# Human Performance Regression Testing

Amanda Swearngin, Myra B. Cohen
Dept. of Computer Science & Eng.
University of Nebraska-Lincoln, USA
Lincoln, NE 68588-0115
{aswearn,myra}@cse.unl.edu

Bonnie E. John, Rachel K. E. Bellamy
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
{bejohn,rachel}@us.ibm.com

*Abstract*—As software systems evolve, new interface features such as keyboard shortcuts and toolbars are introduced. While it is common to regression test the new features for functional correctness, there has been less focus on systematic regression testing for usability, due to the effort and time involved in human studies. Cognitive modeling tools such as CogTool provide some help by computing predictions of user performance, but they still require manual effort to describe the user interface and tasks, limiting regression testing efforts. In recent work, we developed CogTool-Helper to reduce the effort required to generate human performance models of existing systems. We build on this work by providing task specific test case generation and present our vision for human performance regression testing (HPRT) that generates large numbers of test cases and evaluates a range of human performance predictions for the same task. We examine the feasibility of HPRT on four tasks in LibreOffice, find several regressions, and then discuss how a project team could use this information. We also illustrate that we can increase efficiency with sampling by leveraging an inference algorithm. Samples that take approximately 50% of the runtime lose at most 10% of the performance predictions.

## I. INTRODUCTION

Regression testing (testing after modifications to detect faults that have been introduced by changes, e.g., [1], [2]), has become best practice in the development of commercial software. A large body of research, automated processes, and tools have been created to solve problems related to regression testing for functional correctness. Regression testing of quality attributes, like system response time, has also received attention (e.g., [3], [4]), and there is research into incorporating human expertise to increase test efficiency [5]. But regression testing of one important quality, usability, has remained largely untreated in the software engineering (SE) literature. Regression testing of usability is an important consideration in software development because as systems grow in functionality, they often also grow in complexity, with more features added to user interfaces, which can hurt end-user efficiency and discoverability (i.e., the ability for a new user to discover how to accomplish a task through exploring the interface). For instance, adding toolbars to an interface should, in theory, increase user efficiency (a factor of usability) because only one mouse action is necessary to use an always-visible toolbar as opposed to two or more mouse actions to pull down a menu and select a command. However, the positioning of the toolbar may move other user interface (UI) elements further away from where skilled users need them for common tasks, necessitating longer mouse movements and, in fact, decrease efficiency. In addition, many toolbars with small icons may add screen clutter and may decrease a new user's ability to discover how to accomplish a task over a simpler UI design.

Usability testing has traditionally been empirical, bringing end-users in to a testing facility, asking them to perform tasks on the system (or prototype), and measuring such things as the time taken to perform the task, the percentage of end-users who can complete the task in a fixed amount of time, and the number and type of errors made by the end-users. Both collecting and analyzing the human data is time consuming and expensive. Since regression testing is typically resource constrained [1], [2], the manual effort required to perform usability testing means that regression testing for usability is often intractable.

As a consequence, research in the field of human-computer interaction (HCI) has produced methods and tools for predictive human performance modeling using "simulated end-users" [6]–[9]. Research into predictive human performance modeling has a 40-year history in HCI that has produced theories of human behavior, operationalized in computational models, that can reliably and quantitatively predict some aspects of usability, e.g., the time a skilled end-user would take to complete a task [7], [8], the time it would take an end-user to learn methods to accomplish tasks [9], and, more recently, the behavior novice end-users would display when attempting to accomplish tasks on new interfaces, including the errors they would make [6], [8], [10]. UI designers find such models valuable in their development process, from evaluating early UI design ideas before implementation to evaluating proposed systems during procurement [9], [11].

Although not as time-intensive as empirical testing with human participants, modeling still involves substantial human effort because the UI designer must construct the human performance model and enumerate the different ways that a task can be accomplished on the interface. Thus, the number of tasks, and alternative methods for accomplishing those tasks, that can be analyzed are still limited by the time and resources available for usability evaluation. Typically, a UI designer, even one skilled in using a human performance modeling tool, will only evaluate a handful of tasks, with only one or two methods per task (e.g., using the mouse and menus to perform the task, vs. using keyboard shortcuts).
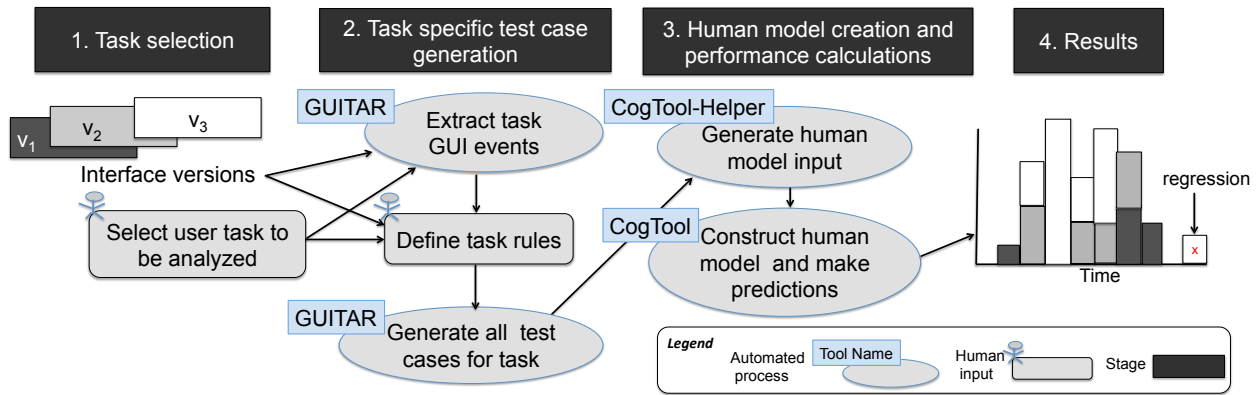
Fig. 1. Overview of Human Performance Regression Testing (HPRT)

In recent work, we proposed merging research from the SE community on Graphical User Interface (GUI) testing with cognitive modeling tools from the HCI community to automate the creation of the human performance models. We built a tool, *CogTool-Helper* [12], that automates some of the process of defining tasks and building human performance models for running systems. However, the UI designer still needs to specify each method an end-user would use to do each task, e.g., using the menus vs. using keyboard shortcuts, a level of manual effort that still limits the number of tasks and methods that can be analyzed in any iteration of the UI design.

