Easing the Generation of Predictive Human Performance Models from Legacy Systems

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

Bonnie E. John
Rachel K. E. Bellamy
IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532
{bejohn,rachel}@us.ibm.com

ABSTRACT
With the rise of tools for predictive human performance modeling in HCI comes a need to model legacy applications. Models of legacy systems are used to compare products to competitors, or new proposed design ideas to the existing version of an application. We present CogTool-Helper, an exemplar of a tool that results from joining this HCI need to research in automatic GUI testing from the Software Engineering testing community. CogTool-Helper uses automatic UI-model extraction and test case generation to automatically create CogTool storyboards and models and infer methods to accomplish tasks beyond what the UI designer has specified. A design walkthrough with experienced CogTool users reveal that CogTool-Helper resonates with a "pain point" of real-world modeling and provide suggestions for future work.

Author Keywords
Predictive human performance modeling; automatic GUI testing.

ACM Classification Keywords
H5.2. [Information Interfaces and Presentation]: User Interfaces; D.2.5 [Software Engineering]: Testing and Debugging

General Terms
Human factors.

INTRODUCTION
When a UI designer sets out to design a new product or the next version of a product, the process often starts with an analysis of existing systems, i.e., competitors' products and/or the current version. Customer complaints and help-desk logs often provide clues about deficiencies in the current version and unfavorable comparisons to competitors, but may not provide a direct connection between the complaints and the UI design, so additional data collection or analysis must be done to figure out exactly how to respond. For example, Bellamy et al. [2] reports that in one design situation, employees trying out a new internal portal commented that it was "considerably slower" than the old system, and that in another situation the "customers requested that we show them that our tool … was as efficient as another product." Predictive human performance modeling could be used to diagnose such issues (and indeed, was used in the work of Bellamy et al. [2]), but that means that modeling is being done on both existing legacy systems, and on proposed design ideas.

Application of human-performance modeling in HCI was originally conceived as an aid to design. "Design is where the action is in the human-computer interface. It is during design that there are enough degrees of freedom to make a difference. An applied psychology brought to bear at some other point is destined to be half crippled in its impact." [3, p.11] But in practice, we see modeling done at least as often on legacy systems as on proposed designs, i.e., as an analysis of existing problems to inspire design or to serve as a benchmark against which a new design is compared (e.g., five examples in Bellamy et al. [2], others in the work of Gray et al., Knight et al. and Monkiewicz [5,9,15]).

When modeling human performance on a legacy system, that system is usually redescribed in a representation dictated by the human modeling framework, that is, it is re-implemented in some text-based description language, as a storyboard, or in a different computational language. At worst, this means reimplementing an entire existing system; at best it is a time-consuming process of capturing screens and drawing hotspots on top of widgets to make a storyboard [7]. Either way, it is a burden to analysts because it is perceived as extra work — the software already exists, why can't it be used? One approach to solve this problem is VisMap [17]. VisMap uses image processing to "see" the screen of an existing UI and pass that information to a human performance model, and simulate motor movements (clicks, key presses, etc.) by manipulating the event queue at the operating system level. However the VisMap approach does not address another issue, the need to describe the tasks to be modeled.

CHI'12, May 5–10, 2012, Austin, Texas, USA.

When modeling, especially with GOMS or Keystroke-Level Model (KLM) [3], the models must be given the knowledge of how to do tasks (either through programming, as in GLEAN (GOMS Language Evaluation and Analysis, [8]), or by demonstration as in CogTool [7]). Therefore, the analyst only obtains predictions of performance for those methods of doing a task for which s/he has explicitly encoded the task knowledge. But in complex systems, there may be many ways to accomplish a task (e.g., using menus, using toolbars, using keyboard shortcuts, and any combination of these methods), and analyzing all of them using current 'by hand' modeling methods may be intractable.

The limitations of current predictive modeling tools that we have presented here have an analogy in an orthogonal and unconnected domain of research; that of software testing of graphical user interfaces (GUIs) [13,14,19,20]. Traditionally, system testers examined their applications that were to be tested, and then created manual use cases to exercise what they believed to be the important behavior [10,19]. But this approach has been shown to miss faults in the application and to be time consuming to implement. In recent years, there has been a drive towards developing techniques and tools for automating both UI-model extraction [13] and test case generation [20]. The GUI widgets and/or buttons are represented as *events* either in the form of a finite state machine or a graph, and this abstraction is then used to perform test case generation by traversing states or nodes in the graph. Once the test cases have been generated, other tools automatically replay these on the actual application. This process automation allows a larger set of test cases to be generated and run, and a broader range of behaviors to be tested than was possible using the manual approach. Research has also shown that automated GUI test case generation can improve fault detection [20].

In this paper, we bridge the gap between these two domains of research and leverage the advances in GUI automated testing to facilitate cognitive predictive modeling. As an exemplar, we present *CogTool-Helper* (Figure 1), an automated design and task generator for CogTool. CogTool-Helper can (1) automatically create a CogTool design storyboard from an existing application; (2) represent methods to accomplish tasks on the design (either through demonstration, by defining a task in an XML format used by software testers, or eventually automatic test case generation); and (3) uncover implicit methods that exist on the design that perform the same tasks in alternative ways. We believe that using such GUI testing tools as input to predictive modeling tools will improve both efficiency and effectiveness of the current predictive modeling process.

The rest of the paper is laid out as follows. In the next section we explain the contributing technologies. We follow this with an overview of our tool and then describe each aspect of CogTool-Helper in detail. We present a design walkthrough of CogTool-Helper, then discuss the potential benefits of this approach, and conclude with a roadmap for future work.

