This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Developing a Single Model and Test Prioritization Strategies for Event-Driven Software

Renée C. Bryce, Sreedevi Sampath, *Member*, *IEEE Computer Society*, and Atif M. Memon, *Member*, *IEEE Computer Society*

Abstract—Event-Driven Software (EDS) can change state based on incoming events; common examples are GUI and Web applications. These EDSs pose a challenge to testing because there are a large number of possible event sequences that users can invoke through a user interface. While valuable contributions have been made for testing these two subclasses of EDS, such efforts have been disjoint. This work provides the first single model that is generic enough to study GUI and Web applications together. In this paper, we use the model to define generic prioritization criteria that are applicable to both GUI and Web applications. Our ultimate goal is to evolve the model and use it to develop a unified theory of how all EDS should be tested. An empirical study reveals that the GUI and Web-based applications, when recast using the new model, show similar behavior. For example, a criterion that gives priority to all pairs of event interactions did well for GUI and Web applications; another criterion that gives priority to the smallest number of parameter value settings did poorly for both. These results reinforce our belief that these two subclasses of applications should be modeled and studied together.

Index Terms—Combinatorial interaction testing, covering arrays, event-driven software (EDS), t-way interaction coverage, test suite prioritization, user-session testing, Web application testing, GUI testing.

1 INTRODUCTION

EVENT-DRIVEN software (**EDS**) is a class of software that is quickly becoming ubiquitous. All EDSs take sequences of events (e.g., messages and mouse-clicks) as input, change their state, and produce an output (e.g., events, system calls, and text messages). Examples include Web applications, graphical user interfaces, network protocols, device drivers, and embedded software.

Testing for functional correctness of EDS such as stand-alone GUI and Web-based applications is critical to many organizations. These applications share several important characteristics. Both are particularly challenging to test because users can invoke many different sequences of events that affect application behavior. Earlier research has shown that existing conventional testing techniques do not apply to either GUIs or Web applications, primarily because the number of permutations of input events leads to a large number of states, and for adequate testing, an event may need to be tested in many of these states, thus requiring a large number of test cases (each represented as an event sequence). Researchers have developed several models for automated GUI testing [1] and Web application testing [2], [3], [4].

Despite the above similarities of GUI and Web applications, all of the efforts to address their common testing problems have been made separately due to two reasons. First is the challenge of coming up with a single model of these applications that adequately captures their event-driven nature, yet abstracts away elements that are not important for functional testing. The absence of such a model has prevented the development of shared testing techniques and algorithms that may be used to test both classes of applications. It has also prevented the development of a shared set of metrics that may be used to evaluate the test results of these types of applications. Second is the unavailability of subject applications and tools for researchers.

In this paper, we focus on the first challenge; i.e., we develop a single abstract model for GUI and Web application testing. To provide focus, we restrict the model to extend our previous work on test prioritization techniques for GUI [5] and Web testing [6]. This allows us to tailor our model to prioritization-specific issues as well as to recast our previous prioritization criteria in a form that is general enough to leverage the single model. In the future, we will extend our model to other testing problems that are shared by GUI and Web applications. Our ultimate goal is to generalize the model and to develop a theory of how EDS should be tested.

The specific contributions of this work include: the first single model for testing stand-alone GUI and Web-based applications, a shared prioritization function based on the abstract model, and shared prioritization criteria. We validate the usefulness of these artifacts through an

- *R.C. Bryce is with the Department of Computer Science, Utah State University, Old Main 414, 4205 Old Main Hill, Logan, UT 84322-4205. E-mail: Renee.Bryce@usu.edu.*
- *S. Sampath is with the Department of Information Systems, University of Maryland, Baltimore County, Baltimore, MD 21250. E-mail: sampath@umbc.edu.*
- *A.M. Memon is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: atif@cs.umd.edu.*

Manuscript received 10 Apr. 2009; revised 21 Oct. 2009; accepted 21 Dec. 2009; published online 19 Jan. 2010.
Recommended for acceptance by A. Bertolino.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2009-04-0079.
Digital Object Identifier no. 10.1109/TSE.2010.12.

empirical study. The results show that GUI and Web-based applications, when recast using the model, showed similar behavior, reinforcing our belief that these classes of applications should be modeled and studied together. Other results show that GUI and Web applications behave differently, which has created opportunities for evolving the model and further experimentation. In future work, we will further generalize the model by evaluating its applicability and usefulness for other software testing activities, such as test generation. Our study also makes contributions toward test prioritization research. Many of our prioritization criteria improve the rate of fault detection of the test cases over random orderings of tests. We also develop hybrid prioritization criteria that combine several criteria that work well individually and evaluate whether the hybrid criteria result in more effective test orders.

Structure of the paper. Section 2 provides background of GUI testing, Web testing, and test prioritization. Section 3 presents our model; Section 4 presents the new prioritization criteria. Section 5 applies our prioritization techniques to seven applications and their existing test suites. Section 6 concludes with a discussion of our current and future work.

2 Background and Related Work

This section provides background on GUI-based applications and Web applications. We summarize the commonalities of these subclasses of EDS and how to combine them into our test suite prioritization model.

2.1 GUI-Based Applications

A GUI is the front-end to a software's underlying back-end code. An end user interacts with the software via *events*; the software responds by changing its state, which is usually reflected by changes to the GUI's widgets. The complexity of back-end code dictates the complexity of the front-end. For example, a single-user application such as Microsoft Paint employs a simple single-user GUI, with discrete events, each completely predictable in its context of use, used to manipulate simple widgets that change their state only in response to user-generated events. More complex applications require synchronization/timing constraints among complex widgets, e.g., movie players that show a continuous stream of video rather than a sequence of discrete frames, and nondeterministic GUIs in which it is not possible to model the state of the software in its entirety (e.g., due to possible interactions with system memory or other system elements) and hence the effect of an event cannot be predicted.

To provide focus, this paper will deal with an important class of GUIs. The important characteristics of GUIs in this class include their graphical orientation, event-driven input, hierarchical structure of menus and windows, the objects (widgets, windows, and frames) they contain, and the properties (attributes) of those objects. Formally, the class of GUIs of interest maybe defined as follows: A **Graphical User Interface (GUI)** is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI.

The above definition specifies a class of GUIs that have a fixed set of events with a deterministic outcome that can be performed on objects with discrete valued properties. **GUI testing**, in this paper, is defined as exercising the entire application by generating only GUI inputs with the intent of finding failures that manifest themselves through GUI widgets. Research has shown that this type of GUI testing finds faults related not only to the GUI and its glue code, but also in the underlying business logic of the application [7].

Current techniques *used in practice* to test such GUIs are largely manual. The most popular tools used to test GUIs are capture/replay tools such as WinRunner[1] that provide very little automation [1], especially for *creating* test cases. There have been attempts to develop state-machine models to automate some aspects of GUI testing, e.g., test case generation and regression testing [8]. In our past work, we have developed an *event-flow model* that represents events and interactions [1]. The event-flow model was designed to capture GUI events and event interactions, but it does not model some of the Web application characteristics, as we describe in Section 3. In this paper, we use the event-flow model to obtain test cases for the GUI applications.

2.2 Web Applications

A Web application consists of a set of pages that are accessible by users through a browser and are transmitted to the end user over a network. A Web page can be static—where content is constant for all users—or dynamic—where content changes with user input. Web applications exhibit characteristics of distributed, GUI, and traditional applications. They can be large with millions of lines of code and may involve significant interaction with users. Also, Web applications are written using many programming languages, such as Javascript, Ajax, PHP, ASP, JSP, Java servlets, and HTML. Languages such as Javascript are referred to as client-side languages, whereas languages such as PHP, ASP, Java servlets, and JSP are referred to as server-side languages. Even a simple Web application can be written in multiple programming languages, e.g., HTML for the front end, Java or JSP for the middle tier, and SQL as the back-end language—which makes testing difficult.

In Web applications, an event can manifest itself in two ways: 1) An event triggered in the client-side code by a user results in a change to the page displayed to the user, without any server-side code execution, e.g., when a user moves the mouse over an HTML link, an event maybe triggered that causes the execution of a Javascript event handler, which in turn results in the link changing color; 2) an event is triggered in the client-side code by a user that results in server-side code being executed, e.g., when the user fills in a form and clicks on the submit button, the data are sent to a server-side program. The server-side program executes and returns the outcome of the execution to the user. In our work, we focus on the latter types of events, i.e., events triggered by a user that result in server-side code execution, as they are readily available in the form of POST or GET requests in server Web

1. http://mercuryinteractive.com.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

logs; we use the logs as the source for our Web application test cases.

Web application testing, in this paper, is defined as exercising the entire application code by generating URL-based inputs with the intent of finding failures that manifest themselves in output response HTML pages. Testing of Web program code to identify faults in the program is largely a manual task. Capture-replay tools capture tester interactions with the application and are then replayed on the Web application [9].

Web application testing research has explored techniques to enable automatic test case generation. Several approaches exist for model-based Web application test case generation [2], [3], [4], [10], [11], [12], [13]. These approaches investigate the problem of test case generation during the development phase of an application. Another approach to generating test cases, and the one used in this paper, is called user session-based testing; it advocates the use of Web application usage data as test cases [14], [15], [16].

2.3 Test Prioritization of GUI and Web Applications

Due to their user-centric nature, GUI and Web systems routinely undergo changes as part of their maintenance process. New versions of the applications are often created as a result of bug fixes or requirements modification [17]. In such situations, a large number of test cases may be available from testing previous versions of the application which are often reused to test the new version of the application. However, running such tests may take a significant amount of time. Rothermel et al. report an example for which it takes weeks to execute all of the test cases from a previous version [18]. Due to time constraints, a tester must often select and execute a subset of these test cases. Test case prioritization is the process of scheduling the execution of test cases according to some *criterion* to satisfy a *performance goal*.

Consider the function for test prioritization as formally defined in [18]. Given T, a test suite, Π, the set of all test suites obtained by permuting the tests of T, and f, a function from Π to the set of real numbers, the problem is to find $\pi \in \Pi$ such that $\forall \pi' \in \Pi$, $f(\pi) \geq f(\pi')$. In this definition, Π refers to the possible prioritizations of T and f is a function that is applied to evaluate the orderings. The selection of the function f leads to many criteria to prioritize software tests. For instance, prioritization criteria may consider code coverage, fault likelihood, and fault exposure potential [18], [19]. Binkley uses the semantic differences between two programs to reduce the number of tests that must be run during regression testing [20]. Jones and Harrold reduce and prioritize test suites that are MC/DC adequate [21]. Jeffrey and Gupta consider the number of statements executed and their potential to influence the output produced by the test cases [22]. Lee and He reduce test suites by using tests that provide coverage of the requirements [23]. Offutt et al. use coverage criteria to reduce test cases [24]. None of these prioritization criteria have been applied to event-driven systems.