We now propose a tighter integration of SE research on regression testing with HCI research in human performance modeling. This will enable us to perform extensive testing of user performance, in what we term *human performance regression testing* (HPRT), that will result in usability testing of an implemented UI on a scale that was previously impossible.

In this paper, we detail our approach to HPRT and implement a prototype to evaluate its usefulness. We first describe a technique to generate functional test cases that mimics a real user. We then exercise our prototype for HPRT on four tasks using three versions of an open source application interface, and show that not only is this approach feasible, but it provides some interesting and unexpected results.

The contributions of this work are:

1) A novel approach for human performance regression testing (HPRT) that:
   a) Performs test case generation of realistic user tasks.
   b) Results in visualizations of user performance across a range of methods to accomplish the same task.
2) A feasibility study showing that HPRT provides useful information, and that an inference algorithm allows us to increase the efficiency of HPRT without much loss of effectiveness.

## II. OVERVIEW OF HPRT

As an illustration, Figure 1, shows an overview of HPRT (the figure points to specific existing SE and HCI tools, which will be explained in the next section, but the process is more general than these specific tools). The process has four main stages, numbered and named in black rectangles. In this example, a UI designer examines an existing interface ($V_1$) and sees that the only way a user can invoke a command is through menus. Assume the UI designer's goal is to increase efficiency for skilled users (one common usability metric). To meet this goal, the designer adds keyboard shortcuts to the interface ($V_2$), and in a subsequent version, ($V_3$), adds toolbars. At each interface feature addition, the UI designer should validate that the new features have indeed improved efficiency, or at least, have not decreased efficiency of human performance (i.e. have not caused a regression).

In Stage 1, the UI designer chooses tasks to evaluate; we'll use *select a block of text and make it bold* as an example task. (In practice many, and more complicated, tasks would be evaluated). In ($V_1$), there is one way for an end-user to do this task, i.e., using menus. But as keyboard shortcuts ($V_2$) and toolbars ($V_3$) are added, the user may use pure (e.g., only keyboard shortcuts) or mixed methods (e.g. using both menu and toolbar) to achieve this task. The tasks to be evaluated and the running versions of the system are the inputs for HPRT.

In Stage 2 of HPRT, task specific test cases are generated. The UI designer uses a GUI functional testing tool, e.g., *GUI-TAR* tool [13] to extract a model of the subset of events that can be used to achieve the task from the running interface. This model represents all possible (and partial) ways to perform this task, many of which make no sense from the perspective of evaluating the efficiency of skilled users. For example, this model can include a path that opens a dialog box and closes it immediately without making any changes to it, unnecessary steps a skilled user would not perform. Inspired by Memon et al.'s work restricting test case generation by writing rules [14], HPRT uses rules provided by the UI designer to constrain its test case generation to those that make sense for evaluating usability, i.e., *humanly-reasonable* methods that a skilled user might follow to accomplish a task. For instance, since both select and bold must be included in our example task for it to be complete, each event must occur once, and the UI designer writes a rule to enforce that requirement. Furthermore, an experienced end-user would make the text bold only once, i.e., with *either* the menu *or* the keyboard shortcut, but not

both, so the UI designer writes a rule to make these events mutually exclusive. Finally, each of these humanly-reasonable methods is turned into a *test case*[1] using a customized version of the GUITAR [13] test case generator.

In the third stage, the test cases are imported into *CogTool-Helper* [12], which recreates the UI design and the test cases in such a way that it can be fed into a human performance modeling tool such as *CogTool* [15]. CogTool runs a simulated user to produce an estimate of the time it would take an experienced end-user to perform that task using each of the test cases on the applicable versions of the UI. The results from each of the versions of the system can then be presented as a histogram (shown as stage 4 to the right in Figure 1), so that the UI designer has an understanding of the range of performance times that are possible given the interface and task. In this example, we see a regression, where $V_3$ (in white in the histogram) has an outlier time-wise, caused by adding toolbars. This result may have arisen from a test case that switches between using the toolbar and using either menus or keyboard shortcuts, requiring a lot of extra hand movement by the user – something the UI designer did not consider. Had the designer examined only one or two test cases for each version of the UI, as is the state-of-the-practice, this regression might have been missed, because most of the other data points lie below the time taken in the previous versions. Given the results of HPRT, the UI designer can now drill down into the test case that causes this regression, determine if this is likely to be a problem for real users, and if so, can fix the UI or train users to avoid the aberrant procedure.

## III. BACKGROUND AND RELATED WORK

The process of HPRT emerges from prior work in both SE and HCI. In this section we present the prior work in these fields on which we base HPRT.

### A. Automated GUI Testing in SE

Techniques and tools for automatic regression testing of the graphical user interface (GUI) for functional correctness are abundant in the SE literature, (e.g., [14], [16], [17]). Automated regression testing in GUIs requires (1) a model of the interface, (2) a technique to generate test cases and (3) a method for replaying the test cases on the interface. Approaches include representing GUIs as finite state machines [18], as graph models [14], [16], [17], [19], or through visual elements such as buttons and widgets [20].

Our proposal for HPRT is most similar to the work of Memon et al. [16], where GUI interfaces are represented as an event flow graph (EFG), which describes the edges between events (i.e. which events can follow others) and their type (e.g. structural or system interaction). An event is the result of an event-handler in the application responding to a user or program interaction with the interface, e.g., by clicking a menu or button, or typing a keyboard shortcut. An EFG can be

extracted from a GUI using a tool called a ripper [17], which performs a single execution of the program, using a depth first traversal of the interface, opening all menus, windows and reachable widgets and buttons. As it performs these events, it extracts the *state* of the interface (each of the widgets, their types and edges, properties and values, etc.).

Test cases can be generated from the EFG by enumerating sequences of nodes along paths in the graph. Existing test case generation algorithms use criteria such as selecting all single edge paths (all length 2 test cases), generating all paths of a particular length, or randomly selecting paths in the EFG. Once generated, test cases can then be replayed on the GUI. Such test cases can provide feedback on the functional correctness and completeness of GUIs, but give no insight into the usability of the GUI.

As mentioned in the overview, typically not all test cases generated for functional testing make sense from a usability perspective; it is quite possible that a test case generated from an EFG would click the bold toolbar button multiple times, or click the bold toolbar button before text has been selected. Memon et. al. [14] used hierarchical AI planning and constraint rules to generate more "meaningful" GUI test cases. Their key conjecture was that "the test designer is likely to have a good idea of the possible goals of a GUI user and it is simpler and more effective to specify these goals than to specify sequences of events that the user might employ to achieve them." Our approach is similar, except that Memon et. al. included only *system interaction events* (e.g., that a command to make text bold was invoked, but not how it was invoked), whereas our work explicitly includes structural alternatives on the interface that can impact user performance (e.g., whether an event is invoked by clicking the mouse or with a keyboard shortcut). In addition, we don't use an AI planner for generation, but use a functional test generator and then drop test cases that violate the rules.

