Exploring the Internal State of User Interfaces by Combining Computer Vision Techniques with Grammatical Inference.

Paul Givens, Aleksandar Chakarov, Sriram Sankaranarayanan and Tom Yeh.
University of Colorado, Boulder.
Email: {paul.givens,aleksandar.chakarov,srirams,tom.yeh}@colorado.edu

***Abstract*—In this paper, we present a promising approach to systematically testing graphical user interfaces (GUI) in a platform independent manner. Our framework uses standard computer vision techniques through a python-based scripting language (Sikuli script) to identify key graphical elements in the screen and automatically interact with these elements by simulating keypresses and pointer clicks. The sequence of inputs and outputs resulting from the interaction is analyzed using grammatical inference techniques that can infer the likely internal states and transitions of the GUI based on the observations. Our framework handles a wide variety of user interfaces ranging from traditional pull down menus to interfaces built for mobile platforms such as Android and iOS. Furthermore, the automaton inferred by our approach can be used to check for potentially harmful patterns in the interface's internal state machine such as design inconsistencies (eg,. a keypress does not have the intended effect) and mode confusion that can make the interface hard to use. We describe an implementation of the framework and demonstrate its working on a variety of interfaces including the user-interface of a safety critical insulin infusion pump that is commonly used by type-1 diabetic patients.**

I. Introduction

In this paper, we present a framework which aims to discover the *internal* state diagram for an interface by interacting with it from *outside*. To enable this, we combine two seemingly disparate tools: (a) computer vision techniques to recognize features on an interface and interact with them [1]; and (b) grammatical inference techniques to learn the internals by observing the interaction [2]. We employ standard computer vision techniques implemented in a tool called *Sikuli script* to automate the interaction with the device [1]. Sikuli script provides the basic infrastructure required to recognize input and output elements on the graphical interface. It can be programmed to systematically interact with these elements while recording the sequence of inputs and outputs. Next, we employ grammatical inference (GI) techniques over the set of input/output sequences resulting from running Sikuli to infer the smallest automaton that is consistent with these observations [3], [2]. Such an automaton can be viewed as a likely model of the device's internal operation. The resulting model can reveal a lot about the interface, including: (a) *Implementation bugs*, wherein the interface implementation deviates from its design specification, (b) *mode confusion* patterns, wherein two internal states that perform inconsistent actions on the same input nevertheless present identical outputs [4], [5], [6], [7], and (c) *cognitive complexity measures* to quantify the usability of the interface for a set of common tasks [8].

This work was supported by the US National Science Foundation (NSF) under CPS award 1035845 and DARPA under grant number FA87501220199. All opinions expressed herein are those of the authors and not necessarily of NSF or DARPA.

The key advantages of our framework are two-fold (a) platform independence: our approach can (and has) been deployed across a variety of platforms such as Mac OSX, iOS and Android; and (b) precision: our approach does not necessitate an expensive and often imprecise analysis of complex code that implements the GUI features. We note that recognizing standard features in a graphical display is often a much easier problem than understanding the underlying code that implements them.

Figure 1 depicts the overall framework and the key components involved. The interface prototype is assumed to be run through a simulator on the screen.

Automated Interaction: The Sikuli script tool actively captures the screenshot image and is able to detect changes in the interface state. Sikuli uses pattern recognition techniques to recognize various input elements such as dialogs, pull down menus and text input areas. It can simulate pointer moves, clicks and key presses to interact with these elements. Our framework can use many exploration strategies including systematic depth first exploration or randomized interactions with periodic restarts. The framework records the sequence of inputs and outputs for each interaction. The outputs are recorded by computing a hash function over the designated output region images.

Grammatical Inference: The grammatical inference module uses the input/output sequences gathered by Sikuli script. It then attempts to build a finite state Moore machine model (FSM) that is consistent with the observations gathered from Sikuli. In other words, when fed an input sequence exercised by Sikuli, the inferred FSM will yield the corresponding observed output sequence. In accordance to the common Occam's razor principle in machine learning, the GI engine seeks a consistent FSM with the minimal number of states.