In our past work, we have developed additional criteria to prioritize GUI and Web-based programs. Bryce and Memon prioritize preexisting test suites for GUI-based programs by the lengths of tests (i.e., the number of steps in a test case, where a test case is a sequence of events that a user invokes through the GUI), early coverage of all unique events in a test suite, and early event interaction coverage between windows (i.e., select tests that contain combinations of events invoked from different windows which have not been covered in previously selected tests) [5]. In half of these experiments, event interaction-based prioritization results in the fastest fault detection rate. The two applications that cover a larger percentage of interactions in their test suites (64.58 and 99.34 percent, respectively) benefit from prioritization by interaction coverage. The applications that cover a smaller percentage of interactions in their test suites (46.34 and 50.75 percent, respectively) do not benefit from prioritization by interaction coverage. We concluded that the interaction coverage of the test suite is an important characteristic to consider when choosing this prioritization technique.

Similarly, in the Web testing domain, Sampath et al. prioritize user session-based test suites for Web applications [6]. These experiments showed that systematic coverage of event interactions and frequently accessed sequences improve the rate of fault detection when tests do not have a high Fault Detection Density (FDD), where FDD is a measure of the number of faults that each test identifies on average.

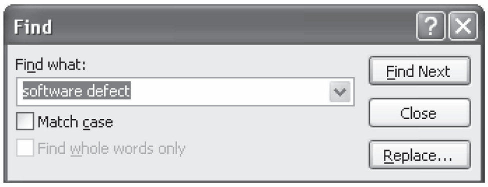
3 Combined Model

To develop the unified model, we first review how GUI and Web applications operate. For GUI applications, action listeners are probably the easiest—and most common—event handlers to implement. The programmer implements an action listener to respond to the user's indication that some implementation-dependent action should occur. When the user performs an event, e.g., clicks a button, chooses a menu item, an action event occurs. The result is that (using the Java convention) an `actionPerformed` message is sent to all action listeners that are registered on the relevant component. For example, the following is an action event using Java code:

```
public class myActionListener ... implements
    ActionListener {
    ...
    //initialization code:
    button.addActionListener(this);
    ...
    public void actionPerformed(ActionEvent e)
          {doSomething();}}
```

The `doSomething()` method is invoked each time the event is executed. Such action listeners are typically implemented for all widgets in the GUI. Due to this reason, in our previous work on GUI testing, we modeled each *event* as an action on a GUI widget. Examples of some events included opening menus, checking check-boxes, selecting radio-buttons, and clicking on the `Ok` button. Each event was modeled in exactly the same way. For example, consider a "preferences setting" dialog in which a user employs a variety of radio-button widgets, check-boxes, and tabs to set an application's preferences. The user terminates the dialog explicitly by clicking the `Ok` button. Our earlier GUI model would model each invocation of each widget as an event, including the final `Ok` action. We

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.



(a)

Parameter, Value
1. <"Find what" drop-box, settext>
2. <"Find what" drop-box, leftclick dropdown>
3. <"Match case" checkbox, leftclick select>
4. <"Match case" checkbox, leftclick unselect>
5. <"Find whole words only" checkbox, leftclick select>
6. <"Find whole words only" checkbox, leftclick unselect>
7. <"Find Next" button, leftclick>
8. <"Close" button, leftclick>
9. <"Replace" button, leftclick>

(b)

(c)

Parameter and value descriptions for Login.jsp
1. < Login text field, guest >
2. < Password text field, guest >
3. < FormAction, Login >
4. < FormName, Login >

(d)

Fig. 1. Examples of a GUI and Web Application. (a) Example GUI application window. (b) Nine parameter values on the GUI window. (c) Example Web application window. (d) Four parameter values in the Web Application window.

did not model the fact that the `Ok` button is the only event that actually causes changes to the application's settings.

On the other hand, Web application GUI widgets behave differently. That is, some widget actions are handled at the client (e.g., in the form of Javascript code in the browser), whereas others, such as the `Submit` button trigger a GET or POST request from the client to the server. In our earlier work, we modeled only the GET/POST actions, i.e., those actions that cause a client to send and receive data from the server. Client-side events were used to set variables that were used as parameters to the actual GET/POST event. Consider the "preferences setting" dialog discussed earlier, except that it is now in a Web page. Our earlier model of a Web event would not treat all the individual radio-button and check-box settings as individual events; instead it would use the widget settings as parameters to the `Ok` button's POST request.

These two earlier models of GUI (each action as an event) and Web (only GET/POST actions as events) were incompatible. If we use these two models to study the characteristics of GUI and Web applications, we would expect to get incorrect and incoherent results. We thus need a new unified model that can tie these application classes together.

3.1 Modeling Windows, Widgets, and Actions

Despite the differences in how GUI and Web applications were modeled in prior research, these two classes of applications have many similarities. This paper draws upon these similarities to create the single model for test suite prioritization of both GUI and Web applications. We now identify similarities in these applications and develop a unified set of terms via examples.

Fig. 1a shows an example *window* from a GUI application entitled "Find." We use the term **window** to refer to GUI windows such as this Find window. The window has several *widgets*. A user typically sets some properties of these widgets (e.g., checking a check-box, adding text to a text field) and "submits" this information. Underlying code then uses these settings to make changes to the software state. Because of how widgets are used in the GUI, we refer to them as **parameters** in this paper. We refer to the *settings for the widgets* as **values**. We refer to the pair <parameter_name, *value*> as **parameter-values**. For instance, in Fig. 1a, the "Find what" drop-down box is a parameter with the value "software defect"; the "Match case" check-box is a parameter with the value "false"; these parameters are used by actions. Fig. 1b shows all possible parameter-values for the window shown in Fig. 1a. In this paper, we refer to a consecutive sequence of user interactions on a single window as an **action**. An example of an action for the Find window is the sequence "enter 'software defect' in text-box," "check 'Match case' check-box," and "click-on 'Find Next' button."

Similarly, for Web applications, we refer to a Web application *page* as a **window**. As with GUIs, widgets in a window are referred to as **parameters**, and their settings as **values**. Fig. 1c shows a sample Web page (one window). Fig. 1d lists the four **parameter-values** on the window. For instance, the "Login" text field is a **parameter** that is set to the **value** "guest."

In addition to parameters receiving values from user interactions, an application may assign values to parameters on the page, e.g., hidden form fields and their values. In Fig. 1d, an example of such a parameter value is the "FormName" parameter that gets the value "Login," which is set by the application. In this paper, we consider both types of parameter values. When a user clicks on the "Login" button on the Web page, an **action** is invoked, that is, an HTTP POST or GET request is sent to the Web server. The parameter value settings in the window are transmitted to the Web server.

Note that we defined a GUI **action** very carefully so that we have a unified terminology between GUI and Web applications for this paper. For instance, in Web applications, there maybe multiple user interactions on a single window in which users set values for parameters before any information is actually sent to the Web server (e.g., a POST or GET request). To maintain consistency in our terminology

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 1
Unified Terminology for GUI and Web-Based Applications

GUI	Web application	Term in paper
window	page	window
widget (e.g., text field)	parameter (e.g., form field)	parameter
a setting for a widget (e.g., a box is checked)	a setting for a parameter (e.g., a name in a text field or a param 'userid')	value
the pair of widget name, and its setting (e.g., <'Find' box, checked>)	the pair of parameter-name and its setting (e.g., <userid, guest>)	parameter-value (PV)
a sequence of user-interactions that set one or more parameter-values on a single window	a sequence of user-interactions that set one or more parameter-values on a single page	action

for both GUI and Web applications, we unify the term **action** to be the consecutive set of all user interactions on a single window before moving to a new window. Table 1 summarizes all our terms.

3.2 Modeling Test Cases

A test case is modeled as a sequence of actions. For each action, a user sets a value for one or more parameters. We provide examples of test cases for both GUI and Web applications next.

Table 2a provides a sample test case for a GUI application called *TerpWord* (this application is described later in Section 5). The test case sets nine parameters to values and visits three unique windows. The test includes visits to the TerpWord main window, Save, and Find windows. An **action** occurs when a user sets values to one or more parameters on a window before visiting a different window. From Table 2a, we see that in *Action 1*, the user selects File->Save from the *TerpWord main menu*. The parameter values associated with this action are shown in first two rows of Table 2b. The parameter-values set in *Action 2* occur on the *Save Window* to set the file name to "exampleFile," select the file type as plain text, and click the OK button. The user sets parameter-values in *Action 3* on the *TerpWord main window* by selecting Edit->Find. *Action 4* involves parameter-values on the "Find" window. The user sets the text of the "*Find what* drop-box" to "software defect" and then executes a "left-click" on the *Find Next* button. Table 2a summarizes the windows, parameters, and values in this test case and assigns unique numbers to each window and action.

Table 3a shows a sample user-session (test case) from the Book application (described later in Section 5) that contains four actions. Table 3b shows the important data that we parse from the test case. From the test case in Table 3a, we see that the *Login* page is accessed with the parameter-values <Password, guest>, <FormName, Login>, <FormAction, login>, and <Login, guest>. For the example test case in Table 3a, all of the parameter values are shown in Table 3b.

4 Prioritization Criteria

We now use the combined model to develop generalized prioritization criteria. But before we present the criteria, we provide a generic function that we use to formally define the criteria and we introduce a running example to help understand the criteria.

4.1 Prioritization Function

The function takes as input a set of test cases to be ordered and returns a sequence that is ordered by the prioritization criterion. Because we have developed a unified model of GUI and Web applications, we need the function to be extremely general so that it may be instantiated for either application class, and is able to use any of our criteria as a parameter. Moreover, our function is not necessarily optimized for each individual prioritization criterion, but rather is intentionally left general to make it easy for readers to quickly implement our criteria. The function (called $OrderSuite$) selects a test case that covers the maximum number of criteria elements (e.g., windows and parameters) not yet covered by already-selected test cases. The function iterates until all test cases have been ordered.