CONTRIBUTING TECHNOLOGIES

CogTool

CogTool [7] is a tool that enables UI designers to create valid KLMs by describing their design in a storyboard and demonstrating tasks on that design. In CogTool's

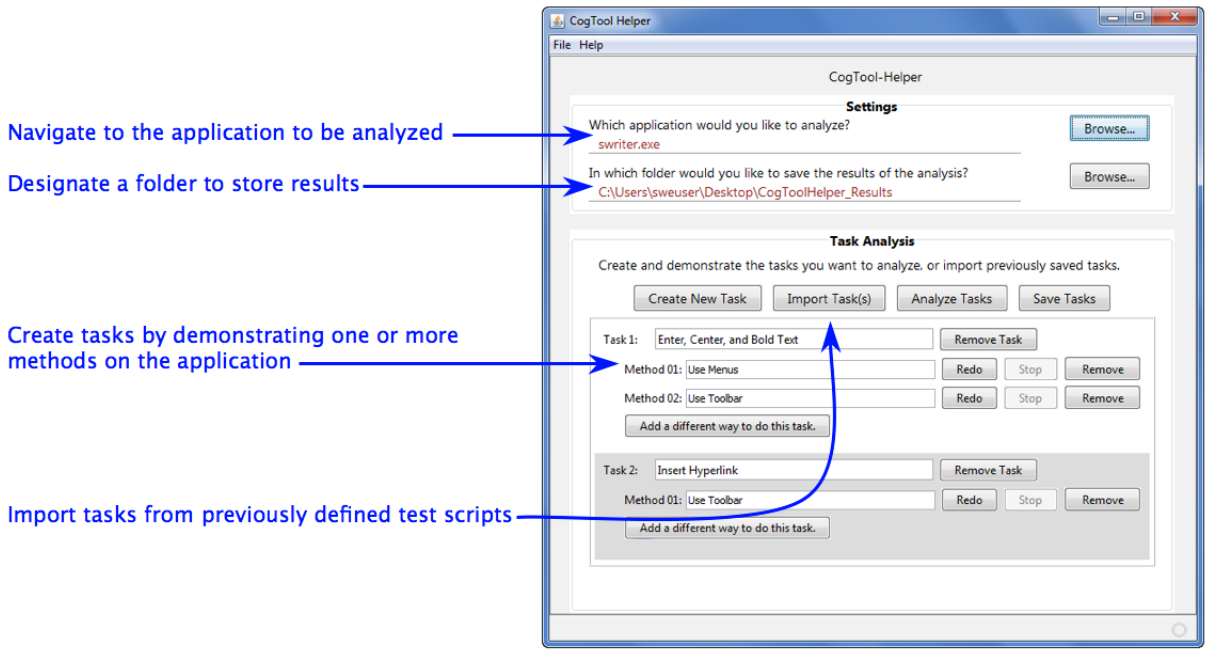

Figure 1. CogTool-Helper

storyboard, each design can contain one or more devices (e.g., keyboard, touchscreen, microphone), each state of the UI is represented as a frame, each actionable interface item is represented as a widget with position, size, label, and type (e.g., link, button) and each action on a widget or device (e.g., mouse click, keys typed on the keyboard) is represented as a transition between frames.

To build a KLM, the designer creates a storyboard with the widgets required to do the task, walks through the task on that storyboard by selecting a start frame and performing the appropriate actions on that frame. CogTool records these actions, automatically inserts additional KLM operators based on prior research (e.g., eye-movements, thinking time), and follows the pre-defined transition to the next frame in the storyboard. The designer continues to walk through the steps in the task until it is complete and the entire KLM is built. When the analyst hits the "Compute" button, CogTool runs a computational model (implemented in ACT-R [1]) of what a user would see, think and do, producing a quantitative estimate of skilled execution time and a timeline visualization of what the underlying cognitive model was doing at each moment to produce that estimate. A designer can create multiple different UI storyboards, walk through the same task on each storyboard, and then compare predictions of skilled execution time for each. The designer can also analyze many different tasks and alternative methods for doing these tasks to explore the efficiency of different UIs for different tasks and methods.

Recent research has added CogTool-Explorer [18] to CogTool's capabilities, so it can now also predict novice exploration behavior using an underlying model of information foraging [16]. CogTool-Explorer uses the same form of a storyboard described above. While KLMs require only the widgets used along a correct path in the task (because skilled users know where these widgets reside and are not confused by widgets irrelevant to the task at hand), CogTool-Explorer requires all widgets to be represented because it models the time it takes a novice to visually search through the widgets and the confusion imposed when widgets irrelevant to the task have labels that seem similar to the task goal. Complicated UIs, therefore, require substantial effort on the UI designer's part to represent each and every menu item, button, pull-down list, etc. to make a storyboard complete enough to predict novice exploration.

GUI Testing Tools

There have been many approaches to automated GUI testing. One class of tools uses capture/replay, that provides the user with the ability to demonstrate a test case (capture), during which a script is recorded that can then be re-run (replayed) on the given application. While replay is fully automated, capture is a manual process — it requires that the tester decides what to test and then demonstrates those tasks on the application. Although this simplifies the testing process to a great degree, the number of tests that can be created and run, as well as the range of behaviors that can be tested is limited by the need for human effort.

In the work of Memon et al., a set of techniques and tools has been developed to automate the entire testing process, including the generation of tests [11,13,14,20]. We utilize their techniques and tools for CogTool-Helper. In their testing approach, the buttons, widgets and menus on the user interface are modeled as events in the form of a graph. Nodes are events, and edges are relationships between events indicating which events may follow other events.

