Goodbye Repetitive Tasks, You Won't Be Missed

Christina Chung
UTORid: chungc37
Teaching Labs ID: g4rice

ABSTRACT
The advent of the graphical user interface (GUI) has provided users with an easy and intuitive way to operate a computer. Although it alleviates many of the struggles faced by laypersons when adopting new technologies, the GUI suffers from an inherent lack of programmability and the ability to specify abstract commands. Oftentimes, users must manually carry out repetitive tasks, such as the transferring of multiple files from one application into another. This work provides an overview of the literature that has insofar attempted to support repetitive GUI tasks, the problems that still need to be addressed, and possible directions for future work.

DIRECT MANIPULATION IS PASSÉ
Since its introduction in the Apple Macintosh in 1984, the graphical user interface (GUI) supplanted the traditional command line and has come into widespread use. By making elements tangible and directly manipulable, the GUI provides an easy-to-use interface for non-programmers to operate a computer [25]. But with the good comes the bad: while complex, low-level operations of a computer can now be abstracted away, the GUI suffers from an inherent lack of programmability, subjecting users to manually performing repetitive tasks [20].

In concrete terms, a repetitive task is a sequence of actions that is repeated multiple times in succession, such as renaming several files in a directory. Tasks such as these can be tedious for the able-bodied, and challenging for the impaired [13]. Command lines, on the other hand, do not have this problem—users with the expertise can carry out tasks with ease and efficiency. Because of this, some users feel that using a GUI is sometimes more difficult than necessary, and frequently find themselves relegating back to the command line.

So, problem solved. Take the best of both worlds and use command lines and direct manipulation, depending on what the situation calls for. Perhaps this is a viable option for adept users, but learning how to code is not easy an endeavor and many non-programmers struggle when attempting to do so [23].

The begs the question of how one would efficiently carry out repetitive tasks and without being encumbered by command lines. Naturally, this would entail bridging the user-programmer gap. A popular approach is to provide support for end-user programming; that is, the ability for non-expert users to program their computers [20]. This review surveys the literature that has sought to facilitate end-user programming, further discusses the technologies that have been developed for supporting repetitive tasks on the GUI and areas where additional work is needed.

BRIDGING THAT USER-PROGRAMMER GAP
Human brains are hardwired to process multi-faceted data. Textual code, which presents instructions in a one-dimensional manner, does not leverage the full capabilities of the human brain. Visualizations, on the other hand, are upheld as useful aids in program understanding [21].

To assuage the challenges of parsing and writing code, early approaches explored the use of visualizations. The earliest example was in 1959, when Haibt developed a system that could generate flowcharts describing code written in Fortran or Assembly [12].

Since then, researchers also played with the idea of incorporating visualizations into the coding process, giving birth to the notion of a *visual program*. In Myer's definition, a visual program is "any system that allows the user to specify a program in a two (or more) dimensional fashion" [21], for example, by graphically representing sequences of commands that can compiled into executable instructions.

Visual programming has its roots dating back to as early as the 1960s, when Sutherland developed a graphical programming system (with semblance to circuit diagrams), that could compute primitive operations such as the square root of a number [26]. Further work explored the use of flow charts. Grail, for instance, would compile into machine language directly from charts produced by the user [11] and in a similar vein, GAL compiled into Pascal [1].

In spite of this, the skills needed to parse a visual program are almost up to par with those in making sense of textual code. While these tools may benefit those who are already experienced programmers, they lack practicality for the layperson. Furthermore, when procedures grow in size and complexity, visual code often becomes overwhelming, both in terms of physical space that they occupy and the cognitive load that they impart on the user. Visual code begins to resemble a "maze", as Myer purports [21]. In such situations, visualizations lack feasibility.

To truly bridge the non-programmer technological gap, all code must be abstracted away. This is where programming-by-demonstration (PBD) comes into play. In PBD, the computer attempts to intuit intended behaviors from user demonstration [7]. PBD systems are a particularly attractive options, because they make it possible for users to produce executable instructions without having to touch a single piece of code. The very first PBD systems, however, were

quite rudimentary—users would essentially provide examples of the input and the desired output of a program, the computer would then use this information to infer executable instructions [24]. Later on, more advanced systems were developed that could actually learn from users demonstrating behaviors [8,19].