The function for the test selection process is presented in Fig. 2. $OrderSuite$ takes four parameters:

1. The suite to be ordered—note that this is a set.
2. A function f that takes a single test case as input and returns a set of elements that are of interest to the criterion being used as the basis for prioritization. For example, if we prioritize tests by the number of new unique windows that they cover, then $f(x)$ returns the set of windows covered by the test case x.
3. Another function F (related to f above) operates on the sequence of test cases, S, selected thus far. For the example discussed in the above paragraph, $F(S)$ returns the set of all windows covered by the test cases in sequence S. In this example, $F(S)$ essentially applies the above f to each element in S and takes a set-union of the results.
4. An operation $\oplus$ assigns a "fitness" value to the current test case. For the above example, $\oplus$ is the composed function (`SetCardinality` $\circ$ `SetDifference`), i.e., "cardinality of the set difference." Hence, a test case that covers the maximum number of unique windows not yet covered by the test cases selected thus far will have the largest value for this function's output and hence, "most fit"; it will be selected next to be inserted in the ordered sequence. If two or more test cases share the top position for selection, then a random choice is made using the RANDOM() (returns a random real number between 0 and 1) function in $BestNextTestCase$.

Function $OrderSuite$ starts with an unordered sequence and invokes $BestNextTestCase$ until all of the test cases

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 2
Example GUI Test Case

Start of TC	<Testcase>
No. of Actions	<Length>4</Length>
Action 1	<Menu> <Window>TerpWord</Window> <Nonterminal>File</Nonterminal> </Menu> <Menu> <Window>TerpWord</Window> <Nonterminal>Save</Nonterminal> </Menu>
Action 2	<Component> <Window>Save</Window> <Nonterminal>File Name text field</Nonterminal> <Eventtype>SETTEXT</Eventtype> <Eventvalue>exampleFile</Eventvalue> </Component> <Component> <Window>Save</Window> <Nonterminal>Files of Type drop-down box</Nonterminal> <Eventtype>LEFTCLICK SELECT</Eventtype> <Eventvalue>Plain Text File (*.txt)</Eventvalue> </Component> <Component> <Window>SAVE</Window> <Nonterminal>OK button</Nonterminal> <Eventtype>LEFTCLICK</Eventtype> </Component>
Action 3	<Menu> <Window>TerpWord</Window> <Nonterminal>Edit</Nonterminal> </Menu> <Menu> <Window>TerpWord</Window> <Nonterminal>Find...</Nonterminal> </Menu>
Action 4	<Component> <Window>Find</Window> <Nonterminal>Find what drop-box</Nonterminal> <Eventtype>SETTEXT</Eventtype> <Eventvalue>software defect</Eventvalue> </Component> <Component> <Window>Find</Window> <Nonterminal>FindNext button</Nonterminal> <Eventtype>LEFTCLICK</Eventtype> </Component>
End of TC	</Testcase>

(a)

Window name	P-V No.	P-V description (<parameter,value>)
TerpWord	PV.1 PV.2	<File,null> <Save,null>
Save	PV.3 PV.4 PV.5	<File name text field, SETTEXT="exampleFile"> <Files of Type drop-down box, LEFTCLICK SELECT="Plain Text File (*.txt)"> <OK button, LEFTCLICK>
TerpWord	PV.6 PV.7	<Edit, null> <Find..., null>
Find	PV.8 PV.9	<Find what drop-box, SETTEXT="software defect"> <FindNext button, LEFTCLICK>

(b)

(a) Sample GUI test case. (b) Windows and user interactions in test case.

TABLE 3
Example Web Test Case

No. of actions in Test	Action
Action 1	GET Default.jsp
Action 2	GET Login.jsp
Action 3	POST Login.jsp?Password=guest &FormName=Login& FormAction=login&Login=guest
Action 4	GET BookDetail.jsp?item_id=22

(a)

Window name	Parameter-value No.	PV Description (<parameter,value>)
GET Default.jsp	PV.1	<null,null>
GET Login.jsp	PV.2	<null, null>
POST Login.jsp	PV.3 PV.4 PV.5 PV.6	<Password,guest> <FormName,Login> <FormAction,login> <Login,guest>
GET BookDetail.jsp	PV.7	<item_id,22>

(b)

(a) Sample user session-based Web test case. (b) Windows and parameter values in test case.

Input Parameters:
$Suite$: Test suite to be prioritized (represented as a set);
f: Function returns criteria elements of a single test case;
F: Function returns criteria elements in sequence of test cases;
$\oplus$: Operation combines results of f and F; returns number;
Output:
$OrderedSequence$: Priority ordered sequence containing all tests;
Computation:
$S \leftarrow EMPTY$;
$T \leftarrow Suite$;
REPEAT
$t \leftarrow BestNextTestCase(S, T, f, F, \oplus)$;
$S \leftarrow InsertAtEnd(S, t)$;
$T \leftarrow T - t$;
UNTIL ($T == \phi$);
$OrderedSequence \leftarrow S$;

(a)

Input Parameters:
S: Priority ordered sequence of test cases selected so far;
T: Set of remaining test cases;
f: Function returns criteria elements in single test;
F: Function returns criteria elements in sequence of tests;
$\oplus$: Operation combines results of f and F; returns number;
Output:
t: a test case from T;
Computation:
$Max \leftarrow MININT$;
$fs \leftarrow F(S)$;
FORALL $x \in T$ {
$y \leftarrow f(x) \oplus fs$;
IF ($(Max < y)$ || ($(Max == y)$ && (RANDOM() ≤ 0.5))) {
$Max \leftarrow y$;
$t \leftarrow x$;
}
}
RETURN(t);

(b)

Fig. 2. The $OrderSuite$ function used in our work. (a) **Function:** $OrderSuite$. (b) **Function:** $BestNextTestCase$.

have been ordered. We will instantiate f, F, and $\oplus$ for each of our prioritization criteria.

4.2 Running Example

Table 4 shows our running example; it contains five windows and 18 values for the parameters on the windows. We label the windows with numeric values, W1 through W5, and label the values for the parameters on the windows as 1 through 18. For instance, the first window (W1) includes five possible values for parameters (labeled as 1-5). In practice, these numeric IDs for the windows and values map to the actual window names and actual values for the parameters on those windows. (Refer to Tables 2b and 3b in the previous section for examples of windows and values.) We also show six tests that allow us to provide hand traces of the criteria.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 4
Example Application and Example Test Suite

Windows				
W1	W2	W3	W4	W5
1	6	12	15	17
2	7	13	16	18
3	8	14		
4	9			
5	10			
	11			

Test	Parameter-values	Windows visited
t_1	1→2→5→6→15→8→4→8	W1→W2→W4→W2→W1→W2
t_2	1→3→6→17	W1→W2→W5
t_3	1→4→5→6→1→9→14	W1→W2→W1→W2→W3
t_4	2→3→6→8→10→11→ 12→9→13→16	W1→W2→W3→W2→W3→W4
t_5	1→4→5→6→14→15→18	W1→W2→W3→W4→W5
t_6	3	W1

4.3 Parameter-Value Interaction Coverage-Based Criteria

One aspect of event-driven software and test cases which also holds for other types of systems is their dependency on parameters and values for execution. Interactions between multiple parameter values make the program follow a distinct execution path and are likely to expose faults in the system. This basic premise led to the development of our first set of prioritization criteria, which are based on giving higher priority to test cases with a large number of parameter values interactions. The *1-way* and *2-way* parameter value interaction coverage techniques select tests to systematically cover parameter value interactions between windows.

4.3.1 1-way

The *1-way* criterion selects a next test to maximize the number of parameter values that do not appear in previously selected tests. We hypothesize that the faster systematic coverage of settings for parameters may expose faults earlier. For $OrderSuite$, we instantiate $f(x)$ to return the set of parameter values in test case x, $F(S)$ to return the set of parameter values accessed by all test cases in sequence S; $\oplus$ is the function (`SetCardinality` $\circ$ `SetDifference`) discussed earlier.

For our running example, the first selected test is t_4 because it covers 10 parameter values, i.e., (2,3,6,8,9,10,11,12, 13,16). The next test selected is t_5 because it covers six parameter values that were not covered in the first selected test (t_4), including parameter values (1,4,5,14,15,18), whereas tests t_1 and t_3 only cover four new parameter values, t_2 covers two new parameter values, and t_6 does not cover any new parameter values that were not covered in a previous test. The final prioritized sequence is t_4, t_5, t_2, t_1, t_3, and t_6, where the last three test cases are ordered at random since t_4, t_5, and t_2 have already covered all *1-way* parameter values.

4.3.2 2-way

The *2-way* criterion selects a next test to maximize the number of 2-way parameter value interactions between windows. We hypothesize that interactions of parameters set to values on different windows may expose faults. For $OrderSuite$, we instantiate $f(x)$ to return the set of 2-way parameter value interactions between windows accessed by test case x; $F(S)$ is similar, except that it operates on the sequence S; $\oplus$ is the function used earlier.

Table 5a shows the 2-way interactions for our running example. Test case t_4 would be chosen first since it covers the most 2-way interactions. Table 5b shows a summary of the 2-way interactions left to cover after t_4 is chosen as the first test case. Test case t_5 is chosen after t_4 because it covers the most previously untested 2-way interactions. Test t_6 is a special case in this example since it does not cover any 2-way interactions since only one parameter is set to a value on a single window (W1). The final prioritized sequence is t_4, t_5, t_3, t_1, t_2, and t_6, where there was a tie between t_1 and t_2 for the selection of the fourth test case.

4.4 Count-Based Criteria

Another factor important to test cases for event-driven systems is the implicit dependency between the variety and number of window artifacts it accesses and the amount of code covered (and possibly faults exposed) on executing these test cases. Our next set of criteria prioritize test cases based on counts of the number of windows, actions, or parameter values that they cover.

TABLE 5
2-Way Interactions in the Tests of Table 4

Test No.	No. 2-way interactions	List of 2-way interactions
t_1	13	(1,6),(1,15),(1,8),(2,6),(2,15),(2,8),(5,6),(5,15), (5,8),(6,15),(4,6),(4,15),(4,8)
t_2	4	(1,6),(1,17),(3,6),(3,17)
t_3	11	(1,6),(1,9),(1,14),(4,6),(4,9),(4,14),(5,6),(5,9), (5,14),(6,14),(9,14)
t_4	30	(2,6),(2,8),(2,9),(2,10),(2,11),(2,12),(2,13),(2,16), (3,6),(3,8),(3,9),(3,10),(3,11),(3,12),(3,13),(3,16), (6,12),(6,13),(6,16),(8,12),(8,13),(8,16),(9,12), (9,13),(9,16),(10,12),(10,13),(10,16),(12,16),(13,16)
t_5	18	(1,6),(1,14),(1,15),(1,18),(4,6),(4,14),(4,15),(4,18), (5,6),(5,14),(5,15),(5,18),(6,14),(6,15),(6,18), (14,15),(14,18),(15,18)
t_6	0	none

(a)

Test No.	No. 2-way interactions	List of 2-way interactions
t_1	11	(1,6),(1,15),(1,8),(2,15),(5,6),(5,15), (5,8),(6,15),(4,6),(4,15),(4,8)
t_2	3	(1,6),(1,17),(3,17)
t_3	11	(1,6),(1,9),(1,14),(4,6),(4,9),(4,14),(5,6),(5,9), (5,14),(6,14),(9,14)
t_5	18	(1,6),(1,14),(1,15),(1,18),(4,6),(4,14),(4,15),(4,18), (5,6),(5,14),(5,15),(5,18),(6,14),(6,15),(6,18), (14,15),(14,18),(15,18)
t_6	0	

(b)

(a) All 2-way interactions. (b) Untested interactions after t_4 selected as first test.