We use Memon's GUI Testing Framework, GUITAR, [13], a set of open source tools available for a variety of user interface platforms, as the basis for our prototype for HPRT. It includes a ripper, a replayer and functional test case generator. We modified the ripper to extract sub-graphs of events, added the ability to represent keyboard shortcuts, and modified the test case generator to check constraint rules (see Section IV).

### B. Predictive Human Performance Modeling

Research into predictive human performance modeling has a 30-year history in HCI, but new tools have made it possible for UI practitioners to create valid human performance models without having PhD-level knowledge of cognitive and perceptual psychology. With one such tool, CogTool [15], a UI designer can create a storyboard of an interface without programming and demonstrate methods for accomplishing tasks on that storyboard. (A *task* is a goal that will be achieved on the interface such as creating a table, or inserting the date. *Methods* are concrete steps in the UI that accomplish the task.) CogTool captures each demonstration and creates a computational cognitive model of a skilled end-user based on

---

[1]Regression testing calls each path from a start state to an end state a *test case*, while human performance modeling calls it a *method*. These terms are equivalent and will be used interchangeably.

the Keystroke-Level Model (KLM) [7]. CogTool then runs this model producing a quantitative prediction of mean execution time by a skilled user. In practice, UI designers have used these predictions in many aspects of the software development process, from competitive analysis to evaluating proposed design ideas, to assessing contract compliance [11].

Although CogTool is an order-of-magnitude faster than previous tools for creating models [15], CogTool still requires UI designers to build a storyboard of the existing or proposed UI against which the models can be run, which, depending on how much of the UI is being tested, can take the UI designer hours or days. (A *storyboard* is a common tool of UI designers, comprised of a series of pictures of what end-users would see as they work with the UI and the user actions, e.g., click this button, type that text, that transition from one picture to another.) When evaluating efficiency, the storyboard need only represent the correct paths through the UI for each task and the UI designer demonstrates those paths on the storyboard. CogTool produces the human performance models from the storyboard and demonstrations, and runs the models to extract quantitative predictions of skilled task execution time. Although we are using the efficiency usability metric to illustrate HPRT in this paper, CogTool can also predict new-user exploration behavior (including errors) [10]. For this type of evaluation, the UI designer must specify the end-users' task goal and a full storyboard with all the widgets in the UI for the human performance model to explore. Imagine re-creating the entire interface of OpenOffice manually from screenshots, hotspots, and links; this is too much manual effort to allow frequent testing on rapidly changing interfaces.

### C. Merging the Two: CogTool-Helper

In [12] we presented *CogTool-Helper*, a tool that links Cog-Tool [15] and GUITAR [13] to reduce the human effort needed to produce CogTool predictions. CogTool-Helper generates storyboards and task methods from existing applications and delivers them to CogTool. After some initial setup in which the UI designer points CogTool-Helper to the application that he/she will evaluate, the UI designer creates one or more tasks either by demonstrating one or more methods for each task on the application (i.e., *capture*) or by directly encoding methods (i.e., test cases) in an XML format.

CogTool-Helper then opens the application, extracts needed information from the menus (which is analogous to a partial ripping of the interface), replays the test cases (taking screen-shots, and recording the corresponding UI state as the set of visible widgets and their properties, in the process). It then encodes information about the UI widgets (e.g., type, size and position). The result is a translation into a CogTool project XML that contains a UI design storyboard, one or more tasks, and one or more test cases for each task. The design storyboard is a graph with nodes made up of the unique UI states (shown as screenshots taken while replaying the test cases), and edges that represent the user actions that produced the next screenshot. From this graph, CogTool-Helper infers methods not explicitly provided by the UI designer by traversing the graph to identify alternative paths between the start state and end state of the task. In [12], we saw as high as a 75% increase in the number of methods over those manually specified, due to the inference algorithm. We evaluate the impact of this capability on performance of HPRT, as our second research question in Section VI.

The output of CogTool-Helper is an XML project that can be imported into and analyzed in CogTool. Once imported, CogTool produces the valid human performance model, runs it, and produces quantitative predictions of skilled end-user performance time for each method within each task. The UI designer can then use CogTool to explore differences between GUIs (e.g., competitors or previous versions of a system) or modify the UI designs and see if the modifications can make a new version more efficient.

Although CogTool-Helper greatly reduces the human effort required to build CogTool models, it still requires human effort to create methods for each task through capture or writing XML. Its method inference algorithm discovers some alternative methods, but it cannot infer all methods because it depends on the methods the UI designer defined. For example, if the task was to make some selected text bold and centered, and the UI designer demonstrated only (1) making text bold *then* centered using the menus and (2) bold *then* centered using the toolbar, the inference algorithm could discover making the text bold using the menus then centered using the toolbar, but could never discover the alternative method of making the text centered *then* bold. The HPRT process proposed in this paper goes beyond CogTool-Helper in that it generates all humanly-reasonable methods.
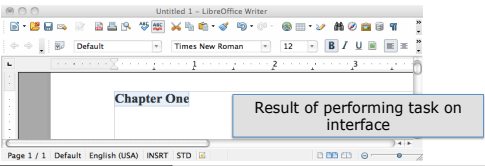
### IV. DETAILS OF HPRT: TEST GENERATION

In Section II, Figure 1, we presented an overview of our vision for HPRT. Stage 1 and Stage 4 represent inputs (tasks and versions of a UI) and outputs (histograms of performance predictions) of HPRT. The tools used in Stage 3 already exist and were just described in Section III as our prior work on CogTool and CogTool-Helper [12]. Thus, the key technical challenge that we face to realize HPRT is Stage 2, *Task Specific Test Case Generation*. Figure 1 shows that Stage 2 includes extracting task GUI events, defining task rules, and generating all test cases for the task, each of which will be detailed below.

To make our example task (from Section II) concrete we add some text; the task is to type in the text Chapter One, select the text, and make it bold. The resulting state of this task is shown in the top of Figure 2 as executed in LibreOffice [21] an office application. We have limited this task for illustration purposes, restricting our analysis so that the user types in the text *first* rather than starting the task by setting the font to bold, but, in practice, tasks would not be as restrictive. For this example, we assume $V_3$ of the UI that has menus, keyboard shortcuts, and toolbars, any of which, in different combinations, can be used to perform this task.

