of the seeded faults may not represent natural faults. Because the faults may not be realistic, we may bias the results to some comparator because it does or does not detect those types of faults. Since we only consider the number of HTML pages containing manifestations of the fault detected/missed by the oracles and not the severity or detectability of the faults detected/missed, the conclusions of our experiment could be different if the results were weighted by fault severity or detectability. We also do not analyze sources of false positives beyond non-determinism or real-time behavior.

Since our experiments ran for several days or weeks, the comparator timings were slowed down by other processes, such as network activity and daily backups, but these processes should affect all comparator timings equally.

# 5 Results and Analysis

We present the experimental results and analysis for each experiment in this section. For each graph, the box represents 50% of the data and spans the width of the inner quartile range ($IQR$), with each whisker extending $1.5 * IQR$ beyond the top and bottom of the box. The center horizontal line within each box denotes the median false positives reported, $+$ represents the mean, and $\circ$ represents an outlier. We abbreviate the comparator names as shown in Figure 3.

## 5.1 Nondeterminism/Real-Time Behavior Effect

Figure 4 shows the oracle comparators' normalized false positive results from running each comparator on the HTML responses from executing test suites on the clean applications (answering question 3). Since Masplas and Book showed no differences in the responses for even the naive **Document** comparator (i.e., exhibited only deterministic behavior), we do not include their results. For each comparator in Figure 4, CPM is the light gray bar on the left and DSpace is on the right in dark gray. Recall that any failure (difference in expected and actual processed responses) reported by an oracle is a false positive because the application is the clean version. To normalize the results so that we can compare across applications, we divide a given comparator's reported number of failures by the number of failures reported by **Document** (i.e., total number of differences between suites' responses).

On average, **Document** reported 90.5 differences in CPM's HTML responses between test suite executions out of 12,352 responses, with a standard deviation of 26.69. The sources of differences were dependence on the current date, the order that a hashtable of a form's hidden name-value pairs was printed out, and tie-breaking when more than one demo had the same modified date. As the time between executions increased, the number of differences in responses increased due to time dependence in one servlet.

For DSpace, **Document** reported an average of 1183 differences in HTML responses between test suite executions out of 17892 HTML responses with a standard deviation of 419. The difference in when the suite executed did not have as much of an effect on DSpace, except when the suite executed in different years because users could select to view publications from an additional year (2007). Most of the nondeterministic behavior came from tie-breaking the order of publications with the same publication date or the order of the associated documents (such as the paper and presentation slides).

As expected, the comparators' false positives follow the partial ordering: the higher the comparator is in the hierarchy, the more reported false positives. The **Document**-based comparators are most susceptible to reporting false positives because they consider the entire document, while the other comparators only consider subsets of the document. In CPM, the **TagNames** and **UnorderedLinks** comparators reported no failures, and **Tags+ImptAttrs** reported less than 5% of the responses with differences as failures. All comparators reported some failures in DSpace because, for some responses, DSpace randomly displays between 3 and 5 publications from a collection. None of our comparators can handle such randomness.

Since some nondeterminism only manifests itself in either the response's structure or content, the **Tags** and **Content**-based oracles' performance varied greatly. In CPM, **Tags** had the same false positives as **Document**, while in DSpace, **Content** had nearly as many false positives as **Document**. The **Content**-based oracles had false positives in DSpace for a publication's upload time and changes in publication or document orders and in CPM for changes in the demo order. Ignoring dates (**C-WD**) greatly reduced failures reported for DSpace. **Forms** had many false positives in CPM because it reported changes in a form's default selected value as failures; **Forms-Select** did not. Removing style tags reduced the number of failures reported for DSpace because of how DSpace interleaves style tags in the publications.

## 5.2 Failure Detection Effectiveness

We analyzed the results of the comparators' failure detection effectiveness by individual application, by appli-