4.4.1 Unique Window Coverage

In this criterion, we prioritize tests by giving preference to test cases that cover the most *unique windows* that previous tests have not covered. We hypothesize that faults will be exposed when we visit windows and that we should visit all windows as soon as possible. For $OrderSuite$, we instantiated $f(x)$ to return the set of windows accessed by test case x; $F(S)$ is similar, except that it operates on the sequence S; $\oplus$ is the function used earlier. For our running example, we select t_5 first because it covers all five windows of the application, and then randomly select the remaining test cases, yielding, e.g., t_5, t_3, t_1, t_4, t_2, t_6.

4.4.2 Action Count-Based

In this criterion, we prioritize tests by the number of actions in each test (duplicates included). Recall, from Table 1, an action is a sequence that sets one or more parameter values in a single window. The prioritization includes selecting the test cases, with preference given to those that include the most number of actions, *Action-LtoS*. For $OrderSuite$, we instantiated $f(x)$ to return the number of actions (also counting duplicates) in test case x; because this criterion does not care about test cases that have already been selected, $F(S) = 0$; $\oplus$ returns its first parameter, i.e., the value of $f(x)$. *Action-StoL* gives priority to test cases with the smallest number of actions. For $OrderSuite$, $f(x) =$ `Negative` of the f function used in *Action-LtoS*. For our running example, when using *Action-LtoS*, there is a tie between tests t_1 and t_4 as they each include six actions. When we apply *Action-StoL*, test t_6 is selected first because it contains the shortest sequence of actions by covering only one action. The final prioritized sequence for *Action-LtoS* is t_4, t_1, t_3, t_5, t_2, t_6, where there is a tie between t_4 and t_1 for the selection of the first test case and another tie between t_3 and t_5 for the third test case. The final prioritized sequence for *Action-StoL* is t_6, t_2, t_5, t_3, t_1, t_4, where there is a tie between t_5 and t_3 for the selection of the third test case and between t_1 and t_4 for the fifth test case.

4.4.3 Parameter-Value Count-Based

Test cases contain settings for parameters that users set to specific values. We prioritize tests by the number of parameters that are set to values in a test case (duplicates included). We hypothesize that test cases that set more parameters to values are more likely to reveal faults. This includes selecting those tests with the largest number of parameter value settings in a test first, called *PV-LtoS*. For $OrderSuite$, we instantiated $f(x)$ to return the number of parameters that are set to values (also counting duplicates) in test case x; again, $F(S) = 0$ and $\oplus$ returns its first parameter, i.e., the value of $f(x)$. We also prioritize in the reverse manner by selecting those tests with the smallest number of parameter value settings first, called *PV-StoL*. Here too, $f(x) =$ `Negative` of the f function used in *PV-LtoS*.

For our running example, the first selected test is t_4 because 10 parameters are set to values (2,3,6,8,9,10,11,12, 13,16). The next test that would be selected is t_1 because it covers eight parameter values, whereas tests t_3 and t_5 only cover seven parameter values, t_2 covers four parameter values, and t_6 covers one parameter value. The final prioritized sequence for *PV-LtoS* is t_4, t_1, t_5, t_3, t_2, and t_6, where a tie occurs between t_5 and t_3 for the selection of the third test case. The final prioritized sequence for *PV-StoL* is t_6, t_2, t_3, t_5, t_1, and t_4, where a tie occurs for the selection of the third test case.

4.5 Frequency-Based Criteria

Our final set of criteria give higher priority to test cases that cover windows that are perceived to be important to the EDS from a testing perspective. Since our Web test cases are based on usage profiles, in this paper, we define importance of a window as the number of times the window is accessed in the test cases. Because of the user-centric design of event-driven software, these windows are likely to contain more code functionality (and likely to contain more faults), and thus test cases that cover such windows are given higher priority.

The following three criteria differ in how they view the frequency of presence of a window sequence in a test case, and thus produce different prioritized orders. We consider window sequences of size 2 in this paper.

4.5.1 Most Frequently Present Sequence (MFPS) of Windows

In this criterion, *MFPS*, we first identify the most frequently present sequence of windows, s_i, in the test suite and order test cases in decreasing order of the number of times that s_i appears in the test case. Then, from among the test cases that do not use s_i even once, the most frequently present sequence, s_j, is identified, and the test cases are ordered in decreasing order of the number of times s_j appears in the test case. This process continues until there are no more remaining test cases.

For $OrderSuite$, our function f uses a helper function g in its computation. Function g takes the original test suite ($Suite$) as input, extracts all pairs of windows accessed by its constituent test cases, and computes the frequency of access of each pair. The pair p with the largest frequency is of interest to our criterion *MFPS*. Function $f(x)$ simply returns the number of times p appears in test case x. Function F trivially returns zero and operation $\oplus$ returns the value of its first argument.

For the example test suite of Table 4, we determine the number of times each sequence appears in the test suite. The second column in Table 6 shows the frequency of presence for each sequence ordered in decreasing order. Since $\mathrm{W1} \rightarrow \mathrm{W2}$, is the most frequently present sequence, *MFPS* selects test cases in decreasing order of the number of times $\mathrm{W1} \rightarrow \mathrm{W2}$ appears in the test case. We first select t_1 and t_3 because they have the sequence $\mathrm{W1} \rightarrow \mathrm{W2}$ the greatest number of times. We continue selecting test cases that have the sequence $\mathrm{W1} \rightarrow \mathrm{W2}$ in them. From Table 4, we see that test cases t_2, t_4, and t_5 have the sequence $\mathrm{W1} \rightarrow \mathrm{W2}$ the same number of times, i.e., once. Therefore, these three test cases are randomly ordered and appended to the test suite. The prioritized order for the test suite in Table 4 is now t_1, t_3, t_2, t_5, t_4. Since t_6 is the only test case that does not have the sequence $\mathrm{W1} \rightarrow \mathrm{W2}$, it is appended to the end of the test order, creating the final prioritized test order t_1, t_3, t_2, t_5, t_4, t_6.

4.5.2 All Present Sequence (APS) of Windows

Since *MFPS* gives importance to only the frequency of occurrence of a single most frequently present sequence, the

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 6
Frequency of Presence Table

Sequence Name	Total No. of Occurrences	Test cases with max. occurrences of sequence
W1→W2	7	t_1, t_3
W2→W3	4	t_4
W3→W4	2	t_4, t_5
W2→W1	2	t_1, t_3
W2→W4	1	t_1
W4→W2	1	t_1
W2→W5	1	t_2
W4→W5	1	t_5
W3→W2	1	t_4

criterion is likely to lose important information about other frequently present sequences which may not be accessed as many times as the most frequently present sequence but are still accessed frequently in the test suite. To accommodate all present sequences during prioritization, the *APS* criterion is proposed. In *APS*, the frequency of occurrence of all sequences is used to order the test suite. For each sequence, s_i, in the application, beginning with the most frequently present sequence, test cases that have maximum occurrences of these sequences are selected for execution before other test cases in the test suite.

Consider the example test suite from Table 4 and the corresponding frequency table shown in Table 6. The third column in Table 6 shows the test cases that have the maximum occurrences of the corresponding sequence. Starting with sequence W1 → W2, we see that test cases t_1 and t_3 have the maximum occurrence of this sequence (i.e., twice). Thus, they are selected first for the prioritized order. The tie between them is broken at random. The prioritized order is now t_1, t_3. Then, the *APS* algorithm proceeds to the next most frequently present sequence, W2 → W3. From Table 6, since t_4 covers W2 → W3 the maximum number of times, and t_4 is not already selected in the prioritized order, test case t_4 is selected next for the prioritized order. The next sequence from Table 6 is W3 → W4. From the two test cases in W3 → W4's test cases list, test case t_4 is already in the prioritized order. The only remaining test case corresponding to this sequence, t_5, is added to the prioritized order. The prioritized order is now t_1, t_3, t_4, t_5. Since all of the test cases corresponding to the next three sequences in Table 6 are already in the prioritized order, the algorithm then adds t_2 for sequence W2 → W5 to the prioritized order. All of the test cases corresponding to the remaining sequences in Table 6 are already in the prioritized order. When all rows in the table are covered, any remaining test cases in the test suite are randomly ordered and appended to the prioritized order of test cases. The final prioritized order is t_1, t_3, t_4, t_5, t_2, t_6.

For $OrderSuite$, our function f again uses a new helper function g in its computation. Function g initializes itself by taking the original test suite ($Suite$) as input, extracting all pairs of windows accessed by its constituent test cases, and computing the frequency of access of each pair. It also maintains a list of test cases with each frequency entry. Actually, the implementation of g maintains an object similar to Table 6. It also has access to the object S of $OrderSuite$, which contains all tests selected thus far. Function $f(x)$ invokes g with parameter x. If x is in S, then g returns zero. Otherwise, it returns the largest frequency value (column 2 of Table 6) associated with x (from the lists in Column 3 of Table 6). Function F trivially returns zero and operation $\oplus$ returns the value of its first argument.

As seen from this example, instead of focusing on only a single *most* frequently present sequence, *APS* also gives importance to other frequently present sequences. Thus, W2 → W3 is given consideration and the test case that covers this sequence is selected before other test cases in the prioritized order, e.g., test case t_4, which appeared at the end of the prioritized order in *MFPS*, is given priority and ordered third by *APS*.

4.5.3 Weighted Sequence of Windows (Weighted-Freq)

While *MFPS* gives importance to a particular window sequence and *APS* selects test cases based on only one sequence, the weighted technique assigns each test case a weighted value based on all of the window sequences it contains and the importance (the weight of a sequence of windows is measured by the number of times the sequence appears in the suite) of the window sequence.