### A. Extract Task GUI Events

Since most EFGs for real applications are very large, (OpenOffice Writer 3.3.0 has 605 nodes and 79,107 edges

Fig. 2. Example Task on Word Processor

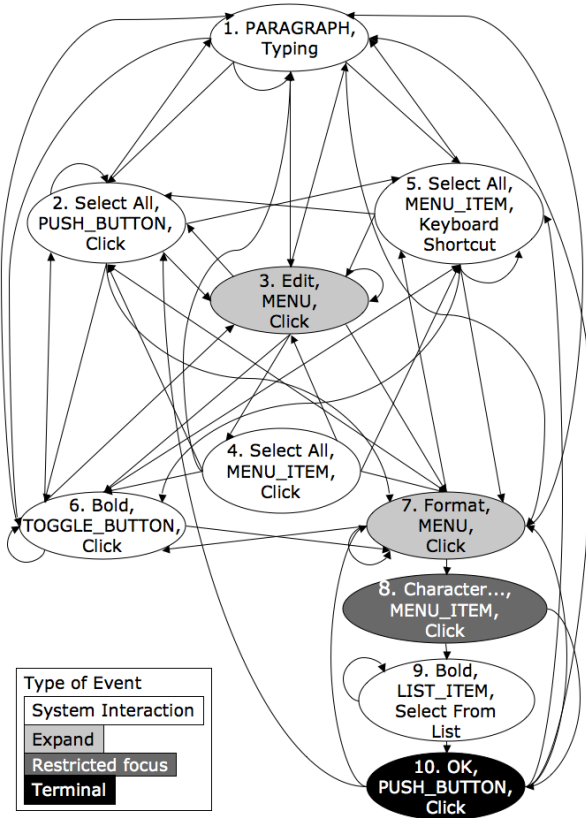| Sub-Goal | Approach | Partial Event Tuple: <Name, Type, Action> |
|---|---|---|
| Type Text: Chapter One | | 1. <..., PARAGRAPH, Typing> |
| Select All of the Text | A. Toolbar | 2. <Select All, PUSH_BUTTON, Click> |
| | B. Menu | 3. <Edit, MENU, Click><br>4. <Select All, MENU_ITEM, Click> |
| | C. Keyboard | 5. <Select All, MENU_ITEM, Keyboard Shortcut> |
| Make Text Bold | A. Toolbar | 6. <Bold, TOGGLE_BUTTON, Click> |
| | B. Menu | 7. <Format, MENU, Click ><br>8. <Character..., MENU_ITEM, Click><br>9. <Bold, LIST_ITEM Select_From_List><br>10. <Okay, PUSH_BUTTON, Click> |



Fig. 3. Resulting Event Flow Graph

[13]), we created a *filter* that works with the ripper to reduce the EFG to a sub-graph containing only events related to our task. The input to the filter is a list of event-tuples, one for each event related to our task, of the form: <Title (T), Class (C), Window (W), Action (A), Parent (P), Parameter (R)>. Title is the textual label of the widget in the code, i.e., what an end-user would see displayed. When the widget does not have a textual label, such as a button that only displays an icon, but has a tooltip, then the tooltip text is used as the Title. Class describes the type of widget, such as a PUSH_BUTTON, TOGGLE_BUTTON, etc. Window is the textual label of the window containing this event-tuple.

Action is more complex, defining which event handler will be used for this event. It's values include *Click* (currently we only support left click), *Typing* (typing with the keyboard), *Set Value*, *Select from List*, *Keyboard Shortcut* and *Keyboard Access*. We support several actions when Typing text, *insert*, *replace*, *select*, *unselect* and *cursor*, some of which have additional parameters, such as the text to insert or replace. *Keyboard Access* is used when the keystrokes walk through a hierarchical menu instead of directly accessing a command as a Keyboard Shortcut does (e.g., Alt-oh opens the Format menu and selects the Character item).

Parent is optional. It is the title of the container for this event-tuple, which provides a way of disambiguating information when more than one widget in a window matches the same Title text, or when a widget does not have any Title text. Parameter, also optional, is only used for widgets with the action Typing.

The event-tuples for our task are shown in the table at the bottom of Figure 2. We have reduced the event-tuple to show only <T, C, A> as they are enough to make each event unique in our example. The first column in the table is a task sub-goal. The second column lists the approaches that would lead to the events (e.g. using the menu or keyboard). The last column shows the event-tuples associated with each approach. In the top row, Event-Tuple 1, the main paragraph widget for the document has no title. This is a situation where we would use the optional Parent parameter, which would be, in this case, the text for the main document window, "Untitled 1 - LibreOffice Writer".

The EFG representing our example task is shown in Figure 3. It has 10 events (nodes) corresponding to the event-tuples in Figure 2, and 52 node relationships (edges). The shading in Figure 3 shows the *type* of event for each node, encoded by GUITAR as a property of the node [17]. A *System Interaction* event causes a functional change to the system (e.g., selecting all the text). The rest of the events cause structural changes. An *Expand* event opens a menu or a list to allow the user to select from a set of actions. A *Restricted Focus* event opens a modal window; the user must interact with that window until it is closed. Finally, a *Terminal* event closes a window. We will need to use these types in the next step.

### B. Define Task Rules

To restrict the generated test cases to those reasonable for human performance testing, we constrain the generator with two kinds of rules. The first is a *global rule*, enforced for all tasks. Global rules stem from typical user behavior, apply to most tasks, and are embedded in our test generation tool; the user does not write them. The second kind of rule is a *task specific* rule. These arise from the logic of the specific task and interface and therefore need to be written anew for each task and/or interface to be analyzed. Task specific rules can override global rules if the task and UI so require.

*1) Global Rules:* To generate humanly-reasonable test cases for our example task, we have defined four global rules. The first ensures that the task is logically complete and the

rest apply to the efficiency usability metric assumed to be the goal of the UI design evolution.

1) *End in Main Window*. The test case must end with a system interaction event in the main window, or with a terminal event resulting in only the main window being opened. This prevents ending in a dialog box with changes that haven't been applied, with an open menu, etc. No expand event, restricted focus event, or an event between a restricted focus event and a terminal event can be the last event in a test case, eliminating events #3, #7, #8, and #9 as potential last events in Figure 3.