Backend Analysis: We have implemented some elementary

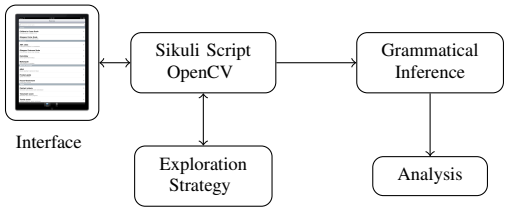

Fig. 1. Major components in our analysis framework and their interactions.

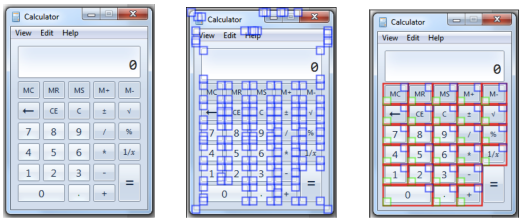

Fig. 2. Illustration of Sikuli's look and recognize actions. **(Left)** The original calculator interface, **(Middle)** Sikuli identifies patches that are edges of likely buttons and **(Right)** Sikuli identifies the buttons and the text/number on them.

backend visualization and analysis tools. The visualization tools reproduce the output of the interface, allowing the user to examine the automaton. The analysis tool can search for common property violations as specified by the user and check for potential mode confusion.

We have implemented a GI engine using the Z3 SMT solver [9]. As a proof of concept, we apply our implementation to two complex interfaces: (a) a windows scientific calculator and (b) a widely used commercial insulin infusion pump for infusing insulin to type-1 diabetic patients to control their blood glucose levels. We show how our framework goes beyond a simple aggregation of the GUI inputs and outputs to reveal facts about the internal state of the GUI.

Related Work: Tools such as *Monkey* can automate GUI fuzz testing in platforms such as Android, helping developers to stress test their applications [1]. Our approach is targeted towards finding subtle errors such as mode confusion that are hard to discover using a fuzzing approach. In particular, data from multiple traces need to be correlated for finding these errors. Our approach in this paper has been inspired by the previous work by Gimblett and Thimbleby [10]. Therein, the authors provide a formal framework for systematically interacting with a GUI and building models of the output. This work extends their program by providing a truly platform independent technique for interaction that is less cumbersome to implement and performing more ambitious analysis on the resulting interaction sequences through grammatical inference.

There have been many approaches to statically analyzing models of interfaces. These include the analysis of interface models for patterns that indicate mode confusion potential [5], [6] and a systematic exploration of the interface in conjunction with user models [11].

The inference of likely specifications by observing the executions of a program has been well studied by software engineering researchers [12], [13]. Our work differs in its focus on studying the internal state of graphical user interfaces as opposed to programming APIs. Grammatical inference techniques have been employed in this context to obtain abstractions for components [14].

Roadmap: The rest of this paper will provide further details on the various components of our framework. Section II provides a brief overview on Sikuli script and its use in our framework. Section III briefly describes the grammatical inference procedures used and our implementation. Section IV describes the experimental evaluation of the ideas presented in this paper. Concluding remarks are provided in Section V.

II. Sikuli Script

Sikuli [2] automates the interaction with a GUI by executing it, recognizing widgets such as buttons and text fields from their visual appearance on the screen, and interacting with those widgets by simulating mouse pointer or keyboard actions [1]. As such, the tool is intended to automate laborious tasks performed using a GUI and to automate GUI testing in a platform independent manner [15]. The three core technical capabilities of Sikuli are *Look*, *Recognize* and *Interact*. We explain each of these capabilities below.

Look: Sikuli uses a system API to grab the pixel data from the screen buffer and analyzes it. This basic system function for screen capture is available on most modern platforms including Windows, Mac, Linux and Android.

Recognize: Sikuli "recognizes" widgets on a GUI using pattern matching based on visual appearance. There are two use cases that must be dealt with separately: recognizing a specific widget and recognizing a class of widgets. To recognize a specific widget such as the `OK` key, Sikuli needs a programmer to provide an example image of the key. It uses standard template matching techniques to recognize areas in the screen that match the given image. To recognize a class of widgets such as all the buttons on a GUI, we exploit the consistency in the style of GUI elements across interfaces. Sikuli extracts style features for common GUI elements including the corners (see Fig. 2), the border, the background color, and the foreground color. Sikuli scans the screenshot from the GUI, and first looks for all small patches of pixels that exhibit some preset style features. It then considers all possible groupings of these patches and score each grouping based on geometry (e.g., are corners aligned?) and continuity (e.g., are edges connected?). Finally, groupings with scores above a threshold are recognized, as illustrated in Figure 2.