Initially, we identify the frequency of appearance of each unique sequence of windows in the test suite and build a weighted matrix for each unique window sequence. This frequency of appearance is the weight of the unique sequence of window. The second column in Table 6 represents the weight of each sequence in the example test suite Table 4. In the following example, the weight of a sequence, e.g., W1 → W4, is denoted as $WS_{w1\rightarrow w4}$.

Thereafter, in each test case, we count the number of times each unique sequence of windows appears. The test case has a weighted value based on the summation of the product of the number of times each unique sequence of windows appears in the test case and the corresponding weight of this unique sequence in the weighted matrix table.

Table 7 shows the weighted value for each test case for the example test suite from Table 4. Each test case in the test suite is assigned a weighted value based on the window sequences that the test case contains. Test cases are prioritized by decreasing order of their weighted value. In this example, the final prioritized order is t_3, t_4, t_1, t_5, t_2, t_6. By assigning a weighted value to each test case based on all of the sequences contained in the test case, *Weighted-Freq* identifies test case t_3 as the most important test case,since it covers both the important sequences, W1 → W2 and W2 → W3.

For $OrderSuite$, $f(x)$ simply computes the weighted value of x, F trivially returns zero, and $\oplus$ trivially returns its first argument.

5 Empirical Study

The two main underlying questions this research strives to answer are: *1) Does the model help us to study GUI and Web application testing using a common set of metrics/criteria? and 2) Does the model help to identify commonalities between GUI and Web applications?*

Only because of the development of the model proposed in Section 3 were we able to define prioritization criteria that accurately capture characteristics of both

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 7
Weighted Frequency Table

Test case name	Weighted value computation	Weighted value
t_1	$(2 * WS_{w1\rightarrow w2}) + (1 * WS_{w2\rightarrow w4}) + (1 * WS_{w4\rightarrow w2}) + (1 * WS_{w2\rightarrow w1})$	18
t_2	$(1 * WS_{w1\rightarrow w2}) + (1 * WS_{w2\rightarrow w5})$	8
t_3	$(2 * WS_{w1\rightarrow w2}) + (1 * WS_{w2\rightarrow w1}) + (1 * WS_{w2\rightarrow w3})$	20
t_4	$(1 * WS_{w1\rightarrow w2}) + (2 * WS_{w2\rightarrow w3}) + (1 * WS_{w3\rightarrow w2}) + (1 * WS_{w3\rightarrow w4})$	18
t_5	$(1 * WS_{w1\rightarrow w2}) + (1 * WS_{w2\rightarrow w3}) + (1 * WS_{w3\rightarrow w4}) + (1 * WS_{w4\rightarrow w5})$	14
t_6	N/A	0

TABLE 8
Composition of the Applications and Test Suites in Our Study

	Calc	Paint	SSheet	Word	Book	CPM	Masplas
Windows	2	11	9	12	9	65	18
Parameter-values	85	247	188	156	415	4166	646
LOC	9916	18376	12791	4893	7615	9401	999
Classes	141	219	125	104	11	75	9
Methods	446	644	579	236	319	173	22
Branches	1306	1277	1521	452	1720	1260	108
Total no. of tests	300	300	300	250	125	890	169
Largest count of actions in a test case	47	51	50	50	160	585	69
Average count of actions in a test case	14.5	19.7	19	27.8	29	14	7
2-way parameter-value interactions covered in test suite	99.34%	46.34%	50.75%	64.58%	92.50%	97.80%	96.20%
No. of seeded faults	175	182	79	96	40	135	29
Fault Detection Density (FDD)	.05	.02	.02	.29	.59	.056	.19
Min. no. faults found by a test	0	0	0	0	6	0	1
Avg. no. faults found by a test	9.4	1.6	4	24	21.43	4.67	4.62
Max. no. faults found by a test	48	64	71	87	32	33	15

GUI and Web systems. Thus, the successful development of the prioritization criteria presented in Section 4 implicitly answers question 1. To study question 2, we take a more quantitative approach. We design an empirical study that studies the effectiveness of the prioritization criteria to determine whether the criteria, and therefore the model, help in identifying commonalities between the two classes of applications. The two research questions in this empirical study are:

1. RQ1: **Which prioritization criteria are among the best/worst criteria for both GUI and Web systems?**
2. RQ2: **(Exploratory Question) Is a combination of different prioritization criteria more effective than any single criterion?** Because each criterion targets a unique aspect of the EDS, this question is designed to explore whether a combination of criteria (such a combination would account for multiple aspects of the EDS) is more effective than any single criterion.

5.1 Subject Applications

We use four GUI and three Web-based applications, shown in Table 8.

5.1.1 TerpOffice GUI Application Suite

The GUI applications are part of an open source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course. It is called TerpOffice[2] and includes TerpCalc (a scientific calculator with graphing capability), TerpPaint (an image editing/manipulation program), TerpSpreadSheet (a spreadsheet application), and TerpWord (a small word processor). They have been implemented using Java. We have described these applications in several earlier reported studies [1].

5.1.2 Web Application Suite

The Web applications were partly developed at the University of Delaware and used in earlier reported studies [16]. **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. Since our interest was in testing consumer functionality, we did not include the administration code in our study [16]. Book uses JSP for its front end and a MySQL database back end. **CPM** enables course instructors to login and create *grader* accounts for teaching assistants. Instructors and teaching assistants create *group* accounts for students, assign grades, and create schedules for demonstration time slots. Users interact with an HTML application interface generated by Java servlets and JSPs. **Masplas** enables users to register for a workshop, upload abstracts and papers, and view the schedule, proceedings, and other related information. Masplas is written using Java, JSP, and MySQL.

2. http://www.cs.umd.edu/users/atif/TerpOffice.

Table 8 gives an overview of our subject application's characteristics, such as the lines of code in each application, number of classes, methods, branches, windows, and events. The applications are nontrivial as most contain several thousand lines of code, between 9 to 219 classes, 22 to 644 methods, and between 108 to 1,720 branches. The test cases exercise between 85 to 4,166 parameter-values on the user interfaces of these applications. The longest test cases for each of these applications exercise between 47 to 585 actions.

5.2 Test Suites

Models of the TerpOffice applications, called event-flow graphs [1], were used to generate test cases. The test case generation algorithm has also been described earlier [1]; in summary, the algorithm is based on graph traversal; starting in one of the events (represented by a node in the event-flow graph) in the application's main window, the event-flow graphs were traversed, outputting the encountered event sequences as test cases. In all, 300 test cases were generated for each application.

The suites for Web applications are based on usage of the application, also referred to as user-session-based test suites [15]. These suites were previously collected by Sampath et al. [16]. A total of 125 test cases were collected for Book by asking for volunteer users by sending e-mails to local newsgroups and posting advertisements in the University of Delaware's classifieds. For CPM, 890 test cases were collected from instructors, teaching assistants, and students using CPM during the 2004-2005 and 2005-2006 academic years at the University of Delaware. A total of 169 test cases were collected when our third subject application, Masplas, was deployed for the Mid-Atlantic Symposium on Programming Languages and Systems in 2005.

Table 8 shows the characteristics of the test cases used in our study, such as the total number of test cases for each application and statistics on the lengths of the test cases. We also report the total number of unique parameter values and the percentage of 2-way parameter value interactions covered in the test suites. We compute the percentage of 2-way parameter value interactions by counting the number of unique parameter values on each window that can be selected in combination with unique parameter values on other windows within the application. For instance, if window 1 has four parameter values (1, 2, 3, and 4) and window 2 has two parameter values (5 and 6), there are $4 * 2 = 8$ parameter-value interactions. These parameter-value interactions include: $(1, 5), (2, 5), (3, 5), (4, 5), (1, 6), (2, 6), (3, 6)$, and (4, 6). We do not consider constraints of invalid interactions here, but may consider it in future work.

5.3 Faults

Each of our applications has a preexisting *fault matrix*, i.e., a representation of a set of faults known to be detected by each test case. Recall from Section 2 that GUI/Web testing refers to exercising the entire application code with the intent of identifying failures that manifest themselves in the output GUI widgets/response HTML pages, respectively. Therefore, the faults in our study are seeded in the underlying application code and the test oracles analyze the output GUI widgets/response HTML pages for failures. These faults were similar to those described in earlier reported research for the TerpOffice applications [1] and for the Web-based applications [16].

During fault seeding of the GUI applications, classes of known faults were identified, and several instances of each fault class were artificially introduced into the subject program code at relevant points. Care was taken so that

1. the artificially seeded faults were similar to faults that naturally occur in real programs due to mistakes made by developers,
2. faults were seeded in code that was covered by an adequate number of test cases, e.g., they were seeded in code that was executed by more than 20 percent and less than 80 percent of the test cases,
3. faults are seeded "fairly," i.e., an adequate number of instances of each fault type were seeded,
4. we avoid fault interaction, and
5. we employ multiple people to seed the faults.

Multiple fault-seeded versions of each application were created. We adopted a history-based approach to seed GUI faults, i.e., we observed "real" GUI faults in real applications. During the development of TerpOffice, a bug tracking tool called *Bugzilla*[3] was used by the developers to report and track faults in the previous version of TerpOffice while they were working to extend its functionality and developing the subsequent version. The reported faults are an excellent set of representative faults that are introduced by developers during implementation. Some examples include *modify relational operator* $(>, <, >=, <=, ==, !=)$, *negate condition statement*, *modify arithmetic operator* $(+, -, *, /, =, ++, -, +=, -=, *=, /=)$, and *modify logical operator* (&&, ||).

As described in previous work [16], faults in the Web applications were seeded manually by graduate and undergraduate students. In addition, some naturally occurring faults discovered during deployment were also seeded in the applications. In general, five types of faults were seeded into the applications—data store (faults that exercise application code interacting with the data store), logic (application code logic errors in the data and control flow), form (modifications to parameter-value pairs and form actions), appearance (faults which change the way in which the user views the page), and link (faults that change the hyperlinks location). Fault categories are not mutually exclusive. The fault matrices used in this paper are generated by using the *struct* oracle for CPM and Masplas and the *diff* oracle for Book [25]. Fault matrices used in this paper were collected using the *with_state* replay mechanism [16], where the application state is restored before each test case is replayed on the fault-seeded version to closely match the clean (non-fault-seeded) execution.