2) *Expand Followed by Child Event*. An event that expands a menu or list, must be immediately followed by an event that is executed on a child of that menu or list. This prevents expanding a menu or list and then performing no actions on it. After event #3 (Edit, MENU, Click) is performed, the only valid event that can follow in this graph would be #4 (Select All, MENU_ITEM, Click). There is an edge on this graph leading from #3 to #7, but this rule prevents this edge from being in a test case in our example.

3) *Window Open and Close Can't Happen*. A window can't be opened and immediately closed without some other event happening in between. We can't have event #8 immediately followed by #10, despite an existing edge. This rule will force the test case generator to take the path from event #8 to #9.

4) *No Repeat Events*. No event can appear more than once in a test case unless it appears in a task specific rule that overrides this general rule. This rule prevents a test case from pressing a button over and over again. This may be a valuable test case for functional testing to make sure the system doesn't crash with repeated button-pressing, but it is not often reasonable for usability testing.

Rules 2 and 3 apply only to the efficiency usability metric; new users will often open a menu, a list, or a window, just to see what it reveals. Furthermore, if a specific task requires a skilled user to check the status of information in a menu, a list, or a window (e.g., if a menu item can be toggled on or off), then a task-specific rule would be written to override these general rules for those specific tasks. Rule 4 will need to be overridden when a task requires the same event to be executed multiple times. For example, if our task also italicized the text, then we would would need to allow #7 (Format MENU, click) to appear more than once since a task that performs both bold and italic using only menus needs to expand the same menu more than once (see the *Repeat* task-specific rule below).

*2) Task Specific Rules:* The global rules are primarily structural, enforcing constraints that are common to many tasks. However, individual tasks and UIs also have constraints that restrict them based on their functional properties. We have identified four types of constraints and created rules for each.

1) *Exclusion*. This is a mutual exclusion rule. It says that exactly one of the events in an exclusion set must be included in each test case. Examples of events that would be in an exclusion set for our task are #2, #4, and #5. They all achieve the same goal – selecting the text.

2) *Order*. This rule specifies a partial order on events. We group events into *Order Groups*, i.e., sets of events that are in the same ordering equivalence class, and then place the groups in the required order. Only system interaction events appear in order sets, since the other types of events only cause structural changes. In our task, we required typing the text to happen before all other events to make this example simple enough to explain. Thus we place #1 (PARAGRAPH, Typing) alone in the first order group. Since selecting the text must happen in our example before it can be made bold, we place events #2, #4 and #5 in the second order group and events #6, and #9 in the last order group. If the example text also centered the text, then we would include both center and bold within the same partial ordering group.

3) *Required*. Events in the required list must appear in all test cases. In our example the only event that is required is event #1 (PARAGRAPH, Typing).

4) *Repeat*. Events in the repeat list allows us to include specific events in a test case more than once, overriding the global rule against repeated events. As mentioned above, this rule would allow event #7 (Format MENU, click) to appear more than once if the task required several changes in format to some selected text.

### C. Generating All Test Cases for the Task

Once we have the EFG and the set of rules, we supply these as input to an existing test case generator and generate all possible tests for this EFG that are valid with respect to the rules. For our initial implementation we use the existing GUITAR test case generator [13] to enumerate all possible test cases of particular lengths (corresponding to the possible lengths of completing our task) and add a filter to discard test cases that do not pass the rules as they are generated. Although simple to implement in a feasibility study, this generate-then-filter approach is likely to be too inefficient for larger applications. Future work will include direct test generation algorithms that first reduce the EFG with respect to the rules, and/or utilize constraint solvers to check satisfiability.

## V. FEASIBILITY STUDY

We conducted a preliminary study to determine the feasibility of our approach.[2] We answer two research questions.

**RQ1:** Does our approach to test generation for HPRT provide potentially useful information for a UI designer?

**RQ2:** Will inferred methods allow us to sample test cases during HPRT without diminishing the value?

### A. Feasibility Study Scenario

We selected three modules of LibreOffice 3.4 [21], `swriter`, `simpress` and `scalc`, to illustrate the process a product team would go through and the resulting information

---

[2]Experimental artifacts and results can be found at: http://www.cse.unl.edu/~myra/artifacts/HPRT-2013/

157

it would gain from HPRT. The first step is to identify tasks that the end-user would do in the real world and create representative instances of those tasks. This information usually results from field studies, interviews, questionnaires, or log reports interpreted by user experience professionals. As an illustration, we created four tasks for our study, described in Table I.

Our study considers three hypothetical versions of LibreOffice that introduce different UI features to the end-users. The first version (*M*) presents only menus to access the functions needed for these tasks. The second (*MK*) adds the ability to access these functions with keyboard shortcuts. The third (*MKT*) adds toolbars for common functions (the default appearance of LibreOffice 3.4).

### B. Metrics

Quantitative predictions of skilled task performance time for each method on each version of the system, and the resulting distributions of those predictions, will speak to whether automatic test case generation would produce interesting results for UI usability evaluation (*RQ1*).

For *RQ2*, the metrics are the run time required to generate the test cases, the total number of methods resulting from these test cases in the final CogTool project, the number of inferred methods added by CogTool-Helper, and the human performance predictions for all of the methods.

### C. Study Method

To simulate the first two hypothetical versions of LibreOffice, we removed the toolbars using LibreOffice 3.4's customization facility. We encoded the necessary widgets and actions in GUITAR's format described in Section IV. The number of events listed for each task and version of the system are shown in Table I (No. Evt.).

We then wrote the task-specific rules (rightmost column, Table I, also found on our website). We ran each in CogTool-Helper and imported the resulting design and test cases into CogTool. Finally, we extracted the user times using the CogTool export function and used these to generate histograms, used to answer RQ1.

To investigate whether inferred methods can be used to reduce the run time cost of HPRT without diminishing the value of the resulting human performance metrics (RQ2), we use the LibreOffice version with menus, keyboard shortcuts

and toolbars, since it has the largest number of test cases. We randomly select (without replacement), the required number of test cases for 10%, 25% and 50% of the complete set. We sample five times at each percentage for each task, to prevent bias from a single outlier run. We then run CogTool-Helper on the samples of test cases and capture the total number of methods in the final CogTool project, the number of those that were inferred by CogTool-Helper, the run time required to create the designs, and the human performance predictions for all of the methods. We report averages of these values.

## VI. RESULTS AND DISCUSSION

### A. RQ1: Usefulness of HPRT

Table II shows the three versions of each task: menu only (*M*), menu + keyboard (*MK*) and menu + keyboard + toolbar (*MKT*). For each version we show the number of test cases generated, the mean time predicted for a skilled user to accomplish this task, the minimum predicted time, the maximum predicted time, and the standard deviation. From the raw predictions, we show histograms of the number of test cases by time predictions for each task in each version of the system (Figure 4).