Interact: Sikuli uses the Java Robot class to simulate mouse and keyboard interaction. After recognizing a widget, Sikuli knows that widget's location on the screen, and can move the pointer to that location. At that location, it can issue a click command which will effectively click on that widget. If that

[1]Cf. http://developer.android.com/tools/help/monkey.html/

[2]Sikuli script is available free online at http://www.sikuli.org.

widget is a text field, the widget can gain focus. Sikuli can type or paste text into the text field.

We use Sikuli with two basic exploration strategies: (a) random exploration first identifies all widgets on the current screen image and chooses to interact with one uniformly at random. The interaction is continued upto a user given length cutoff and restarted from the initial state of the application. (b) Depth-first search exploration systematically explores all paths upto a given depth. Systematic exploration relies on the basic assumption that the application is deterministic.

III. Grammatical Inference

Grammatical inference pertains to a large area of machine learning and natural language processing that is concerned with learning an unknown language from observations [2]. Our framework in this paper uses grammatical inference techniques to infer parts of the internal state machine from the set of input/output sequences produced from the interaction. As mentioned earlier, this is achieved by finding the smallest size automaton that is consistent with the recorded observations.

Unfortunately, the problem of finding the smallest automaton is well known to be NP-hard, and thus quite hard to solve in practice [16]. However, the significant breakthroughs achieved in solving the propositional SAT problems over the last decade gives us hope that grammatical inference is feasible for large enough inputs. Promising, SAT-based schemes for learning minimal consistent automaton have been proposed by Gupta et al. [14] and by Heule & Verwer [17]. However, these techniques have been proposed for learning FSMs with two types of labels (accepting or rejecting). Our setup requires us to learn a Moore machine with many labels arising from the interface output images. The encodings obtained are quite large (quadratic in the size of the trie) in this case since a disequality constraint is needed for each pair of nodes with differing output and there are many such pairs.

We use an SMT-based algorithm for grammatical inference that is suited for inference of Moore machines which have output labels associated with states. Our approach encodes the problem of finding the smallest automaton into a set of constraints which are solved to obtain the smallest solution. Our approach first builds a trie based on the input/output sequences. The nodes of the trie with the same output labels can potentially be merged to yield the final FSM. However, care must be taken to ensure that nodes that yield different outputs on the same inputs should not be merged. We encode these constraints in linear arithmetic by associating a number with each node. Nodes with the same number are considered to be merged. Our tool uses the SMT solver Z3 to solve the constraints [9]. Our approach uses heuristic clique finding techniques to set certain nodes in the trie to fixed constant IDs. This yields a significant reduction in the running time.

IV. Experiments

In this section, we describe the use of our tool on two example interfaces: a calculator application that is available in windows and the mockup of a commercial insulin infusion pump that is used by diabetic patients to infuse insulin.

TABLE I
Performance of Grammatical Inference Engine on benchmark examples. **Inter**: total length of interaction with Sikuli, **#Reset**: number of times the interface was forced to reset to the start state, **#Trie**: length of the trie, **#AutSize**: final automata size learned, **Time** (seconds) and **Memory** (MBs).