Table 8 shows the number of faults seeded in each application, and statistics on min., max., and avg. number of faults found by a test. In addition to traditional ways to evaluate the characteristics of faults seeded in the applications, we define a metric called the Fault Detection Density, which is a measure of the average number of faults detected by each test case [6]. Given a set of test cases, $t_i \in T$ and a set of faults F detected by test cases in T, let tf_i be the number of faults detected by t_i, then the fault detection density

3. http://bugs.cs.umd.edu.

$$FDD = \frac{tf_1 + tf_2 + \cdots + tf_n}{|T| * |F|}. \quad (1)$$

FDD is the ratio of the sum of the total number of faults detected by each test case tf_i and the total number of test cases |T|, normalized with respect to the total number of faults detected |F|. An FDD of 1 for a test suite indicates that each test case in the suite detects every fault. Table 8 shows the FDD for each of our subject applications. A high value for FDD means that each test case in the suite is detecting a large number of the faults. In such cases, even a random ordering of the test cases will yield an effective prioritized order. A low value for FDD indicates that each test case detects only a small number of the faults. The test suites for our applications have a relatively low FDD, in the range of 0.02-0.19. The low FDD values, in combination with our control of *Random*, reduce our threats to internal validity.

5.4 Evaluation Metrics

We study RQ1 and RQ2 using two metrics which aim to capture the effectiveness of a prioritization criterion. For evaluating the prioritization criteria, we assume that prior knowledge of the faults detected by the regression test suites is available to the testers.

Metric 1. The first evaluation metric measures the rate of fault detection of the prioritization criteria using the *Average Percentage of Faults Detected (APFD)* metric defined by Rothermel et al. [18]. APFD is a commonly used metric to evaluate the effectiveness of prioritized test orders [18]. We present the *APFD* using the notation in [18]. For a test suite T with n test cases, if F is a set of m faults detected by T, then let TF_i be the position of the first test case t in T', where T' is an ordering of T, that detects fault i. Then, the APFD metric for T' is given as

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_3 + \cdots + TF_m}{mn} + \frac{1}{2n}. \quad (2)$$

Informally, APFD measures the area under the curve that plots test suite fraction and the number of faults detected by the prioritized test case order.

Metric 2. The second evaluation metric measures the number of test cases executed before all faults are detected. We are interested in determining which prioritization criterion finds all the faults with the least number of test cases.

5.5 Implementation

We implemented each of the prioritization criteria as described in Section 4 in C++ and Perl. In all of the implementations, in case of a tie between two or more tests that meet the prioritization criterion, a random tie-breaking strategy is implemented using the RANDOM() function discussed earlier in the text that describes Fig. 2. To account for the nondeterminism introduced by random tie-breaking, we execute each prioritization criteria five times and report the average rate of fault detection, and APFD.

In addition to the criteria described earlier, we developed three controls—a greedy optimal ordering (*G-Best*), a greedy worst ordering (*G-Worst*), and a random ordering (*Random*). The *greedy criterion* (*G-Best*) uses a greedy step to select the next test case that detects the most yet-undetected faults and repeat this process until all the tests are selected.

TABLE 9
An Example Fault Matrix with Four Test Cases That Find the Set of Faults: $f = \{1, 2, 3, 4, 5\}$

Test	Faults found by test
t_1	1,2,3
t_2	1,2,5
t_3	3,4
t_4	5

This greedy algorithm does not guarantee an optimal ordering. For instance, assume that we have four test cases, as shown in the fault matrix in Table 9. In *G-Best*, test t_1 is chosen first because it covers the most unique faults. There is then a 3-way tie between tests t_2, t_3, and t_4 because each will locate exactly one new fault that t_1 did not already identify. Test case t_3 is chosen by the random tie-breaking. After this, there is another tie since both t_2 and t_4 will cover the last fault. With random tie-breaking, we assume that t_4 is chosen next, followed by t_2. All of the faults are found after three test cases in this example. However, the greedy step did not guarantee the optimal ordering. In this example, the ordering of $t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_1$ is optimal because all of the faults are found after two test cases, as opposed to three test cases for the greedy best example.

In contrast to the greedy best ordering, we define a *greedy worst ordering* (*G-Worst*) criterion, where in each iteration, the algorithm selects the next test case that covers the least uncovered faults. We repeat this until all of the tests are selected. Ordering by *Random* selects a next test uniformly at random.

5.6 Results

We now present the results for each research question.

5.6.1 RQ1: Which Prioritization Criteria Are Among the Best/Worst Criteria for Both GUI and Web Systems?

We summarize the results for this question based on two metrics: 1) APFD and 2) the number of tests used to find 100 percent of the faults. First, we present the results when the criteria are evaluated w.r.t. their rate of fault detection using the APFD metric. The APFD values for the prioritization criteria are shown in tabular form. Due to space constraints, the tables report the APFD after every 10 percent increment of the test suite execution for TerpCalc and Book only (results for other applications are presented in this paper's supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2010.12). In each table, we highlight the *best* APFD values for each increment with a bold font, along with the results for the *G-Best* control. We note that none of our prioritization criteria outperform *G-Best* nor are any worse than *G-Worst*.

For **TerpCalc**, the results in Table 10 show that prioritization by *PV-LtoS* has the overall best APFD with *2-way* slightly outperforming *PV-LtoS* in the first 10 percent of test suite execution. The prioritization criteria of *2-way*, *Weighted-Freq*, and *MFPS* are in the second tier of best prioritization criteria. For instance, *2-way* is more effective

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 10
APFD for TerpCalc (Each Increment of 10 Percent of the Test Suite Is 30 Tests)

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
1-way	65.68	76.04	85.86	87.21	88.15	88.69	89.08	89.08	89.33	89.34
2-way	**75.10**	82.5	85.14	87.15	89.43	90.18	90.67	90.82	90.89	90.9
PV-LtoS	74.65	**83.95**	**87.48**	**88.95**	**89.58**	89.86	**91.09**	**91.09**	**91.14**	**91.15**
PV-StoL	17.47	38.92	41.53	51.54	58.86	62.37	63.15	64.11	65.41	65.45
Action-LtoS	46.71	72.35	77.45	81.36	82.82	84.21	84.88	85.26	85.27	85.3
Action-StoL	66.65	69.66	78.08	80.12	81.76	82.71	83.83	84.78	84.98	84.99
UniqWin	62.3	74.51	83.31	85.94	87.06	87.44	87.74	87.8	88.03	88.04
APS	58.56	71.19	75.53	79.77	81.9	84.05	84.63	84.78	85.06	85.09
MFPS	71.42	82.43	85.75	87.75	88.13	88.45	89.07	89.56	89.73	90.07
Weighted-Freq	70.79	83.48	86.15	87.12	88.86	**90.82**	91	91.07	91.07	91.07
Random	55.08	69.58	74.51	76.89	79.91	81.28	82.15	82.81	82.88	82.99
G-Best	**90.45**	**94.84**	**95.70**	**95.70**	**96.35**	**96.35**	**96.35**	**96.35**	**96.45**	**96.45**
G-Worst	7.68	16.87	22.08	26.5	31.7	35.53	39.12	41.46	43.02	43.6

than *Weighted-Freq* and *MFPS* in the first 10 percent of the test execution, but for the remainder of the test execution, *Weighted-Freq* and *MFPS* each alternate in obtaining the second best APFD and are both in the range of 2 percent better or worse APFD of each other. The prioritization criteria of *1-way* and *UniqWin* are slow starters, but after the first 10 percent increment of test suite execution, they become more competitive. The prioritization criteria of *Action-LtoS*, *PV-StoL*, and *Random* are less effective than the other criteria.

For **Book**, the results in Table 11 show that *APS* produces the best APFD, although *1-way* maintains an APFD that is within 0.3 percent of *APS* throughout the entire test suite execution. *1-way* is 0.02 percent more effective than *APS* during 50-90 percent test suite execution. Only three additional prioritization criteria have better or equal APFDs than *Random* after all tests are executed. These include *2-way*, *PV-LtoS*, and *UniqWin*. The criteria that produce worse APFDs than *Random* include *MFPS*, *Action-LtoS*, *Weighted-Freq*, *PV-StoL*, and *Action-StoL*. This is the first study in which we have found that many of our prioritization criteria are less effective than *Random*. To study this behavior in Book's test suite, we examine the *fault detection density* of Book's test cases (shown in Table 8). A fault detection density of 1 for a test suite indicates that each test case in the suite detects every fault. As seen from Table 8, Book's test cases have an FDD of 0.59 (compared to 0.056 for CPM and 0.19 for Masplas). With a small test suite size (125 test cases) and a high FDD, *Random* has a greater chance of selecting a test case that detects several of the Web application's faults and thus creates an effective test suite order.

In summary, techniques that give priority to large numbers of parameter values (*PV-LtoS*) or large number of interactions between parameter values (*2-way*) generally tend to perform well in both the GUI and Web application domains. The frequency-based prioritization techniques created the best prioritized test orders in two of the three

TABLE 11
APFD for Book (Each Increment of 10 Percent of the Test Suite Is 12 Tests)

	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
1-way	93.44	93.44	93.44	93.44	94.96	**96.13**	**96.13**	**96.13**	**96.13**	96.13
2-way	93.22	93.22	93.22	93.22	94.69	94.69	95.62	95.62	95.62	95.62
PV-LtoS	93.11	93.11	93.11	93.11	93.11	94.47	95.56	95.56	95.56	95.56
PV-StoL	70.13	70.13	78.17	79.86	84.12	86.73	86.73	86.73	86.73	86.73
Action-LtoS	92.96	92.96	92.96	92.96	92.96	94.29	94.29	94.96	94.96	94.96
Action-StoL	63.43	69.33	75.56	80.82	81.97	86.32	86.32	86.32	86.32	86.32
UniqWin	91.93	93.35	94.13	94.87	95.19	95.19	95.19	95.19	95.58	95.58
APS	**93.74**	**94.19**	**95.88**	**95.88**	**95.88**	96.11	96.11	96.11	96.11	**96.18**
MFPS	93.33	93.33	93.33	93.33	93.33	93.82	94.2	94.36	94.49	94.69
Weighted-Freq	93.29	93.29	93.29	93.29	93.29	94.28	94.49	94.49	94.49	94.62
Random	86.36	93.7	94.52	94.86	94.86	95.11	95.27	95.56	95.56	95.57
G-Best	**94.02**	**96.38**	**96.38**	**96.38**	**96.38**	**96.38**	**96.38**	**96.38**	**96.89**	**96.89**
G-Worst	53.73	71.91	84.82	84.82	84.82	84.82	85.82	85.82	85.82	85.87