In a single run of the application, called "ripping" [13], a depth first traversal of the interface is executed, opening all menus, windows and reachable widgets and buttons. The output of ripping is a directed graph called an event flow graph (EFG) that describes both the event relationships and types. Suppose an *edit* menu has a choice called "cut". The event, *cut*, will have an edge from the event, *edit*, indicating that *cut* can follow *edit,* and a test case may be generated that executes first *edit* followed by *cut.* The EFG for an application can be traversed to automatically generate test cases of a specified length and satisfying specific event coverage criteria [20].

Test cases are stored as XML and can then be automatically replayed on the GUI (and or manually edited by the tester and then replayed). The GUI Testing Framework, GUITAR, [11], consists of a set of open source tools, available for a variety of UI platforms such as Java, OpenOffice, Web Applications, Android and the iPhone. It includes a ripper, a replayer and test case generator.

Other techniques for generating GUI tests include representing them as finite state machines [11] or by using visual elements such as buttons and widgets to describe a test case [4]. Although we have selected a single technique in which to base CogTool-Helper, we believe that other methods for automated testing of GUIs may also benefit predictive human performance modeling and will examine some of these as future work.

COGTOOL-HELPER

Overview of CogTool-Helper

CogTool-Helper is a standalone Java application (Figure 1) built on top of the existing test case replayer, GUITAR [11]. For our prototype, we have incorporated two versions of GUITAR, one that works on OpenOffice applications and one that works on Java Swing applications. The UI designer need not have access to source code to analyze an application because GUITAR uses Java Accessibility API [6] to interact with the widgets on the UI.

CogTool-Helper consists of two main phases, *Task Construction* and *Design Construction* as illustrated in Figure 2 and presented in detail below. The output of this process is an XML file representing a complete CogTool model that can then be imported into CogTool for design and task analysis.

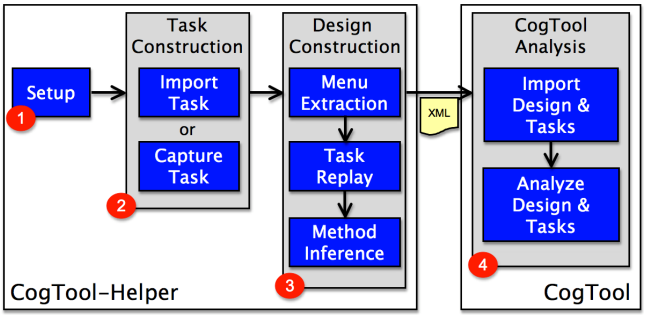

Figure 2. CogTool-Helper's process & relationship to CogTool.

Operation and Implementation details

(1) Setup

When CogTool-Helper is launched, the designer selects a legacy application to be analyzed. In Figure 1, the OpenOffice text editor (swriter.exe) has been selected. The designer must also choose a location to store the output of CogTool-Helper (c:\...\CogToolHelper_Results in Figure 1). CogTool-Helper then creates a connection to the application and the designer is ready to specify the tasks.

(2) Task Construction

In the *Task Construction* phase, the UI designer defines one or more tasks and creates one or more methods to achieve each task. As shown in Figure 2, there are two different ways to achieve this goal, *Import Task*, where the designer loads in previously defined tasks and methods, or *Capture Task*, where the designer demonstrates methods on the application being analyzed and CogTool-Helper captures their actions.

Each method is stored in CogTool Helper as a *GUI test case* in the GUITAR format. This allows for scripting and automatic generation of methods, and makes for easy integration with the test case replayer.

As an example, consider using OpenOffice Writer to enter text, center it on the page, and make it bold (*Enter, Center, and Bold Text*). A UI designer has entered two methods for doing this task in CogTool-Helper (Figure 1), one method using all menu commands and the other using all toolbar buttons. These demonstrations create two test cases in the GUITAR representation. A test case is a series of *Steps* containing the *Window* that is being accessed and the button or widget (*Component*) on which some *Action* such as a click is performed. For example, clicking the Select All toolbar button is represented as follows.

```
<Step>
  <Window>
    Untitled 2 - OpenOffice.org Writer_49
  </Window>
  <Component>Select All_44</Component>
  <Action>
    edu.umd.cs.guitar.event.OOActionHandler
  </Action>
  <WindowFlag>windowFlag</WindowFlag>
</Step>
```

Different methods to achieve the same effect result in different GUITAR test cases. For instance, when using the menus instead of the toolbars, selecting all text takes two steps, an action handler event to click on the Edit menu and then one to click on the Select All menu item.

```
<Step>
  <Window>
    Untitled 2 - OpenOffice.org Writer_49
  </Window>
  <Component>Edit_34</Component>
  <Action>
    edu.umd.cs.guitar.event.OOActionHandler
  </Action>
  <WindowFlag>windowFlag</WindowFlag>
</Step>
<Step>
  <Window>
    Untitled 2 - OpenOffice.org Writer_49
  </Window>
  <Component>Select All_36</Component>
  <Action>
    edu.umd.cs.guitar.event.OOActionHandler
  </Action>
  <WindowFlag>windowFlag</WindowFlag>
</Step>
```

Tasks can be scripted by hand or through an automated test case generator, and then loaded into CogTool-Helper with the "Import Task" button in Figure 1, but CogTool-Helper also allows designers to demonstrate methods using the *capture task* feature.

To capture a new method for a task, the designer provides names for the task and method in the appropriate text fields (Figure 1). Next, the designer clicks "Start"[1], and CogTool-Helper launches the legacy application. The designer then demonstrates the task on the legacy application, while CogTool-Helper captures the user's actions and converts them to the test case format. The designer clicks "Stop" when s/he has completed the task; recording stops and the method is saved.