ID	Inter	#Reset	#Trie	#AutSize	Time	Mem
Pump	400	20	360	13	0.6	115
Pump	1000	20	965	15	21.8	581
Pump	1000	50	871	15	16	447
Pump	1250	25	1199	15	95.4	903
Pump	2500	50	2367	- mem out-		
Calc	100	5	96	24	0.1	78
Calc	500	20	350	49	10h10m28s	1957

Windows Calculator: The windows calculator was used off-the-shelf (see Figure 2 (**left**)). Sikuli was used to identify buttons on the calculator and systematically interact with them. The screen was recorded for each interaction as the output. Our use of sikuli did not involve any kind of instrumentation of the calculator code. The input/output sequences were recorded and analyzed by our GI solver to yield a likely state machine describing the internals of the calculator. We note that the calculator can have a large number of internal states depending on the current output of the calculator and the contents of its internal registers. To simplify the problem, we restricted Sikuli to press the numerical key "5" along with arithmetic operators and the "=" key. This vastly reduces the number of states of the internal machine that is well in excess of 10^{12} to a much smaller set. Figure 3 (a,b) depict the infered automton. Fig. 3 (b) demonstrates how GI techniques identify different internal states even when the output is the same.

Insulin Pump Mockup: We performed a mockup of a commonly available commercial insulin infusion pump model. Figure 4 shows the original pump placed alongside a mockup constructed. A physical pump served as the basis for constructing the mockup. Figure 3(c,d) show the inferred likely state diagram. In Fig. 3(d), we highlight a potential mode confusion involving two states with the same display "Enter BG" that prompts the user to enter the blood glucose by pressing arrow keys. In one instance, the "B" key leads to a new screen that prompts for the manual bolus type whereas in the other case, it does not have an effect. On the other hand, pressing the "A" key in both modes have similar effects. Another potential interface fault involves the "E" key, which serves to take the user back to the previous screen on all but a few notable occasions. We are investigating these findings to check if they can potentially lead to dangerous mode confusion scenarios in diabetic patients.

Table I summarizes the performance of the GI engine. We note that GI can be carried out for interactions that are longer than 1000 steps for the Pump example and around 400 steps for the calculator. However, the running time and memory used increase sharply beyond a limit that varies, depending on the

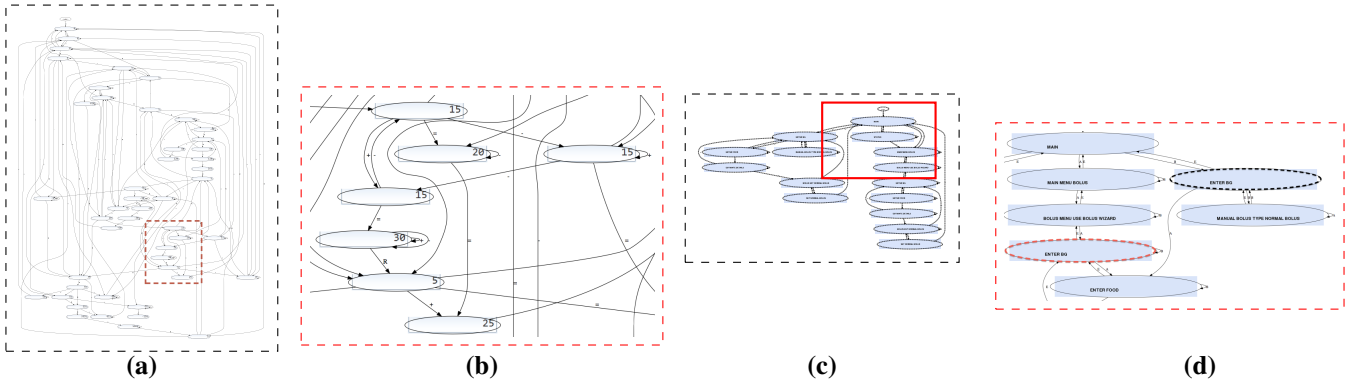

Fig. 3. **(a)** Inferred automaton for windows calculator, **(b)** Magnified region showing the state transitions. Each state shows the image captured on the output area. **(c)** Inferred automaton for the insulin infusion pump and **(d)** Magnified region showing the states and transitions between them. Two highlighted states have identical screens but entirely different actions when keys "E" or "A" are pressed showing potential mode confusion.

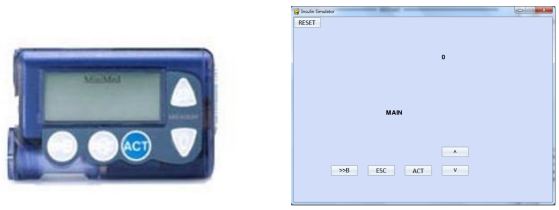

Fig. 4. (**Left**) Original pump interface and (**Right**) a mockup constructed by physically interacting with a pump.

interface. The number of states in the final machine is strongly correlated with the overall running time, while the length of the interaction matters to a lesser extent.