TABLE 12
Best and Worst Criteria for Prioritization

Application	Best 3	Worst 3
Calc	{*PV-LtoS*, *2-way*, *Weighted-Freq*}	{*PV-StoL*, *Action-StoL*, *Action-LtoS*}
Paint	{*PV-LtoS*, *2-way*, *MFPS*}	{*PV-StoL*, *Action-LtoS*, *UniqWin/APS*}
SSheet	{*UniqWin*, *2-way*, *1-way*}	{*PV-StoL*, *Action-LtoS*, *Action-StoL*}
Word	{*PV-LtoS*, *2-way*, *MFPS*}	{*PV-StoL*, *Action-LtoS*, *UniqWin/APS*}
Book	{*APS*, *1-way*, *2-way*}	{*Action-StoL*, *PV-StoL*, *Weighted-Freq*}
CPM	{*2-way*, *1-way*, *PV-LtoS*}	{*PV-StoL*, *Action-StoL*, *UniqWin*}
Masplas	{*Weighted-Freq*, *Action-LtoS*, *2-way*}	{*PV-StoL*, *Action-StoL*, *UniqWin*}

TABLE 13
Number of Tests for 100 Percent Fault Detection

Prio. Tech.	Calc (300)	Paint (300)	SSheet (300)	Word (250)	Book (125)	CPM (890)	Masplas (169)
1-way	272	253	258	**94**	72	740	68
2-way	283	162	**91**	168	83	**346**	58
Action-StoL	297	**151**	257	100	**65**	890	148
Weighted-Freq	**217**	267	155	195	115	804	**15**

Web applications. The Action-based prioritization techniques (*Action-LtoS* and *Action-StoL*) work better for the Web applications than the GUI applications, suggesting that it might not be enough to look at the number of actions covered when prioritizing test cases for pure GUIs; however, in some cases, they may work well for the Web-domain. *PV-StoL* is always a poor technique for prioritization, and in most cases, *Action-StoL* was the next worst prioritization technique, suggesting that prioritizing test cases such that fewer parameter values are covered early in the test execution cycle is a bad choice.

Table 12 summarizes the results for **RQ1** by showing the best 3 and worst 3 criteria. We see that *2-way* (underlined) was either the best or the second best technique in three out of the four GUI applications; it also did well for our Web applications. *PV-StoL* (also underlined) was the worst for all applications except Book, for which it was second worst. This was an encouraging result as it demonstrated that GUI and Web applications show common behavior when studied using the new model. Note that 2-way interactions for the applications will consider actions that cover at least two *different* windows, as required by our new model.

This finding is also consistent with our earlier results [5], [6] as well as that of others; previous studies have shown that combinatorial testing is useful at detecting faults that are triggered by interactions of two or more parameter-values [26] and we believe that the concept of covering combinations of parameters set to different values is useful for test suite prioritization. For instance, Kuhn et al. study 109 medical devices that have some type of software control and find that 97 percent of flaws are uncovered with 2-way interaction coverage [27]. Several other studies have also shown that 2-way interaction coverage is useful; see [28] and references therein for a summary of many of these studies.

Next, we examine the prioritization criteria using our second metric, which measures the number of tests that each prioritization technique chooses before locating 100 percent of the faults. Table 13 shows the number of tests that are executed to locate 100 percent of the faults for the seven applications. We show only the best criteria in the table. The number in parenthesis next to the application name in the heading in Table 13 represents the total number of test cases in that application.

Table 13 shows that *Weighted-Freq* finds 100 percent of the faults soonest after 217 test cases for TerpCalc and after 15 test cases for Masplas. For both applications, this is faster than the *G-Best*. As described earlier in Section 4, this is due to the greedy implementation of the *G-Best* algorithm that selects one test at a time, where each "next test" is selected to cover the maximum number of faults in relation to the previously selected tests. We also see that *Action-StoL* finds 100 percent of the faults sooner than the other prioritization criteria, using 151 test cases for TerpPaint and 65 for Book. *2-way* finds 100 percent of all faults in the fewest test cases or TerpSpreadsheet and CPM. Table 13 also shows that *1-way* is better than the other criteria in finding 100 percent of the faults in TerpWord, using 94 out of 300 test cases. Thus, four out of the 10 prioritization criteria, *1-way*, *2-way*, *Weighted-Freq*, and *Action-StoL*, consistently found 100 percent of the faults earlier than the other criteria for our seven subject applications. These results show that GUI and Web applications exhibit similar behavior for many of the criteria.

5.6.2 RQ2: Is a Combination of Different Prioritization Criteria More Effective than Any Single Criterion?

Based on the above results, we find that test cases prioritized by usage-based frequency and interaction-based criteria often perform better than other criteria. As a preliminary proof-of-concept, we examine these hybrid techniques for the Book application and test suite. We choose to examine Book because it is one example subject application in which some of the frequency-based prioritization criteria perform better than *2-way*. However, using the frequency-based prioritization criteria alone suggests that there may be room for improvement because they select blocks of 10 percent, 20 percent, or more of the test cases from the test suite without increasing the cumulative APFD. Therefore, we examine whether a hybrid combination of *2-way* with *APS*, *MFPS*, or *Weighted-Freq* can further improve the results. Again, we emphasize that this is just one proof-of-concept for a hybrid technique and that future work may examine the exponential number of hybrid possibilities. The two hybrid techniques that combine the frequency-based and interaction-based prioritization criteria are studied in the following ways:

1. Prioritize test cases by the frequency-based prioritization criteria:
   (*MFPS*, *APS*, *Weighted-Freq*) until the first 10 percent block of test cases encountered where there is no increase in APFD, then switch to interaction-based prioritization criteria (*2-way*).
2. Prioritize test cases by the frequency-based prioritization criteria:
   (*MFPS*, *APS*, *Weighted-Freq*) until the first 20 percent block of test cases encountered where there is no increase in APFD, then switch to interaction-based prioritization criteria (*2-way*).

From Table 11, we note that *APS* has a better APFD than *2-way*. When a hybrid technique is used, e.g., both *APS* and *2-way* are used to create a prioritized test order, we see in

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

TABLE 14
Book: Hybrid—Average Percentage Faults Detected

% of test suite run	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
APS and *2-way*										
2-way	93.22	93.22	93.22	93.22	94.69	94.69	95.62	95.62	95.62	95.62
APS	**93.74**	**94.19**	**95.88**	**95.88**	**95.88**	**96.11**	96.11	96.11	96.11	96.18
APS-2way-10%-no-APFD-increase	**93.74**	93.74	93.74	93.74	95.13	95.13	96.08	96.08	96.08	96.08
APS-2way-20%-no-APFD-increase	**93.74**	**94.19**	**95.88**	**95.88**	**95.88**	95.88	**96.92**	**96.92**	**96.92**	**96.92**
MFPS and *2-way*										
MFPS	**93.33**	**93.33**	**93.33**	**93.33**	93.33	94.28	94.49	94.49	94.49	94.69
MFPS-2way-10%-no-APFD-increase	**93.33**	**93.33**	**93.33**	**93.33**	**94.78**	**94.78**	**95.73**	**95.73**	**95.73**	**95.73**
MFPS-2way-20%-no-APFD-increase	**93.33**	**93.33**	**93.33**	**93.33**	**94.78**	**94.78**	**95.73**	**95.73**	**95.73**	**95.73**
Weighted-Freq and *2-way*										
Weighted-Freq	**93.29**	**93.29**	**93.29**	**93.29**	93.29	94.4	94.4	94.4	94.4	94.62
WF-2way-10%-no-APFD-increase	**93.29**	**93.29**	**93.29**	**93.29**	**94.73**	**94.73**	**95.69**	**95.69**	**95.69**	**95.69**
WF-2way-20%-no-APFD-increase	**93.29**	**93.29**	**93.29**	**93.29**	**94.73**	**94.73**	**95.69**	**95.69**	**95.69**	**95.69**

Table 14 that the prioritized test cases by the hybrid technique of *APS-2way-20%-no-APFD-increase* performs better than *2-way* in the latter 30 percent of the test suite execution. The *APS-2way-10%-no-APFD-increase* does not increase the APFD over *APS* alone. When we combine *MFPS* and *2-way*, we find that the hybrid techniques *MFPS-2way-10%-no-APFD-increase* and *MFPS-2way-20%-no-APFD-increase* achieve the best overall APFD. Also, when *Weighted-Freq* and *2-way* are combined to create hybrid prioritization techniques, the hybrid techniques create test orders that achieve higher APFDs than the two techniques *Weighted-Freq* or *2-way* alone. In all but one case here, we find that these hybrid techniques improve the APFD over *2-way* or *APS* alone. Again, this is just one proof-of-concept example and future work may examine the APFD of hybrid techniques in further detail.

Table 15 presents the results for RQ2 using the second metric, which measures the number of test cases needed to locate 100 percent of the faults. We find that the hybrid techniques find 100 percent faults quicker than the control *Random*, but neither are as effective as *2-way* or *Weighted-Freq*. This exploratory evaluation of RQ2 suggests that hybrid techniques may have a better rate of fault detection than the individual techniques. However, the hybrid techniques do not necessarily find 100 percent of the faults earlier. Further study is required on the hybrid techniques and their effectiveness.

TABLE 15
Book: Number of Tests Run to Find 100 Percent of All Faults

Prioritization Technique	No. of tests for 100% fault detection	No. of additional or fewer tests for 100% fault detection in comparison to Random ordering
1-way	72	-52
2-way	83	-41
PV-LtoS	76	-48
PV-StoL	68	-56
Action-LtoS	96	-28
Action-StoL	64	-60
APS	124	0
MFPS	118	-6
MFPS-2way-10%-no-APFD-increase	82	-42
MFPS-2way-20%-no-APFD-increase	82	-42
Weighted-Freq	115	-9
APS-2way-10%-no-APFD-increase	82	-42
APS-2way-20%-no-APFD-increase	80	-44
MFPS-2way-10%-no-APFD-increase	82	-42
WF-2way-20%-no-APFD-increase	82	-42
MFPS-2way-20%-no-APFD-increase	90	-34
Random	124	0
G-Best	19	-105
G-Worst	124	0

6 Conclusions and Future Work

Previous work treats stand-alone GUI and Web-based applications as separate areas of research. However, these types of applications have many similarities that allow us to create a single model for testing such event-driven systems. This model may promote future research to more broadly focus on stand-alone GUI and Web-based applications instead of addressing them as disjoint topics. Other researchers can use our common model to apply testing techniques more broadly. Within the context of this model, we develop and empirically evaluate several prioritization criteria and apply them to four stand-alone GUI and three Web-based applications and their existing test suites. Our empirical study evaluates the prioritization criteria. Our ability to develop prioritization criteria for two types of event-driven software indicates the usefulness of our combined model for the problem of test prioritization. Our results are promising as many of the prioritization criteria that we use improve the rate of fault detection over random ordering of test cases. We learn that prioritizations by *2-way* and *PV-LtoS* generally result in the best improvement for the rate of fault detection in our GUI applications and one of our Web applications. However, for our Web applications, frequency-based techniques provide the best rate of fault detection in two out of the three subjects. We attribute this to the source of the test cases. The test suites for the Web applications come from real user-sessions, whereas the GUI test cases were automatically generated without influence from users. While the majority of prioritization techniques provide benefits in our study, we caution readers that two techniques, *Action-StoL* and *PV-StoL*, generally provided the worst rates of fault detection. This was expected as we anticipated that test cases that do not exercise much functionality are less likely to find faults. As a proof-of-concept, we examine a hybrid technique that uses combinations of multiple prioritization criteria. These preliminary results motivate future research on hybrid prioritization criteria.