In Figure 1, we see CogTool-Helper with two tasks defined. The first task, *Enter, Center and Bold Text*, has two methods: "Use Menus", and "Use ToolBar". The second task, *Insert Hyperlink* has only one method, "Use Toolbar". The first task has been imported from a file, but is indistinguishable from the other task that was captured by the designer. Once the designer has created and saved all of the tasks to be analyzed by CogTool, s/he will click the "Start Analysis" button, which begins the second phase of CogTool-Helper, *Design Construction*.

(3) Design Construction

The goal of the design construction phase is to generate all the information needed by CogTool to model a UI design and tasks performed on it and represent this information in

[1] Figure 1 shows "Redo" instead of "Start" because the button's label changes after a method has been demonstrated.

XML so it can be imported into CogTool. There are three key components that contribute to this phase of CogTool-Helper: *Menu Extraction*, *Task Replay* and *Method Inference*.

Menu Extraction. CogTool-Helper captures simple widgets, (e.g., buttons) as they appear during replay, but for menus and pull-down lists it extracts them during the menu extraction process (analogous to GUI ripping). It systematically opens all menus, pull-down lists, dialog boxes, and any other elements that are not initially visible within the root window of the application, via a depth first traversal. CogTool-Helper's menu extraction records the size, position, label and type of these widgets in CogTool XML format. This set of widgets will be used in the next process, task replay, so appropriate widgets can be added to the frames in the CogTool design.

Menu extraction can take a considerable amount of time for complex interfaces, even with it opening hierarchical menus as fast as possible, so it is shown to the designer as feedback that CogTool-Helper is working. The experience is much like watching a player piano produce music.

Task Replay. During task replay, a CogTool design that supports all tasks specified by the designer during task construction is created incrementally (Figure 3). Each method of each task is automatically performed on the interface using the GUITAR replayer, which treats each method as a test case and performs each step in turn on the UI. We have modified GUITAR to capture the UI information that CogTool needs and translate the test case into a CogTool design. As with menu extraction, the task replay process takes time and it is shown to the designer. But instead of opening all menus, dialog boxes, etc, task replay performs just those actions recorded in this method for accomplishing this task.

Before each method is performed, CogTool-Helper re-launches the application. If this is the first method being analyzed, CogTool-Helper starts with *Build Frame*, which constructs the XML for the initial CogTool frame. Build Frame begins by capturing a full screen image of the current desktop. This image data is placed in the background slot of CogTool's XML for this frame. Because all the background images are full screen, and the background image sets the size of a frame in CogTool, all

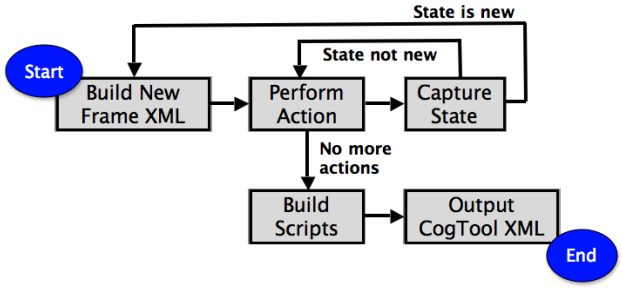

Figure 3. Task Replay Process

frames are the same size and scale; keeping all frames at the same scale is important for CogTool's human performance model to make accurate predictions of the duration of mouse movements.

CogTool-Helper then determines which windows are visible at this point in the task from the GUITAR test case. If a window is modal, CogTool-Helper will include only the widgets in the active modal window on this frame because a user, and therefore the CogTool model, can only interact with the modal window at this point in the task. If the window is modeless, CogTool-Helper includes widgets from all open windows because a user (and the CogTool model) could interact with any one of them.

To get the widgets in a frame, CogTool-Helper traverses the accessibility trees provided by the application for each window, collecting every object corresponding to a CogTool widget (e.g., buttons, links, text boxes). For each widget, CogTool-Helper extracts the position of its upper left corner, its height and width, its label and type, and creates a new widget in CogTool XML format. CogTool-Helper maps each type of accessibility object to the corresponding CogTool widget. For some CogTool widget types, we also capture additional information, such as the toggle state for a toggle button. A frame created by CogTool-Helper for OpenOffice Writer, with its first widget (the Select All toolbar button), is encoded as follows.

```
<design name='OpenOffice.org Writer'>
  <device>mouse</device>
  <device>keyboard</device>
  <frame name='Frame 002'>
    <backgroundImageData>…</backgroundImageData>
    <topLeftOrigin y='11' x='11'/>
    <widget   w-is-selected='false'
              w-is-standard='true'
              name='SelectAll_button_1'
              shape='rectangle'
              w-is-toggleable='false'
              type='button'>
      <displayLabel>Select All</displayLabel>
              <extent height='27' y='82'
                      width='25' x='1020'>
              </extent>
    </widget>
```

After the frame has been built, CogTool-Helper proceeds to the next phase of Task Replay, *Perform Action*. Perform Action looks at the action at this point in the test case and begins to create the CogTool XML representation for a transition. This step in the test case says which widget is the source of the transition and what type of transition it is (e.g, mouse click, keystrokes on the keyboard), but not what state it transitions to. Perform Action performs the action, creating a new *current state*, which is passed to the Capture State process.

The *state* of the application consists of all information that can be obtained through all widgets on the interface. It consists of attributes such as text in the document, the current font, the current font size, and the current selection

state of a toggle button. GUITAR captures state information for comparison with a pre-defined oracle when used as an automatic GUI tester. We have modified GUITAR to capture some extra details for our purpose.