GOODBYE REPETITIVE TASKS

To assist users in carrying out repetitive tasks, many technologies turned to PBD. In essence, these technologies attempt to automate repetitive behaviors by learning from user demonstration. Some early examples include CoScriptor for automating web-processes [16], DocWizards supports tasks for the Eclipse platform [4], and SMARTedit automates the formatting of text based on a few examples provided by the user [15].

These systems, however, are limited in a number of ways: they cannot support cross-application interaction, they are unable to generalize beyond what was demonstrated by the user, and they lack the ability to understand user context.

PROBLEMS, MORE PROBLEMS, AND A SOLUTION?

The issues surrounding prior GUI automation systems in supporting repetitive tasks, as well as how modern-day systems have sought to address them, are discussed below. We further describe a prototypal system that we developed for the purposes of experimenting with new design considerations.

Cross-application Interaction

One primary challenge in supporting automation on a cross-application level is knowing where GUI elements are positioned, as this information is often not provided by application developers. To detect GUI elements, a popular technique is to leverage computer vision, using variations of template matching and feature detection [18,27]. Tasks are then automated by detecting the location of the GUI elements in question, and inducing the necessary user actions on these elements. For instance, if one would like to automate the process of deleting a file, these systems would first detect the file's location, move the mouse to where the file is located on the screen, and then perform the click events needed to delete the file. These computer visions techniques have been used by a number of applications: providing contextual help when interacting with a desktop computer [28], testing GUI elements [5], creating context-aware video tutorials [6,22], and automating GUI tasks on computers [13].

Unfortunately, computer vision techniques break down when GUI elements substantially change in appearance. With this approach, users are also literally shown the sequences of actions being carried out on the screen during the automation of a task. We believe this may be hindrance, as it precludes users from engaging in other computer activities while tasks are executing. Updating the GUI repeatedly also consumes computational resources, leading to an increased processing time.

Generalizability

Many earlier systems lack the ability to generalize to new tasks, and simply repeat exactly what was done by the user—nothing more, nothing less. Modern systems have attempted to support some generalizability. Notably, Help, It Looks Confusing (HILC), which automates GUI interactions on desktop computers might detect that a user is importing images into PowerPoint generalize this behavior to other images [13]. Additionally, SUGILITE automates GUI tasks on mobile devices [17]. One might teach SUGILITE to order a Starbucks Cappuccino but the macro for that task could be generalized to ordering an Iced Cappuccino as well. Yet, the functionality is still quite limited. For instance, a macro for ordering and Ice Cappuccino in SUGILITE would not be able to work for ordering a coffee, despite the similarity between the two tasks. Furthermore, generalizability brings forth a new problem. To the user, it can be a mystery how behaviors are learned by the system. Users may find themselves uncertain of whether the system will do as intended, or go completely awry. Often, they feel apprehensive about running automations [14].

To address this problem, systems supporting generalizability incorporate corrective mechanisms, yet the approaches taken are laden with issues. For example, users of SUGILITE have access to each macro's source code, for which they can modify to its intended behavior. But in its evaluation, users did not perceive the feature to be useful, as the code was often hard to parse. In a similar vein, HILC queries users with follow-up questions if an action is deemed ambiguous. This may diminish the user's sense of control, since they cannot modify macros directly to their intentions, but must rely on the system to propose them [3].

Context-awareness

To automate repetitive tasks, previous systems employed macros, which essentially require users to pre-record the actions that they would like to automate. These macros could then be replayed by the user when needed.

With advancements in artificial intelligence, there is a growing desire to have systems that can understand its user's intentions [3]. This is known as *context-awareness*, the ability of a computer "to provide relevant information and/or services to the user" [10]. With respect to PBD systems, this would entail programming the system to detect repetitive behavior purely by recognition (i.e., without the use of a macro), as well as identifying the contexts and conditions for which a macro should execute. Systems should be able to recognize the time of day or location for which an automation should occur, for instance.