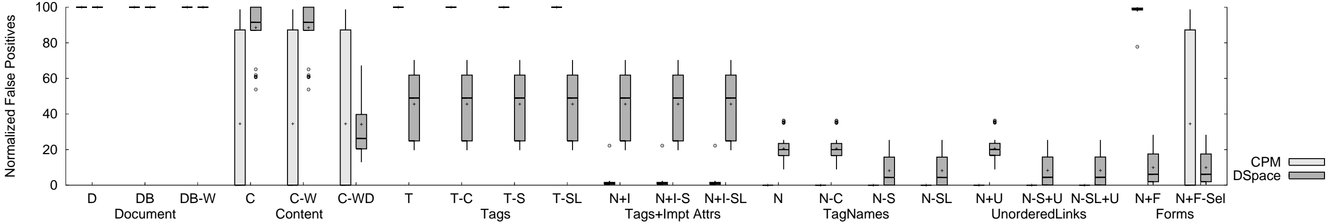


**Figure 4. Effect of Nondeterminism/Real-Time Behavior on Comparators**

cation behavior (deterministic or nondeterministic), across all applications, and by fault category. Due to space constraints, we only show the comparators' effectiveness results in terms of precision (light gray, left) and recall (dark gray, right) for all faults across all applications (Figure 5) to answer questions 1-4.

***Deterministic Applications.*** Since Book and Masplas exhibit only deterministic behavior, all comparators had consistently high precision. Only **Document** and **DocumentBase** had false positives, which were whitespace changes as a side effect of a fault that commented out JSP code. The **Document**-based comparators had the best recall with high consistency for all fault types because they identified failures in both the document's structure and content. Across all faults, the **Document**-based comparators had the best mean effectiveness, as measured by the harmonic mean of precision and recall.

***Nondeterministic Applications.*** The **Document**-based comparators have the lowest precision and the highest recall for the nondeterministic applications. **Forms-Select**, the **TagNames** comparators, and the **UnorderedLinks** comparators have near-perfect precision consistently, which differs from the previous experiment because, in that experiment, test suites were executed over a longer period of time. Overall, **Forms-Select** has the best mean effectiveness (.86), as measured by the harmonic mean of precision and recall, followed by **Tags+ImptAttrs** and **Tags+ImptAttrs-Style** (both .80).

***Form Faults.*** Surprisingly, the **Tags** comparator family had the best mean effectiveness for form faults across all applications. However, **Forms-Select** had equivalent mean precision to **Tags**. **Forms-Select** has the best overall effectiveness for the nondeterministic applications. As expected, **Content**, **TagNames**, and **UnorderedLinks**-based oracles only detected a few form failures.

***Appearance Faults.*** As expected, content-based comparators and structure-based comparators that ignored attributes and style and layout tags had the most difficulty detecting failures from appearance faults. For all applications, the **Document**-based comparators and the **Tags** comparators consistently exposed nearly all the responses with faulty appearance behavior.

Since logic and data faults frequently overlap and are the majority of the seeded faults, thus following the trends in Figure 5, we do not include a separate detailed analysis.

## 5.3 Costs

Since all the comparators require the actual and expected HTML responses (the sizes of which are in Figure 2), we ignore space costs and focus on execution time to answer question 5. At the extremes, **Document** is the fastest comparator on average, while the **UnorderedLinks** comparators are the slowest. The other comparators fall in between these two, and the differences between execution times are relatively small. As expected, the size of the HTML responses affects each comparator's execution time; the per-response execution time of the comparators is longer on Book and DSpace than on Masplas and CPM. Due to space constraints, we do not show the comparators' execution costs; however, on our largest test suite (DSpace with 17,892 pairs of HTML responses to compare, totaling 150 MB per suite), the slowest comparator, **UnorderedLinks**, executed in 14 minutes on average, whereas replaying the suite takes about 90 minutes.

## 5.4 Analysis

***Similarity of Oracles.*** If the results from a set of comparators are significantly similar, testers can safely choose one representative from the set. We measured similarity between two comparators as $1 - \%$ difference, where $\%$ difference is

$$\frac{\#\ of\ responses\ that\ oracles\ differed\ in\ pass\ results}{\#\ of\ responses\ that\ Document\ reported\ as\ no\ pass}$$

We divided by **Document**'s reported no-pass outputs rather than the total number of responses because the majority of responses from the faulty versions were the same as the responses from the clean version. We examined the comparators' output similarity by subject application and in aggregate. Their similarity varied slightly with the application, but the majority of the results were as we expected. The **Tags**, **TagNames**, **UnorderedLinks**, and **Forms** comparator classes produced the same results over 95% of the time,