The purpose of capturing the state is to determine whether CogTool-Helper needs to build a new frame for this state, or whether it should link to an existing frame. CogTool-Helper keeps a list of all states that have been encountered while building the design, each of which is linked to a particular frame. If the current state is not in the list, CogTool-Helper places it in the list, maps it to an empty frame, and sets the target of the transition to the empty frame. If it is in the list, CogTool-Helper sets the target of the transition Perform Action has just created to the frame associated with the current state. The transition associated with the Select All button in Frame 002 shown above, is encoded as follows, just before the end of the widget definition.

```
  <transition  durationInSecs='0.0'
               destinationFrameName='Frame 003'>
    <action>
      <mouseAction button='left' action='downUp'>
      </mouseAction>
    </action>
  </transition>
</widget>
```

Next, if the state was not new, CogTool-Helper repeats Perform Action for the current step in the test case. If the state was new, CogTool-Helper goes back to Build Frame to fill in the empty frame.

Once there are no steps left in the test case, CogTool-Helper has finished representing this method in CogTool XML and returns the set of states and frames that have been defined so far. These will be the input to the next method if there are any left to process. CogTool-Helper proceeds in the same way for every method of every task. Once all tasks are complete, the CogTool XML contains all the information for a complete CogTool design storyboard, with all frames, widgets and transitions.

The last part of the Task Replay process (Figure 3) is building CogTool scripts, i.e., representations of the demonstrated steps in each method in CogTool XML form. Scripts include the mouse and keyboard actions assembled by CogTool-Helper, to which CogTool will add psychologically valid "undemonstrated steps", like eye movements and thinking time, when it builds the ACT-R cognitive model. For each action in a method, CogTool-Helper creates a demonstration step in the script, with the widget on which the action was performed and the type of action (e.g., mouse click, mouse double-click, keyboard action). In the following step, the left mouse button is clicked on the Select All button.

```
<demonstrationStep>
  <actionStep
    targetWidgetName='SelectAll_button_1'>
    <mouseAction button='left'
            action='downUp'>
      </mouseAction>
  </actionStep>
</demonstrationStep>
```

After all tasks provided by the designer are scripted, Task Replay completes and the next phase of Design Construction, Method Inference, begins.

Method Inference. There may be alternative methods possible in the design that were not explicitly specified by the UI designer. CogTool-Helper uncovers these alternative methods and creates scripts for them so the UI designer can determine if their existence is a problem or an opportunity for the end user. This would be intractable if the UI designer had to manually create scripts for every possible path in the design. The method inference process generates all possible alternative methods.

Figure 4 is a schematic of the frames in the Enter, Center and Bold Text task in our example of CogTool-Helper. In Frame 3, "CHI2012" has been typed into a document in OpenOfficeWriter and is selected. The two paths shown on the left (red dotted line and black solid line) represent the two methods for centering and bolding the text that were created by the designer using CogTool-Helper. But there is nothing preventing an end user from taking different paths through the design. The right side of this figure shows two such paths (thin gray line and thick blue line), where the model will switch from using the toolbar buttons to using the menus (or visa versa) to accomplish the task. We call these *Inferred Methods*. The last part of the Design Construction phase is to calculate and create all of the possible inferred methods for a task.

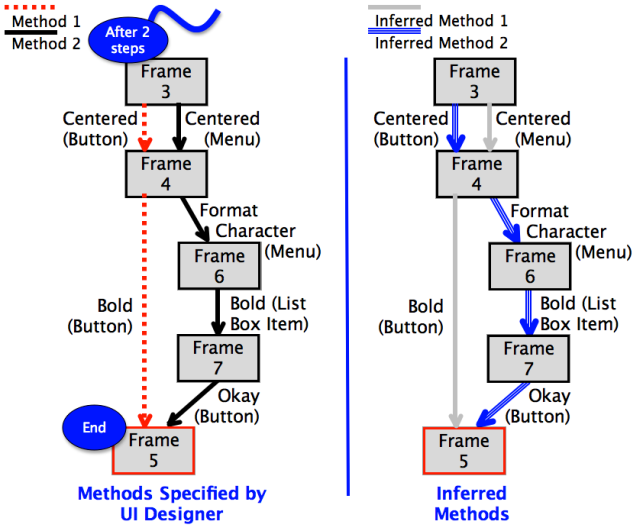

Figure 4. Example of Inferred Methods.

To compute these methods, we extract the portion of the design corresponding to a single task, and then build a directed graph where the nodes are the frames and the edges are the transitions. The graph is a multigraph because two nodes can be connected by more than one edge.

We use a depth first search algorithm to traverse this graph from the start to end node, storing each transition that we visit along the way. Once we reach the final node (the end state), we check to see if the path we have currently followed is in the set of paths we already have. If it is not, then we save this path as an inferred method, and create a CogTool method script for it.

Once we have created all of the inferred methods, CogTool-Helper has finished and we can import and analyze the CogTool Designs that we have created.

Importing Designs/Tasks from CogTool-Helper

Once CogTool-Helper is finished, the designer can launch CogTool in either MacOS or Windows and import their designs and tasks from the XML file. We have added an option to CogTool that will automatically construct and run the ACT-R model and calculate predictions for each task upon loading. This way, the designer can open the XML file and immediately compare each method of each task for performance.

Figure 5 shows the imported design, tasks, methods, and predictions. Notice that the "Enter, Center, and Bold Text" task includes six inferred methods.