Unfortunately, GUI automation systems have not been able to meet these goals. At present, identifying repetitive behavior without prior knowledge is a complex problem that necessitates robust noise and sequence detection in high-volume, high-dimensional data [9]. And unlike other approaches in machine learning, which have access to large

amounts of training data, PBD systems must learn behavior from a few examples.

Some researchers have forayed into this domain, but have largely failed due to faulty detections [2]. Resultantly, many modern systems, such as HILC and SUGILITE, resort to the use macros.

Autopilot: A Prototypal System

Autopilot is a prototypal system that we developed to test the feasibility of some novel interaction techniques for supporting repetitive tasks. Autopilot, like other modern-day systems, leverage PBD, given its benefits in eliminating all need to code. The design of Autopilot was largely shaped by SUGILITE and HILC, as they are both, respectively, the state-of-the art for mobile and desktop GUI interactions. Due to current setbacks that impede accurate detection of repetitive behavior, like SUGILITE and HILC, Autopilot opted for the use of macros. The system is also able to generalize its macros to other tasks that are similar in nature.

Autopilot proposed two design considerations aimed to improve existing systems. The first were its corrective mechanisms. In Autopilot, users may remove any undesired action from a macro. Secondly, using principles in visual programming to help users make sense of commands [21], each step of a macro is displayed with an accompanying screenshot and a descriptive text of the associated action. SUGILITE, on the other hand, provides the users with editable source code, which was reportedly hard to parse; and HILC queries users with follow-up questions if an action is deemed ambiguous, rather allowing users to decide when corrective measures are needed. We posit that this design choice might compromise the user's sense of control.

In HILC and SUGILITE, macros are literally carried out on the screen. We believe this is a shortcoming, since it precludes engaging in other computer activities during the execution of a macro. For example, when ordering an Iced Cappucino, SUGILITE literally has to pull up the Starbucks application and manipulate GUI elements on the screen. The user is then forced to halt whatever tasks they were previously working on. Autopilot mitigates this issue by enabling repetitive tasks to occur behind the scenes.

In Autopilot's preliminary evaluation, we were unable to determine whether these design considerations are indeed better than those of the state-of-the-art. However, we did receive insights regarding each participant's thoughts on GUI automation systems. Essentially, the same concerns about prior systems were brought up by our participants. Notably, participants were apprehensive about running a macro as they were unsure of how it might alter the state of their operation system. Participants also expressed the desire for advanced capabilities, such as greater context-awareness, by being able to detect repetitive behaviors without the use of a macro.

TOWARDS THE FUTURE

PBD is now one of the more largely attempted approaches used to support repetitive GUI tasks. Yet, the paradigm still faces a slew of issues that need to be tackled.

Hide it All

Users should not have to see GUI tasks being carried out on the screen. Developers and designers should seek ways to make this possible. Perhaps this could be solved by having developers of applications provide an API for interacting with GUI elements, despite the additional overhead in supporting the functionality. This will allow for GUI operations to occur behind the scenes, rather than having to induce literal mouse and keyboard events using the traditional computer vision approach [27].

Correct Me If I'm Wrong

Moving forward, it is of paramount importance to find an easy way for users to visualize an automation's behavior and modify its rules if needed, perhaps by exploring different visualization techniques. Existing systems currently have no adequate way of doing this and suffer tremendously in this domain. Better corrective mechanisms will increase the amount of trust that users have in their systems, and prevent these systems from going awry.

Sweeping Generalizations Are Good

The generalizability of these systems are still subpar in comparison to user expectations and more work is needed in developing better algorithms for generalizing learned rules to other tasks.

Provide Some Context

Attention should be directed toward improving context-awareness. For example, participants in our evaluation of Autopilot seemed to want the system to accomplish more than its current capabilities, for instance, by defining the specific contexts for which automation should occur, and to detect repetitive behavior purely by recognition. This will ultimately entail giving computers the ability to identify of human routine behavior, and the contexts and conditions for which particular repetitive tasks should be automated.