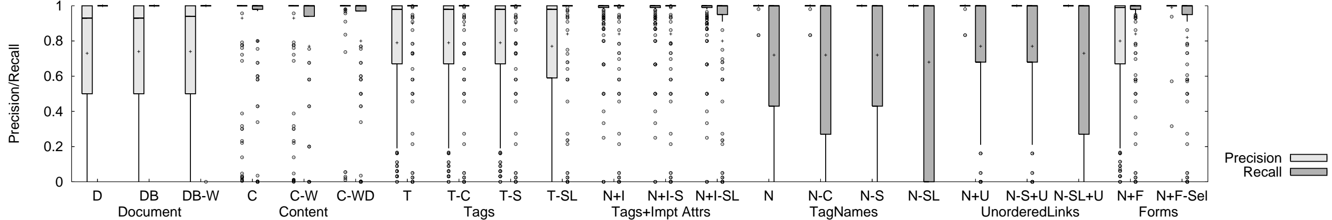


**Figure 5. Comparators' Precision and Recall in Failure Detection**

| Oracle Combination | Average F-Measure Oracle1 | Oracle 2 | O1 ∪ O2 |
|---|---|---|---|
| N+I ∪ C-WD | .83 | .79 | .91 |
| N+I-S ∪ C-WD | .83 | .79 | .91 |
| N+I-SL ∪ C-WD | .79 | .79 | .91 |
| N+F-Sel ∪ C | .83 | .75 | .87 |
| N+F-Sel ∪ N+I | .83 | .83 | .84 |
| N+I ∪ N+U | .83 | .78 | .84 |
| N+I ∪ N-SL+U | .83 | .74 | .84 |

**Table 4. Oracle Combinations' Effectiveness**

and the **Document**-based comparators produced the same results over 90% of the time. The **Tags+ImptAttrs** and **Content** comparators were surprisingly similar (99%) since they examine mutually exclusive parts of the response.

***Combining Comparators.*** Since there are tradeoffs between oracle comparators' precision and recall, we investigated whether we could create a more effective oracle by combining oracles. Rather than combining all pairs of oracles, we chose to combine oracles that were complementary in terms of the partial ordering (based on information used from the HTML response, as illustrated in Figure 3) and dissimilar in pass/no pass output, and we used our intuition about which combinations would prove most effective at achieving both high precision and high recall.

We combined two comparators by either unioning or intersecting their pass/no pass results. By unioning two comparators, we improve recall (reduce the number of false negatives), while the number of false positives could increase or decrease. If instead we intersect comparators' results, we reduce the number of false positives because both comparators must agree that the response contains a failure, thus improving precision, but also reduce the number of responses recognized as exhibiting faulty behavior when only one comparator detects the behavior, which decreases recall. The combined oracle's execution cost is the total cost of executing the constituent comparators.

Table 4 shows the best oracle combinations in terms of the average f-measure (the harmonic mean of precision and recall) across all applications. The union of **Content-CollapsedWhiteSpace-Dates** (C-WD) with variants of **TagNames+ImptAttrs** (N+I) yields the best comparator for all applications. For deterministic applications, **Document**-based oracle comparators still perform best because they report few false positives while detecting most faulty behavior, but the **Content-CollapsedWhiteSpace-Dates** unioned with variants of **TagNames+ImptAttrs** is the next best. As expected, the intersection combinations improved precision at the expense of recall.

***Recommendations to testers.*** Testers should first execute their test suite several times on the clean version of the application to (1) expose the responses that have nondeterministic and real-time application behavior and (2) identify the comparators with the fewest false positives so that the tester can then use the best comparators as a baseline when looking through the failure detection results. Even if the application has nondeterministic behavior, some responses may be deterministic. The tester should apply appropriate comparators to responses based on knowledge of the response's behavior. If the response is deterministic, the tester should use one of the **Document**-based comparators. If the response has nondeterministic behavior, she should use the combined **Content-CollapsedWS-Dates** and **TagsNames+ImptAttrs** comparator, which has the highest mean effectiveness without a significant increase in execution time.

A tester may want to use a more precise oracle in early testing stages to flesh out severe bugs without needing to sift through many false positives. If a tester is most concerned about the comparator reporting the fewest false negatives, she should use one of the **Document**-based comparators. If false positives are a major concern, the tester should use **Forms-Select**, which has the highest recall of the comparators with perfect precision. Depending on what types of faults the tester is looking for, he can choose to include or ignore tags and attributes that affect style and layout.