Figure 6 shows the completed CogTool design storyboard. The frames used in the "Enter, Center, and Bold Text" task run down the left side. The frames for the "Insert Hyperlink" task run down the right side and share no frames with the other task except the first. A portion of the second frame in the storyboard (insert) shows part of the menu structure and toolbar widgets constructed by CogTool-Helper. The Format menu is expanded (occluding other widgets) and the Alignment item is selected, revealing the Centered item used in the Use Menus method.

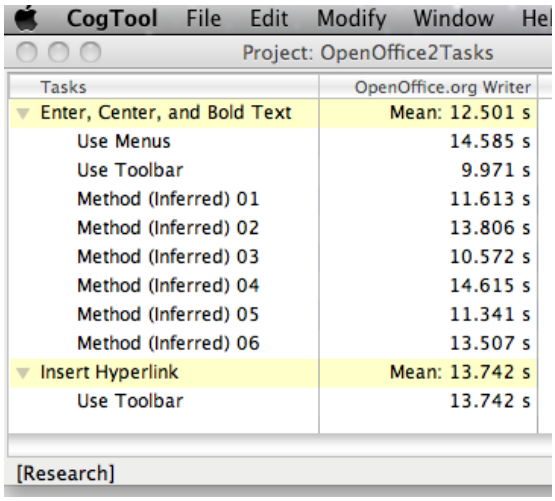

Figure 5. The Imported Design, Tasks and Predictions.

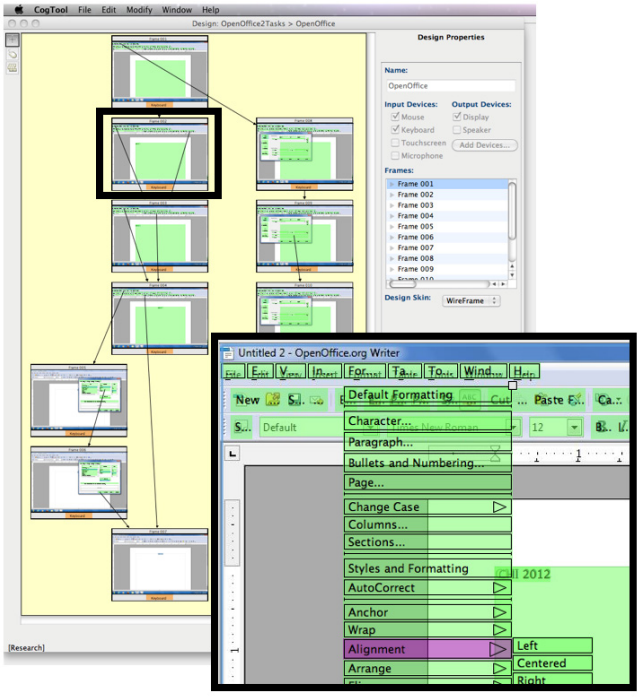

Figure 6. The Imported CogTool Design and One Frame.

Testing and Technical Limitations

CogTool-Helper currently works in two GUI environments. Since different application platforms have slightly different Accessibility APIs, we have to use different versions of the GUITAR framework for each environment and then customize the CogTool script creation. To date, our tool works on applications written in Java and on those that use the OpenOffice interface. Within OpenOffice, we have tested CogTool-Helper on each of the applications in the OpenOffice 3.0 suite including Impress, Calc, Math, Base, and Draw in addition to Writer, which has provided the extended example in this paper. Additionally, we have successfully used CogTool-Helper with LibreOffice 3.4.3, which is written on top of OpenOffice. For the Java UI, we have tested TerpWord and TerpSpreadsheet from the TerpOffice Suite, a set of Office Productivity tools written at the University of Maryland [12]. CogTool-Helper requires Java 1.6 and has been tested on a Windows 7 operating system. In the future we will incorporate more (and mixed) UI platforms into CogTool-Helper and test it on other operating systems.

The goal of this work is to make the entire process of creating the models completely automatic, however, there are two parts of our current implementation that are only semi-automated. The first part is task definition. The designer can either demonstrate the task by hand on the interface (which is automatically captured by CogTool-Helper), or provide test cases in the GUITAR format, which at this time would be manual. GUITAR includes a test case

generator, but we do not yet have a technique to generate specific tasks that a user may want to perform. As part of our future work, we are adding automated task and method generation. We plan to use an approach based on the AI planning work of Memon et al. [14].

The "Analyze Tasks" process is entirely automatic. The designer does not need to interact with the application at all during this stage. However, once the design is created, the designer must import this by hand into CogTool (done by a simple menu action). Automating the connection between CogTool-Helper and CogTool with the new design imported is also left as future work.

Although we have been able to create CogTool designs for all of the systems above, we have also identified a set of limitations. First we are currently constrained by the capabilities of the underlying GUITAR testing tools. For example, the current framework cannot perform tasks involving keyboard commands, such as Ctrl-C, and it does not yet support the right-click of the mouse. Second, we have mapped only a subset of accessibility types to their appropriate CogTool widgets. These cover the majority of widgets in the Java and OpenOffice systems, but we have not attempted to be complete, for instance, we do not yet handle combo buttons. Future work will be to expand our set of widget translations. Finally, we have only implemented the task capture feature for the OpenOffice applications. We plan to implement this for the Java applications as well.

As additional future work, we will add the ability for CogTool-Helper tasks to start in different states of the application and we will optimize the time it takes to build a design. We will also support designs where the menu structure can change dynamically as the application runs, which may require an alternate technique beyond capturing the menu only once at the start of our process.