Looking first at Table II, in all but one case (Absolute Value), the mean time decreases with the addition of keyboard shortcuts and in all cases it decreases again with the addition of the toolbar. Since CogTool predictions have been shown to be within +-10% of empirically observed task execution time, Table II shows no detectable efficiency advantage for toolbars. Perhaps more interesting, is the decrease in minimum time to accomplish each task, which decreased by 40% for the *Format* task. This suggests that the most proficient skilled users could be substantially more efficient on this task, information that might be used for marketing or sales were more tasks to show this advantage. In addition, test cases that displayed this efficiency could feed directly into training videos or "tips of the day" to help users attain such proficiency.

The maximum time tells a different story; in three of the four tasks adding keyboard shortcuts *increases* the maximum predicted time. Although the increase is within the predictive power of CogTool, this result illustrates the concept of detecting a regression with respect to efficiency, Examining the models' behavior for the max-time test cases, the increase

TABLE I
TASKS USED IN THE FEASIBILITY STUDY

| LibOff Module | Task Name | Task Description | Ver. | No. Evt. | No. Rul. |
|---|---|---|---|---|---|
| Writer | Format Text | Text Chapter typed in, selected, made bold and centered | M | 9 | 4 |
| | | | MK | 12 | 5 |
| | | | MKT | 13 | 7 |
| Writer | Insert Hyperlink | Insert Hyperlink to Amazin and make text uppercase | M | 9 | 3 |
| | | | MK | 11 | 5 |
| | | | MKT | 13 | 8 |
| Calc | Absolute Value | Insert abs val function, shift cells right, turn off headers | M | 11 | 4 |
| | | | MK | 14 | 6 |
| | | | MKT | 16 | 10 |
| Impress | Insert Table | Insert a table, add new slide, hide task pane | M | 7 | 3 |
| | | | MK | 9 | 5 |
| | | | MKT | 11 | 7 |

TABLE II
PERFORMANCE PREDICTIONS: SKILLED TASK EXECUTION TIME (SEC)

| Task (Version) | No. TC | Mean Time | Min Time | Max Time | SD |
|---|---|---|---|---|---|
| Format Text (M) | 3 | 13.8 | 13.7 | 13.8 | 0.1 |
| Format Text (MK) | 24 | 13.2 | 12.3 | 14.1 | 0.6 |
| Format Text (MKT) | 81 | 11.8 | 8.6 | 14.1 | 1.7 |
| Insert Hyperlink (M) | 2 | 20.5 | 19.5 | 21.6 | 1.5 |
| Insert Hyperlink (MK) | 8 | 20.1 | 18.3 | 21.6 | 1.4 |
| Insert Hyperlink (MKT) | 18 | 19.8 | 17.6 | 21.6 | 1.3 |
| Absolute Value (M) | 4 | 18.1 | 17.9 | 18.3 | 0.1 |
| Absolute Value (MK) | 32 | 18.3 | 17.7 | 18.8 | 0.2 |
| Absolute Value (MKT) | 72 | 17.1 | 14.1 | 18.9 | 1.6 |
| Insert Table (M) | 3 | 12.8 | 12.7 | 12.9 | 0.1 |
| Insert Table (MK) | 12 | 12.7 | 12.3 | 13.3 | 0.3 |
| Insert Table (MKT) | 36 | 12.3 | 11.3 | 13.3 | 0.4 |
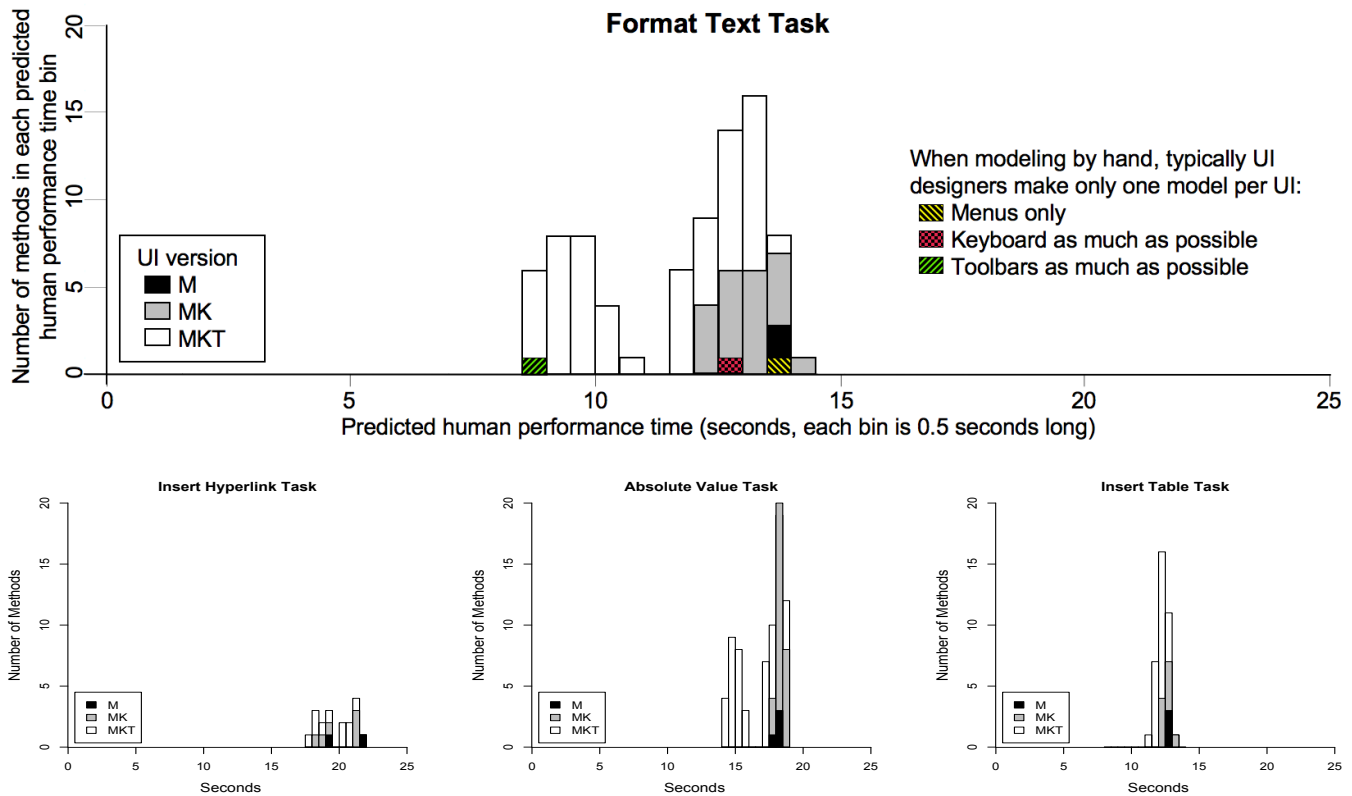
Fig. 4. Histograms of Predictions of Skilled Task Execution Times produced using HPRT. Format text task graph is annotated to show predictions for each interface version as revealed by the single model that is typically done by a UI Designer

