Developing A Generic Model and Ordering Test Cases For Embedded Software

M.Satyanarayana, V.Sri Lakshmi
Swarnandhra College of Engg & Tech, Narsapuram, AP, India

Abstract
Event-Driven Software (EDS) can change state based on incoming events; common examples are GUI and web applications. These EDS 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 all pairs of event interactions did well for GUI and web applications, and 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.

Keywords

I. Introduction
In this paper, we focus on the first challenge: i.e., Event-driven software (EDS) is a class of software that is we develop a single abstract model for GUI and web quickly becoming ubiquitous. All EDS take sequences of application testing. To provide focus, we restrict the model events (e.g., messages, mouse-clicks) as input, change their to extend our previous work on test prioritization techniques state, and produce an output (e.g., events, system calls, text for GUI [5] and web testing [6]. This allows us to tailor our messages). Examples include web applications, graphical user model to prioritization-specific issues as well as to recast our interfaces (GUIs), network protocols, device drivers, and previous prioritization criteria in a form that is general embedded software. Enough to leverage the single model. In the future, we will extend our model to other testing problems that are shared testing for functional correctness of EDS such as GUI and web applications. Our ultimate goal is to stand-alone GUI and web-based applications is critical to many generalize the model and to develop a theory of how EDS organizations. These applications share several important should be tested. The specific contributions of this work characteristics. Both are particularly challenging to test include: the first single model for testing stand-alone GUI because users can invoke many different sequences of events and web-based applications, a shared prioritization function that affect application behavior. Earlier research has shown based on the abstract model, and shared prioritization that existing conventional testing techniques do not apply to criteria. We validate the usefulness of these artifacts through either GUIs or web applications, primarily because the number an empirical study. The results show that GUI and web-of permutations of input events leads to a large number of based applications, when recast using the model, showed states, and for adequate testing, an event may need to be tested similar behavior, reinforcing our belief that these classes of in many of these states, thus requiring a large number of test applications should be modeled and studied together. Other cases (each represented as an event sequence). Researchers results show that GUI and web applications behave have developed several models for automated GUI testing [1] differently, which has created opportunities for evolving the and web application testing [2]–[4]. Despite the above model and further experimentation. In future work, we will similarities of GUI and web applications, all the efforts to further generalize the model by evaluating its applicability address their common testing problems have been made and usefulness for other software testing activities, such as separately due to two reasons. First, is the challenge of coming test generation. Our study also makes contributions towards up with a single model of these applications that adequately test prioritization research. Many of our prioritization criteria captures their event-driven nature, yet abstracts away elements improve the rate of fault detection of the test cases over that are not important for functional testing. The absence of random orderings of tests. We also develop hybrid such a model has prevented the development of shared testing prioritization criteria that combine several criteria that work techniques and algorithms that may be used to test both classes well individually and evaluate whether the hybrid criteria of applications. It has also prevented the development of a result in more effective test orders. shared set of metrics that may be used to evaluate the test results of these types of applications.

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

A. GUI-Based Applications
A GUI is the front-end to a software’s underlying backend 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, frames) they 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 may be triggered that causes the execution of a Java script event handler, which in turn results in the link changing color; 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 a form and clicks on the submit button, the data is 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 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]–[4]. 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

III. 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 action performed message is sent to all action listeners that are registered on the relevant component. 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 did not model the fact that the Ok button is the only event that actually causes changes to the application’s settings.

A. 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.

Fig. 1(a): Example GUI Application Window

Parameter, Value

---Find what! drop-box, settext>  
---Find what! drop-box, leftclick dropdown>  
---Match case checkbox, leftclick select>  
---Match case checkbox, leftclick unselect>  
---Find whole words only checkbox, leftclick select>  
---Find whole words only checkbox, leftclick unselect>  
---Find Next! button, leftclick>  
---Close! button, leftclick>  
---Replace button, leftclick>

Nine parameter-values on the GUI window

Fig. 1(b): Example web application window

Parameter and value descriptions for Login.jsp

---Login text field, guest >  
---Password text field, guest >  
---FormAction, Login >
Table 1: Unified Terminology for GUI and Web-Based Applications

<table>
<thead>
<tr>
<th>GUI</th>
<th>Web application</th>
<th>Term in paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>window page</td>
<td>widget (e.g., text field)</td>
<td>a setting for a parameter value</td>
</tr>
<tr>
<td>window</td>
<td>parameter (e.g., form field) parameter</td>
<td>a parameter value</td>
</tr>
<tr>
<td>(e.g., a box is checked) (e.g., a name in a text field or a param _userid'*)</td>
<td>the pair of widget name, and its setting</td>
<td>(e.g., &lt;‗Find' box, checked&gt;) (e.g., &lt;userid, guest&gt;)</td>
</tr>
<tr>
<td></td>
<td>the pair of parameter-name and its setting</td>
<td>(PV)</td>
</tr>
<tr>
<td>a sequence of user-interactions that set one or a sequence of user-interactions that set one or action</td>
<td>more parameter-values</td>
<td>on a single window</td>
</tr>
<tr>
<td></td>
<td>more parameter-values</td>
<td>on a single page</td>
</tr>
<tr>
<td>window page</td>
<td>widget (e.g., text field)</td>
<td>a setting for a parameter value</td>
</tr>
<tr>
<td>window</td>
<td>parameter (e.g., form field) parameter</td>
<td>a parameter value</td>
</tr>
<tr>
<td>(e.g., a box is checked) (e.g., a name in a text field or a param _userid'*)</td>
<td>the pair of widget name, and its setting</td>
<td>(e.g., &lt;‗Find' box, checked&gt;) (e.g., &lt;userid, guest&gt;)</td>
</tr>
<tr>
<td></td>
<td>the pair of parameter-name and its setting</td>
<td>(PV)</td>
</tr>
</tbody>
</table>

Table provides a sample test case for a GUI application called TerpWord. The test case sets 9 parameters to values and visits 3 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 ―example File‖, 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 Next button. Table 2a summarizes the windows, parameters, and values in this test case and assigns unique numbers to each window and action.

IV. 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 prioritization 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 2 out of the 3 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.
References