A DESIGN WALKTHROUGH OF COGTOOL-HELPER

To learn whether automated analysis of legacy applications would be useful to existing CogTool users, and whether CogTool-Helper in particular would make it easy to do such analyses, we conducted a design walkthrough of CogTool-Helper. Four experienced CogTool users took part. Two were software engineers each with over thirty years of experience, one was an accessibility researcher with over twenty years of research and programming experience, and one was a usability researcher with over forty years of experience. Each had used CogTool for at least two years and had used it to create over 20 models of legacy systems in the past year; three had created over 30 models.

The participants were told that they were going to be shown a video demonstration of a tool we had designed to make it easier to create CogTool models of legacy applications. There was no instruction on CogTool-Helper, treating it as a walk-up-and-use application.

The design researcher advanced the video to a point before a user action is necessary (e.g., the first point would be where the CogTool-Helper GUI appears after launch, Figure 1 without any text fields filled in) and the participant was asked what s/he would do next. For certain interactions, participants were also asked what would happen next (e.g., after pressing the Create New Task button). In addition, we stopped the video after each UI content change and asked participants to describe what they were now seeing. After viewing the entire process (Figure 2), participants were asked for general feedback. During each participant session, we recorded audio and captured the screen.

Feedback

CogTool-Helper was designed to make it easier to create CogTool models of legacy applications, and our participants all recognized that indeed this was the case. One said: *"It's fantastic, I didn't have to do hardly anything."* Three of the four participants recognized that they would be more productive when analyzing legacy applications if they adopted CogTool-Helper. "*Even if it didn't do everything, it does enough that I would use it."* The fourth participant was hesitant about adopting the tool only because he doubted it could cope with multi-application tasks, command line applications and applications as complex as the programming tools he typically analyzed; when asked to assume it could, he said he would definitely try it.

Although recognizing the obvious value of CogTool-Helper to their work, these experienced CogTool users could anticipate a potential problem with task demonstration. CogTool-Helper currently requires its user to demonstrate tasks without error and two participants said it was difficult to demonstrate the correct task steps on the first try. Their current procedure for modeling legacy systems is to use screen capture to make a video of using the legacy system and extracting screen shots and actions by hand from this video to put into their CogTool models. Thus, they had extensive experience attempting to demonstrate tasks on systems they don't know well; they knew that errors were common. One asked: "*Does it let you cut out steps?"* Both requested support for editing a demonstrated task beyond text-editing the GUITAR format; a suggestion to be considered in a future redesign.

Inference of alternative methods for the task is a feature that excited all four participants. "*Wow, that's just impressive, that's pretty cool.*" Two of the participants who typically modeled complex applications wondered whether there would just be too many methods inferred in a complex application, but then speculated that they might be able to demo one method and it would help them discover all the alternatives, and could identify which was the most and least efficient. "*What's the minimal amount of things I could have done to do that?"* They both considered that this feature would be very useful in their modeling work.

In our design walkthrough participants were given no tutorial or other information about how CogTool-Helper worked. Without such information, the flow of CogTool-Helper was confusing to them. In particular, all four expressed surprise when the CogTool-Helper started to do the ripping/capturing of all the menu items. One of the participants expected a separation between creation of the storyboard, which he expected to happen as soon as he had specified the application to be analyzed, and creation of the model for the demonstrated tasks. We expect that minimal instruction, e.g, showing something like the flowchart in Figure 2 could alleviate this confusion.

There was also confusion about the use of "analyze" and "analysis" in the CogTool-Helper GUI as this term is also used to refer to the process of generating human performance predictions in CogTool. All four participants were confused, saying things such as: "*I don't know what start analysis means in this case.*" They expected CogTool-Helper to be more tightly integrated with CogTool. When asked what would happen when they started the analysis, one participant said: "*It's going to give me some kind of a representation of a model based on its interpretation of what the steps were..., I don't really know. I don't know*". This conceptual confusion cleared once the participants were told that CogTool-Helper created a CogTool XML file (again, showing them the flowchart in Figure 2 would have helped), and all participants suggested not using the term "analysis" in the CogTool-Helper UI, a suggestion we will certainly act on in future redesigns.

DISCUSSION AND CONCLUSION

We have presented CogTool-Helper, a tool that extends CogTool by providing an automated way to capture and generate design storyboards, tasks and methods for legacy applications as well as a way to uncover implicit methods that exist for certain tasks on certain designs. By automatically creating the CogTool model, CogTool-Helper allows the analyst to spend their limited time interpreting the result of the CogTool models instead of grabbing screen shots, adding overlays to indicate the location and size of widgets by hand, and typing in widget labels and other text. By inferring methods, CogTool-Helper aids the analyst to discover implicit methods for a task of which they may not have been aware, but may be important for them to consider. An implicit method may turn out to be the most efficient, even if not readily discoverable. Knowing this can provide a basis for a broader analysis.

As we discussed when introducing CogTool above, obtaining predictions of skilled execution time only requires a UI designer to represent the widgets on the correct path in the CogTool storyboard, so CogTool-Helper's representation of all widgets on every frame may seem like overkill for this usability metric. However, CogTool-Explorer [18] can predict novice exploration behavior, or the *discoverability* of how to accomplish a task on a new UI, using information foraging theory. This prediction requires a representation of all widgets because a cluttered screen does distract novice users and CogTool-Explorer must interact with fully-fleshed out storyboard to make its predictions of novice errors. Many realistic applications today are so full of features and widgets that constructing storyboards by hand for CogTool-Explorer will be a burden to the UI designer unless something like CogTool-Helper is also used.