## 6 Related Work

Oracles are a difficult and important research problem [3, 14]. Researchers have worked on pseudo oracles [26], automating oracles [17, 18, 20], and creating specialized oracles for specific domains using model-based techniques [1, 6, 8, 9, 17, 18, 25]. Our comparators are automated, consistent pseudo oracles (focused on a subset of the web applications' output—HTML responses) that do

not require a specialized model. Other groups have mentioned using HTML-based oracles [9, 11] but provided few details on the comparator or experimental investigation into the accuracy of their oracles in the context of nondeterministic behavior. Since many web applications have a database backend, we could use a state validator, such as the one in AGENDA [8], to validate the application's database state. However, web applications may use other state, such as files or internal server state. Using the HTTPUnit [16] framework, testers manually create oracles for each generated web response. We assume developers use HTTPUnit for unit testing, but they need more general oracles for system, integration, and regression testing. One of testing's greatest challenges is interpreting the voluminous test results [14]; to our knowledge, we developed the only tool that aids web application testers in analyzing and evaluating test results of multiple automated comparators.

## 7 Conclusions and Future Work

We developed a suite of 22 HTML-based automated oracle comparators for testing web applications and evaluated the comparators and combinations of comparators on four subject applications in their effectiveness at detecting faulty behavior. We found that the most appropriate comparator depends on the application's behavior and the fault. We also made recommendations about the comparator or combination of comparators to use under various circumstances to achieve the best failure detection precision and recall.

There are several directions for future work. Since there are many permutations of what HTML features are important (more than our 22 implemented comparators), we want to develop a fully customized comparator so that a tester can choose which features to ignore or include based on her knowledge of the application. We also want to continue investigating comparator combinations by combining more than two comparators or using different operations to combine the comparators to find the most effective comparator.

## References


[1] A. Andrews, J. Offutt, and R. Alexander. Testing web applications by modeling with FSMs. *Software Systems and Modeling*, 4(2), April 2005.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Int'l Conf. on Software Engineering*, 2005.

[3] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR01-02, University of Oregon, 2001.

[4] R. Binder. *Testing Object-Oriented Systems*. Addison Wesley, 2000.

[5] M. Blumenstyk. Web application development-bridging the gap between QA and development. http://www.stickyminds.com, 2002.

[6] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *1996 SPIN Workshop*, Aug. 1996.

[7] S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Int'l Conf. on Management of Data*, May 1997.

[8] D. Chays, Y. Deng, P. Frankl, S. Dan, F. Vokolos, and E. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 14:17–44, Mar. 2004.

[9] G. DiLucca, A. Fasolino, F. Faralli, and U. D. Carlini. Testing web applications. In *Int'l Conf. on Software Maintenance*, October 2002.

[10] DSpace Federation. http://www.dspace.org, 2007.

[11] S. Elbaum, G. Rothermel, S. Karre, and M. Fischer II. Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3):187–202, May 2005.

[12] Open source web applications with source code. http://www.gotocode.com, 2003.

[13] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering Methodology*, 2(3), 1993.

[14] D. Hoffman. A taxonomy for test oracles. Quality Week '98, http://www.softwarequalitymethods.com, 1998.

[15] HTML 4.01 Specification. http://www.w3.org/TR/html4, 2007.

[16] HttpUnit. http://httpunit.sourceforge.net, 2007.

[17] A. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Int'l Conf. on Automated Software Engineering*, Oct. 2003.

[18] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Int'l Symp. on Foundations of Software Engineering*, Nov. 2000.

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

[20] D. Richardson. TAOS: testing with analysis and oracle support. In *Int'l Symp. on Software Testing and Analysis*, 1994.

[21] S. Sampath, V. Mihaylov, A. Souter, and L. Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *Int'l Conf. on Automated Software Engineering*, Sep. 2004.

[22] S. Sprenkle, H. Esquivel, B. Hazelwood, and L. Pollock. WebVizOr: A visualization tool for analyzing test results of web applications. Technical Report 206-335, University of Delaware, 2007.

[23] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and fault detection for web applications. In *Int'l Conf. on Automated Software Engineering*, Nov. 2005.

[24] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. A case study of automatically creating test suites from web application field data. In *Workshop on Testing, Analysis, and Verification of Web Services and Applications*, July 2006.

[25] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *Int'l Journal on Software Tools for Technology Transfer*, 4(1), Oct. 2002.

[26] E. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4), 1982.