arises because these test cases mix menus and keyboard shortcuts, requiring the user to move the hand between the mouse and the keyboard. This information might encourage project teams to increase coverage of keyboard shortcuts so that skilled users can keep their hands on the keyboard. Likewise, in the *Absolute Value* task, the maximum time increases when toolbars are added (another regression). In this case, the presence of the toolbar forced a dialog box to be moved, requiring the user to move the mouse further to interact with it. Moving the dialog box is a design decision that could be reversed after HPRT.

Turning to the performance histograms (a contribution in itself), we see information never attained before with predictive human performance modeling of UI efficiency. In the past, a UI designer would typically model only one method using the menus, one using keyboard shortcuts as much as possible, and one using toolbars as much as possible (e.g., [11]) because even this simple modeling was too much effort. The graph for the *Format Text* task is annotated to show the times that a single model for each version of the UI would have produced: 13.7s, 12.6s and 8.6s, respectively. This by-hand analysis would have given the UI designer confidence that the progressive addition of features would improve efficiency, but would not have revealed the poor performance cluster of methods for the version with the toolbar. This bimodal distribution suggests that end-users may not profit from toolbars as much as the designer hoped and may have implications for training,

as it would be desirable to guide users to the lower cluster of methods. This same pattern occurs for the *Absolute Value* (bottom center) task, but not the *Insert Hyperlink* and *Insert Table* tasks, where the progressive addition of features changes the range and mean of the distribution but not the basic shape. Predicting the distribution of times associated with methods using different UI features plus methods using a mixture of those features, is new to human performance modeling and opens up possibilities for the design of UI and training not yet imagined in the field.

Because HPRT provides the mean human performance time that UI designers have already found useful, we can answer *RQ1* in the affirmative; HPRT has potential to provide UI designers with useful information for design and development. Because HPRT expands the available information to include distributions of possible method times, its potential is even greater than current by-hand modeling approaches.

### B. RQ2: Impact of Inferred Methods

We now examine the impact of inferred methods on the design construction phase of CogTool-Helper, to evaluate whether we can sample test cases rather than generate and run every test case for every task. We believe this will help in scalability of HPRT on large tasks.

Table III shows data for two tasks on the last version of the UI (MKT) sampled at 10%, 25% and 50%, with the number of test cases included at that sample level presented after the

| Design Construction | | | CogTool Analysis | | | | |
|---|---|---|---|---|---|---|---|
| Task (Sample %/size) | Run Time(m) | % Reduced | No. Methods | No. Inferred | Mean Time(s) | Min Time(s) | Max Time(s) |
| Format Text (10%/8) | 13.3 | 88.9 | 41.4 | 33.4 | 11.9 | 8.8 | 14.0 |
| Format Text (25%/20) | 31.2 | 74.0. | 76.2 | 56.2 | 11.8 | 8.6 | 14.1 |
| Format Text (50%/41) | 61.0 | 49.1 | 81.0 | 40.0 | 11.8 | 8.6 | 14.1 |
| Format Text (All) | 120.0 | – | 81.0 | – | 11.8 | 8.6 | 14.1 |
| Absolute Value (10%/7) | 18.8 | 89.6 | 25.6 | 18.6 | 16.9 | 14.1 | 18.7 |
| Absolute Value (25%/18) | 45.8 | 74.7 | 56.4 | 38.4 | 17.0 | 14.1 | 18.9 |
| Absolute Value (50%/36) | 90.3 | 50.1 | 69.6 | 33.6 | 17.1 | 14.1 | 18.9 |
| Absolute Value Task (All) | 180.7 | – | 72.0 | – | 17.1 | 14.1 | 18.9 |

slash (the other tasks are similar and can be found on our website). We show the time in minutes taken, averaged over five samples, for CogTool-Helper to run the test cases and create the designs, followed by the average percent reduction over running all of the test cases (% Reduced). We list the average number of methods in the resulting CogTool project, along with the average number of inferred methods (methods that were created by our inference algorithm, discussed in Section III). The last three columns show the times in seconds of the human performance predictions (mean, minimum and maximum). The last row of each task contains data for the full set of test cases.

For the 10% sample, we see an 88.9% and 89.6% reduction in runtime, but we also see a loss in the range of predicted human performance times. In the Absolute Value task, for instance, we have predictions for only one third of the possible methods. However, the mean, maximum and minimum prediction times are not far from the full data set.

In both tasks, the 50% samples show the full range of human performance values, and we either generate all of the methods for that task with the inferred method algorithm (*Format Text*), or come within 10% of all possible methods (*Absolute Value*). The runtime savings are about 50%, which equates to about one and a half hours in the Absolute Value task. Thus, we can also answer *RQ2* in the affirmative; CogTool-Helper's inferred method algorithm enables effective sampling of text cases, which may allow HPRT to scale to larger tasks than illustrated.

### C. Further Discussion and Future Work

All models are approximations of reality, and, as mentioned, CogTool reports +-10% of the average human performance a UI designer would observe were he or she able to collect skilled time empirically. Part of the variability in human behavior that CogTool did not capture in previous studies is just what we are exploring here, i.e., the variation in ways to accomplish a task that skilled users exhibit. Another factor is normal variation in all human performance between and within individuals (e.g., slower performance when fatigued, faster after having drunk a cup of coffee, etc.). HCI research is just beginning to explore modeling tools that predict the latter (e.g., [22]), but our implementation of HPRT is the first we know of to make it easy to predict the former. Until validation research progresses in HCI, it is premature to proclaim that

the results such as those in Table II and the histograms should be trusted to make important UI design decisions. That said, it is important to explore the types of information HPRT *could* provide, and the types of interpretation a project team *might* make, as the science of predicting variability matures.