Feedback collected in the design walkthrough of CogTool-Helper suggested that such a tool would improve the productivity of analysts by making them more efficient. We also uncovered specific areas to be examined in future design work. Of particular note are:

- How should CogTool and CogTool-Helper be integrated? For example, should CogTool-Helper be more tightly integrated with CogTool so that it immediately outputs the analysis results? Or is a loose integration more appropriate for analysts?
- How should analysts be supported in creating and editing task demonstrations?
- Can we support model creation for tasks involving use of multiple applications?
- Can we automatically generate methods for tasks using AI planning techniques where the analyst need only specify a task at a high level rather than demonstrate how to accomplish it?

This paper has focused on testing tools to facilitate creation of CogTool models of legacy applications. Such models are used to compare a new design with an existing one. CogTool modeling is also used during design exploration, allowing modeling of usability issues prior to coding. Facilitating creation of such models is another area where GUI testing technologies may be able to help. In this case, testing technologies based on computer vision such as [4] might facilitate creation of human performance models from hand-drawn user interface sketches.

We have shown how CogTool-Helper bridges the gap between the domains of software testing and HCI research. In doing so, it leverages advances in GUI automated testing to improve predictive human performance modeling in service of design. Bridging these two communities also benefits GUI testing research by highlighting concerns critical to HCI that are also important but too often ignored by the software testing community. For example, our user evaluation revealed the importance of capturing inter-application interactions, such as copying from one app to another. GUI testing tools have traditionally focused on analyzing single applications even though inter-application interactions are often a source of functional GUI errors.

More generally, we believe that bridging the gap between HCI and Software Engineering to create cross-cutting tools will benefit both communities. CogTool-Helper is an

example that shows how a tool for software testing can also be extended and integrated with usability testing tools. We believe that there are other such opportunities to integrate usability tools and methods with the tools and methods used throughout the software life-cycle and that such tool integration is key to integrating usability engineering with software engineering, to the benefit of both.

ACKNOWLEDGMENTS

We would like to thank Peter Santhanam from IBM Research for pointing out the connection between usability and functional GUI testing and Atif Memon from the University of Maryland for providing us with the newest releases of GUITAR, as well as technical support.

This work is supported in part by IBM, the National Science Foundation through award CCF-0747009 and CNS-0855139, and by the Air Force Office of Scientific Research through award FA9550-10-1-406. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the position or policy of IBM, NSF or AFOSR.

REFERENCES

1. Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., and Qin, Y. (2004) An integrated theory of the mind. *Psychological Review 111*, 4, 1036-1060.
2. Bellamy, R., John, B. E., Kogan, S. (2011) Deploying CogTool: Integrating quantitative usability assessment into real-world software development. *Proceeding of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 691-700.
3. Card, S. K., Moran, T. P., and Newell, A. 1983. *The Psychology of Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum Associates.
4. Chang, T., Yeh, T., and Miller, R.C. (2010) GUI testing using computer vision, *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10).* ACM, New York, NY, USA, 1535-1544.
5. Gray, W. D., John, B. E., & Atwood, M. E. (1993) Project Ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human-Computer Interaction, 8,* 237-309.
6. Grechanik, M. Xie, Q. and Fu C. (2009) Creating GUI testing tools using accessibility technologies, Software Testing Verification and Validation Workshop, 243-250.
7. John, B. E., Prevas, K., Salvucci, D. D., and Koedinger, K. 2004. Predictive human performance modeling made easy. In *Proceedings of the SIGCHI Conference on Human Factors in Computing* Systems (CHI '04). ACM, New York, NY, USA, 455-462.
8. Kieras, D. E. (1999). *A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN3*. University of Michigan. Available at: ftp://www.eecs.umich.edu/people/kieras/GOMS/GOMSL_Guide.pdf
9. Knight, A., Pyrzak, G., and Green, C. 2007. When two methods are better than one: combining user study with cognitive modeling. In *CHI '07 Extended Abstracts on Human Factors in Computing* Systems (CHI '07). ACM, New York, NY, USA, 1783-1788.
10. Memon, A.M. (2002), GUI testing: pitfalls and process, *IEEE Computer, 35*(8), 87-88.
11. Memon, A.M. (2011) GUITAR- A GUI testing framework, available at: http://guitar.sourceforge.net.
12. Memon, A.M. (2011) TerpOffice, available at: http://www.cs.umd.edu/~atif/TerpOffice/
13. Memon, A.M., Banerjee, I and Nagarajan, A. (2003) GUI Ripping: Reverse engineering of graphical user interfaces for testing, In *Proceedings of The 10th Working Conference on Reverse Engineering*, 260-269.
14. Memon, A.M., Pollack, M.E. and Soffa, M.L. (2001) Hierarchical GUI test case generation using automated planning, *IEEE Transactions on Software Engineering*, *27*(2), 144–155.
15. Monkiewicz, J. (1992). CAD's next-generation user interface. *Computer-Aided Engineering*, November, 1992, 55-56.
16. Pirolli, P. and Card, S.K. (1999). Information foraging. *Psychological Review, 106*, 643–675.
17. St. Amant, R., and Riedl, M. O. (2001). A perception/action substrate for cognitive modeling in HCI. *International Journal of Human-Computer Studies 55*(1), 15-39.
18. Teo, L., John, B. E., and Blackmon, M. H. (2012) 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 (CHI '12). ACM, New York, NY, USA.
19. White L. and Almezen, H. (2000). Generating test cases for GUI responsibilities using complete interaction sequences, in *International Symposium on Software Reliability Engineering (ISSRE)*, 110–121.
20. Yuan, X. and Memon. A.M. (2010). Generating event sequence-based test cases using GUI runtime state feedback. *IEEE Transactions on Software Engineering, 36*(1), 81-95.