V. Future Directions

We have presented a framework that combines computer vision techniques with grammatical inference to yield a likely model for the internal state. We have noted the ability to uncover potential interface inconsistencies and mode confusion errors. Our framework, however has a long way to go. Our current implementation can handle interactions of upto a 1000 input actions. We are considering improvements to symbolic encodings that can yield further increases in the size of the interactions that can be handled. Yet another challenge is to use "plausible" inputs that are likely to be exercised by a human [18]. To address this, we are considering the integration of ideas from cognitive analysis of interfaces such as automated design walkthroughs [19], cognitive complexity metrics to predict user performance [8] and integratio with cognitive architectures [20]. The ideas presented here can naturally complement some emerging work in the broader area of program *synthesis* from demonstrations [21].

Acknowledgents: We wish to acknowledge Prof. Clayton Lewis for giving us helpful insights, and introducing some of the authors to this problem domain.

References

[1] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," in *UIST'09*. ACM, 2009, pp. 183–192.

[2] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[3] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys*, vol. 15, no. 3, Sept. 1983.

[4] N. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and J. D. Reese, "Analyzing software specifications for mode confusion potential," in *Proc. Human Error and System Development*, 1997, pp. 132–146.

[5] J. Rushby, "Using model checking to help discover mode confusions and other automation surprises," in *Proc. HESSD'99*, June 1999.

[6] A. Joshi, S. P. Miller, and M. P. Heimdahl, "Mode confusion analysis of a flight guidance system using formal methods," in *DASC'03*, 2003.

[7] N. Sarter, D. D. woods, and C. Billings, "Automation surprises," in *Handbook of Human Factors and Ergonomics*. Wiley, 1997, pp. 1–25.

[8] B. E. John and D. E. Kieras, "The GOMS family of user interface analysis techniques: comparison and contrast," *ACM Trans. Comput.-Hum. Interact.*, vol. 3, no. 4, pp. 320–351, 1996.

[9] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[10] A. Gimblett and H. Thimbleby, "User interface model discovery: towards a generic approach," in *EICS'10*, 2010, pp. 145–154.

[11] M. Bolton, R. Siminiceanu, and E. Bass, "A systematic approach to model checking human-automation interaction using task analytic models," *IEEE Trans. Systems, Man and Cybernetics, Part A*, vol. 41, no. 5, pp. 961 –976, sept. 2011.

[12] J. W. Nimmer and M. D. Ernst, "Automatic generation of program specifications," in *ISSTA '02*. ACM press, 2002, pp. 232–242.

[13] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *POPL*. ACM, 2002, pp. 4–16.

[14] A. Gupta, K. L. Mcmillan, and Z. Fu, "Automated assumption generation for compositional verification," *Form. Methods Syst. Des.*, vol. 32, no. 3, pp. 285–301, 2008.

[15] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *CHI'10*. ACM, 2010, pp. 1535–1544.

[16] E. M. Gold, "Complexity of automaton identification from given data," *Information and Control*, vol. 37, pp. 302–320, 1978.

[17] M. J. H. Heule and S. Verwer, "Exact DFA identification using SAT solvers," in *Intl. Conf. on Grammatical Inference (ICGI)*, ser. LNAI, vol. 6339. Springer–Verlag, 2010, pp. 66–79.

[18] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P. Jones, "Model-based testing of GUI-driven applications," in *Prof. SEUS'09*, ser. LNCS, vol. 5860. Springer, 2009, pp. 203–214.

[19] P. Polson, C. Lewis, J. Rieman, and C. Wharton, "Cognitive walkthroughs: A method for theory-based evaluation of user interfaces," *International Journal of Man-Machines Studies*, vol. 36, pp. 741–773, 1992.

[20] J. R. Anderson, D. Bothell, , M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind," *Psychological Review*, vol. 101, no. 4, pp. 1036–1060, 2004.

[21] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.