It should be noted that the results we presented here arise from using an equal weighting of all test cases to determine values in Table II and draw the histograms. In the absence of real-world information about our fictitious versions of Libre-Office and tasks, we used the assumption of equal weighting to demonstrate some of the practical implications of HPRT. However equal weighting is not necessarily a realistic assumption. Card, Moran and Newell [7] observed that people select their methods based on personal preferences (e.g., some prefer menus, others prefer keyboard shortcuts) or characteristics of the task (e.g., if the user's hand is already on the mouse, it is more likely the user will use a toolbar button than a keyboard shortcut). If the analysis is of a system already released, the UI designer may have log data to refine the weighting assumption, or prior experience with the user community (similar to Card, Moran and Newell's observations) may influence the weights. If no information about method-weighting is available, the UI designer could play "what if" to explore the impact of different assumptions about method-weighting on the mean times and distribution profiles. The values and histograms will change with different method-weighting assumptions but the information they provide can be used to reveal properties of the UI design as illustrated above.

It is important to remember that although our illustrative example assessed the performance time of skilled users, efficiency is not the only dimension of usability amenable to HPRT. As mentioned in Section III, recent modeling work with *CogTool-Explorer* has predicted the ability of novice users to discover how to do tasks on a new interface [10]. Future work could provide HPRT for multiple usability metrics. Then, a feature like toolbars, which did not show an advantage for efficiency in some tasks may be shown to be more discoverable for novice users than menus or keyboard shortcuts.

We have presented a first investigation into the ability of HPRT to provide useful information for UI design and to scale to real-world use. Although many more applications need to be tested before HPRT can enter into mainstream regression testing, LibreOffice can be considered representative of a

family of office software and does not have any unusual GUI features that would make it particularly amenable to our study. With respect to the rules that limit the extraction of a task-relevant EFG, we believe that we have identified a good starting set of rules, but additional rules may emerge as future work tackles more complex tasks. For instance, our current task specific rule for *repeat* is binary, but we believe that this may need to allow specific cardinality ranges. For this study we use the existing GUITAR test case generator which traverses all paths of the given length on the EFG and apply rules on each resulting test case. However, if we apply the rules first (or as we explore the graph), then we can avoid this bottleneck. We plan to explore both of these modifications.

Finally, this first illustration of HPRT required substantial skill and effort to identify the widgets and event-tuples necessary to accomplish tasks and to write the global and task-specific rules. We acknowledge that this level of skill and effort is onerous for a real-world project team. Future work must include human-centered design to make tools usable by the right person or people in a software project team.

## VII. Conclusions

We have presented the first attempt that we know of for human performance regression testing, HPRT. We achieved this by extending CogTool-Helper to include a test generation method that leverages existing functional GUI testing tools. By identifying the events of interest and a set of rules, test cases can be generated to mimic humanly-reasonable user tasks. Our feasibility study has shown that ranges of human performance predictions provide rich data that was not previously possible, and that this will allow project teams to explore the impact of their changes more thoroughly. We also examined the impact of sampling and found that we retain almost all of the information, and save as much as hours of runtime using only half of the test cases. This shows the potential of the inferred methods algorithm that is built into CogTool-Helper, which had not been evaluated on this scale before. We believe this feasibility study opens the door to a new method for rapidly assessing the usability of UIs with potential benefit for both SE and HCI research and practice.

### References

[1] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[2] B. Beizer, *Software Testing Techniques*. International Thomson Computer Press, 1990.

[3] C. Yilmaz, A. S. Krishna, A. Memon, A. Porter, D. C. Schmidt, A. Gokhale, and B. Natarajan, "Main effects screening: a distributed continuous quality assurance process for monitoring performance degradation in evolving software systems," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05, 2005, pp. 293–302.

[4] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora, "A framework for measurement based performance modeling," in *Proceedings of the 7th international workshop on Software and performance*, ser. WOSP '08, 2008, pp. 55–66.

[5] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective & scalable prioritisation incorporating expert knowledge," in *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA*, July 2009, pp. 201–211.

[6] M. H. Blackmon, M. Kitajima, and P. G. Polson, "Tool for accurately predicting website navigation problems, non-problems, problem severity, and effectiveness of repairs," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*, 2005, pp. 31–40.

[7] S. K. Card, A. Newell, and T. P. Moran, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1983.

[8] E. H. Chi, A. Rosien, G. Supattanasiri, A. Williams, C. Royer, C. Chow, E. Robles, B. Dalal, J. Chen, and S. Cousins, "The bloodhound project: automating discovery of web usability issues using the InfoScent simulator," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '03)*, 2003, pp. 505–512.

[9] B. E. John and D. E. Kieras, "Using GOMS for user interface design and evaluation: which technique?" *ACM Transactions on Computer-Human Interaction*, vol. 3, no. 4, pp. 287–319, Dec. 1996.

[10] L. Teo, B. E. John, and M. H. Blackmon, "Cogtool-Explorer: A model of goal-directed user exploration that considers information layout," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12, May 2012, pp. 2479–2488.

[11] R. Bellamy, B. John, and S. Kogan, "Deploying CogTool: integrating quantitative usability assessment into real-world software development," in *Proceedings of the International Conference on Software Engineering*, 2011, pp. 691–700.

[12] A. Swearngin, M. B. Cohen, B. E. John, and R. K. E. Bellamy, "Easing the generation of predictive human performance models from legacy systems," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '12, May 2012, pp. 2489–2498.

[13] "GUITAR – a GUI Testing frAmewoRk," http://guitar.sourceforge.net, 2011.

[14] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 144–155, Feb. 2001.

[15] B. E. John, K. Prevas, D. D. Salvucci, and K. Koedinger, "Predictive human performance modeling made easy," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '04, 2004, pp. 455–462.

[16] A. M. Memon and M. L. Soffa, "Regression testing of GUIs," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE-11, 2003, pp. 118–127.

[17] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering*, ser. WCRE '03, 2003, pp. 260–269.

[18] F. Belli, "Finite-state testing and analysis of graphical user interfaces," in *International Symposium on Software Reliability Engineering (ISSRE)*, 2001, pp. 34–43.

[19] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," in *Proceedings of the European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-9, 2001, pp. 256–267.

[20] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10, 2010, pp. 1535–1544.

[21] "LibreOffice 3.4," http://libreoffice.org, 2011.

[22] E. W. Patton and W. D. Gray, "SANLab-CM: A tool for incorporating stochastic operations into activity network modeling," *Behavior Research Methods*, vol. 42, no. 3, pp. 877–883, 2010.