We present our threats to validity in this section because several opportunities for future research are created by the threats to validity of the results of our empirical study. For example, threats to external validity are factors that may impact our ability to generalize our results to other situations. The first threat is the validation of the unified model. We validate the model through the application of test suite prioritization by using several prioritization criteria and three controls applied to seven applications. While this work contributes an initial validation of the model, the domains of both testing and EDS are much larger. For instance, broader testing activities such as test generation and test suite reduction can further validate the unified model in the future. In regard to EDS, we use GUI and Web-based applications. Future work may examine a different type of EDS, such as embedded systems, in this model. The second largest threat to external validity is that we only run our data collection and test suite prioritization process on seven programs and their existing test suites, which we chose for their availability. These programs range from 999 to 18,376 lines of code, contain 9 to 219 classes, 22 to 664 methods, and 108 to 1,521 branches. However, these programs may still not be representative of the broader population of programs. An experiment that would be more readily generalized would include multiple programs of different sizes and from different domains. In the future, we may conduct additional empirical studies with larger EDS to address this threat. Moreover, the characteristics of the original test suites impact our results in how they were constructed and their fault detecting ability. The seeded faults also impact the generalization of our results. We provide the FDD values for each test suite and use *Random* as a control to compare our prioritization techniques in order to minimize this threat. Future work may examine real systems that have real faults that were not seeded.

Threats to construct validity are factors in the study design that may cause us to inadequately measure concepts of interest. In our study, we made simplifying assumptions in the area of costs. In test suite prioritization, we are primarily interested in two different effects on costs. First, there is potential savings obtained by running "more effective" test cases sooner. In this study, we assume that each test case has a uniform cost of running (processor time) and monitoring (human time); these assumptions may not hold in practice. Second, we assume that each fault contributes uniformly to the overall cost, which again may not hold in practice. Future work may examine projects with readily available data on the costs of faults. Another threat to construct validity is that we report results in increments of 10 percent for the hybrid experiments. In the future, we may report the results in different increments (as revealed by a statistical method) that are more appropriate to the increments used to combine criteria in the hybrid experiments.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers whose comments and suggestions helped to extend the second research question, reshape its results, and improve the flow of the text. This work was partially supported by the US National Science Foundation (NSF) under NSF grants CCF-0447864 and CNS-0855055, and US Office of Naval Research grant N00014-05-1-0421.

REFERENCES

[1] A.M. Memon and Q. Xie, "Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 884-896, Oct. 2005.

[2] A. Andrews, J. Offutt, and R. Alexander, "Testing Web Applications by Modeling with FSMs," *Software and Systems Modeling*, vol. 4, no. 3, pp. 326-345, July 2005.

[3] G.D. Lucca, A. Fasolino, F. Faralli, and U.D. Carlini, "Testing Web Applications," *Proc. IEEE Int'l Conf. Software Maintenance*, pp. 310-319, Oct. 2002.

[4] F. Ricca and P. Tonella, "Analysis and Testing of Web Applications," *Proc. Int'l Conf. Software Eng.*, pp. 25-34, May. 2001.

[5] R.C. Bryce and A.M. Memon, "Test Suite Prioritization by Interaction Coverage," *Proc. Workshop Domain-Specific Approaches to Software Test Automation in Conjunction with Sixth Joint Meeting of the European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 1-7, Sept. 2007.

[6] S. Sampath, R. Bryce, G. Viswanath, V. Kandimalla, and A.G. Koru, "Prioritizing User-Session-Based Test Cases for Web Application Testing," *Proc. IEEE Int'l Conf. Software Testing, Verification, and Validation*, pp. 141-150, Apr. 2008.

[7] P. Brooks, B. Robinson, and A.M. Memon, "An Initial Characterization of Industrial Graphical User Interface Systems," *Proc. IEEE Int'l Conf. Software Testing, Verification, and Validation*, pp. 11-20, 2009.

[8] L. White, "Regression Testing of GUI Event Interactions," *Proc. Int'l Conf. Software Maintenance*, pp. 350-358, Nov. 1996.

[9] "Web Site Test Tools and Site Management Tools," http://www.softwareqatest.com/qatweb1.html, Apr. 2009.

[10] D.C. Kung, C.-H. Liu, and P. Hsia, "An Object-Oriented Web Test Model for Testing Web Applications," *Proc. First Asia-Pacific Conf. Quality Software*, pp. 111-120, Oct. 2000.

[11] W. Wang, S. Sampath, Y. Lei, and R. Kacker, "An Interaction-Based Test Sequence Generation Approach for Testing Web Applications," *Proc. IEEE Int'l Conf. High Assurance Systems Eng.*, pp. 209-218, 2008.

[12] W. Halfond and A. Orso, "Improving Test Case Generation for Web Applications Using Automated Interface Discovery," *Proc. Sixth Joint Meeting of the European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, pp. 145-154, Sept. 2007.

[13] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst, "Finding Bugs in Dynamic Web Applications," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 261-272, July 2008.

[14] N. Alshahwan and M. Harman, "Automated Session Data Repair for Web Application Regression Testing," *Proc. IEEE Int'l Conf. Software Testing, Verification, and Validation*, pp. 298-307, Apr. 2008.

[15] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II, "Leveraging User Session Data to Support Web Application Testing," *IEEE Trans. Software Eng.*, vol. 31, no. 3, pp. 187-202, May 2005.

[16] S. Sampath, S. Sprenkle, E. Gibson, L. Pollock, and A.S. Greenwald, "Applying Concept Analysis to User-Session-Based Testing of Web Applications," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 643-658, Oct. 2007.

[17] K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression Testing in an Industrial Environment," *Comm. ACM*, vol. 41, no. 5, pp. 81-86, May 1988.

[18] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.*, vol. 27, no. 10, pp. 929-948, Oct. 2001.

[19] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 159-182, Feb. 2002.

[20] D. Binkley, "Using Semantic Differencing to Reduce the Cost of Regression Testing," *Proc. Int'l Conf. Software Maintenance*, pp. 41-50, Nov. 1992.

[21] J.A. Jones and M.J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 195-209, Mar. 2003.

[22] D. Jeffrey and N. Gupta, "Test Case Prioritization Using Relevant Slices," *Proc. Int'l Computer Software and Applications Conf.*, pp. 411-418, Sept. 2006.
[23] J. Lee and X. He, "A Methodology for Test Selection," *J. Systems and Software,* vol. 13, no. 3, pp. 177-185, Nov. 1990.
[24] J. Offutt, J. Pan, and J.M. Voas, "Procedures for Reducing the Size of Coverage-Based Test Sets," *Proc. Int'l Conf. Testing Computer Software,* pp. 111-123, June 1995.
[25] S. Sprenkle, L. Pollock, H. Esquivel, B. Hazelwood, and S. Ecott, "Automated Oracle Comparators for Testing Web Applications," *Proc. Int'l Symp. Software Reliability Eng.*, pp. 253-262, Nov. 2007.
[26] M. Grindal, J. Offutt, and S. Andler, "Combination Testing Strategies: A Survey," *Software Testing, Verification and Reliability,* vol. 15, pp. 167-199, Mar. 2005.
[27] D.R. Kuhn, D.R. Wallace, and A.M. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 418-421, Oct. 2004.
[28] C.J. Colbourn, "Combinatorial Aspects of Covering Arrays," *Le Matematiche,* vol. 58, (Catania), pp. 121-167, 2004.

Renée C. Bryce received the BS and MS degrees from Rensselaer Polytechnic Institute and the PhD degree from Arizona State University. She is an assistant professor at Utah State University. Her research interests include software testing, particularly combinatorial testing, test suite prioritization, and usability testing. She has served on the program committee of the International Conference on Software Testing, Verification and Validation (ICST) and the International Workshop on TESTing Techniques and Experimentation Benchmarks for Event-Driven Software (TESTBEDS).

Sreedevi Sampath received the BE degree in computer science and engineering from Osmania University in 2000 and the MS and PhD degrees in computer and information sciences from the University of Delaware in 2002 and 2006, respectively. She is an assistant professor in the Department of Information Systems at the University of Maryland, Baltimore County. Her research interests are in the areas of software testing, Web applications, and software maintenance. She is interested in regression testing and test generation for Web applications and in exploring uses of Web application usage data. She has served on the program committees of conferences such as the International Conference on Software Testing, Verification, and Validation (ICST), International Conference on Empirical Software Engineering and Measurement (ESEM), and International Symposium on Software Reliability Engineering (ISSRE). She is a member of the IEEE Computer Society.

Atif M. Memon is an associate professor in the Department of Computer Science at the University of Maryland, College Park. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He is the inventor of the GUITAR system (http://guitar.sourceforge.net/) for automated model-based GUI testing. He is the founder of the International Workshop on TESTing Techniques and Experimentation Benchmarks for Event-Driven Software (TESTBEDS). He serves on various editorial boards, including that of the *Journal of Software Testing, Verification and Reliability*. He has served on numerous US National Science Foundation (NSF) panels and program committees, including the International Conference on Software Engineering (ICSE), International Symposium on the Foundations of Software Engineering (FSE), International Conference on Software Testing, Verification, and Validation (ICST), Web Engineering Track of the International World Wide Web Conference (WWW), Working Conference on Reverse Engineering (WCRE), International Conference on Automated Software Engineering (ASE), and International Conference on Software Maintenance (ICSM). He is currently serving on a National Academy of Sciences panel as an expert in the area of computer science and information technology for the Pakistan-US Science and Technology Cooperative Program sponsored by the United States Agency for International Development (USAID). In addition to his research and academic interests, he handcrafts fine wood furniture. He is a member of the IEEE Computer Society.