A Right Amount of Balance

According to Barkhuus and Dey, context-aware applications can be characterized in two ways [3]. On the one hand, the system can carry out actions automatically regardless of the user's intentions. This is known as *active awareness*. *Passive awareness*, on the other hand, is when the system has contextual information, but does not act upon it without the user's permission. In their study, Barkhuus and Dey found that awareness compromises one's sense of control, since activities that one would normally carry out are now taken over by the system.

As context-aware designs are becoming increasingly prevalent [3], a question of growing concern is how to seek a balance between providing enough context without compromising too much control.

CONCLUSION

Without a doubt, the GUI supersedes the command line when it comes to bridging the user-programmer gap. Sadly, it lags far behind in supporting repetitive tasks. In this review, we survey prior work that has attempted to address this shortcoming, chiefly by supporting end-user programming. From assisting the process of writing code to writing code on the user's behalf—we begin with the grounding literature on visual programs and move onwards to discuss modern-day systems that leverage programming-by-demonstration. While state-of-the-art technologies have come a long way since the GUI was first conceived, there are still a host of issues that have yet to be solved. Primarily, there is a divide between users desires and what can be accomplished with existing technology. Future work should seek to bridge this divide.

REFERENCES

1. Miren Begona Albizuri-Romero and Miren Begona. 1984. GRASE: a graphical syntax-directed editor for structured programming. *ACM SIGPLAN Notices* 19, 2: 28–37. https://doi.org/10.1145/948566.948567
2. Ville Antila, Jussi Polet, Arttu Lamsa, and Jussi Liikka. 2012. RoutineMaker: Towards end-user automation of daily routines using smartphones. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, 399–402. https://doi.org/10.1109/PerComW.2012.6197519
3. Louise Barkhuus and Anind Dey. 2003. Is Context-Aware Computing Taking Control away from the User? Three Levels of Interactivity Examined. . Springer, Berlin, Heidelberg, 149–156. https://doi.org/10.1007/978-3-540-39653-6_12
4. Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. 2005. DocWizards: A System for Authoring Follow-me Documentation Wizards. In *Proceedings of the 18th annual ACM symposium on User interface software and technology - UIST '05*, 191–200. https://doi.org/10.1145/1095034.1095067
5. Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. 2010. GUI Testing Using Computer Vision. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, 1535. https://doi.org/10.1145/1753326.1753555
6. Kai-Yin Cheng, Sheng-Jie Luo, Bing-Yu Chen, and Hao-Hua Chu. 2009. SmartPlayer: user-centric video fast-forwarding. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, 789. https://doi.org/10.1145/1518701.1518823
7. Allen. Cypher, Daniel Conrad. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky. 1993. *Watch what I do : programming by demonstration*. MIT Press.
8. Allen Cypher and David Canfield Smith. 1995. KidSim: end user programming of simulations. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '95*, 27–34. https://doi.org/10.1145/223904.223908
9. Himel Dev and Zhicheng Liu. 2017. Identifying Frequent User Tasks from Application Logs. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces - IUI '17*, 263–273. https://doi.org/10.1145/3025171.3025184
10. Anind K. Dey and Anind K. 2001. Understanding and Using Context. *Personal and Ubiquitous Computing* 5, 1: 4–7. https://doi.org/10.1007/s007790170019
11. T. O. Ellis, J. F. Heafner, and W. L. Sibley. 1969. The Grail Project: An Experiment in Man-Machine Communcation.
12. Lois M. Haibt and Lois M. 1959. A Program to Draw Multilevel Flow Charts. In *Papers presented at the the March 3-5, 1959, western joint computer conference on XX - IRE-AIEE-ACM '59 (Western)*, 131–137. https://doi.org/10.1145/1457838.1457861
13. Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. 2017. Help, It Looks Confusing: GUI Task Automation Through Demonstration and Follow-up Questions. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces - IUI '17*, 233–243. https://doi.org/10.1145/3025171.3025176
14. Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4: 65. https://doi.org/10.1609/aimag.v30i4.2262
15. Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1/2: 111–156. https://doi.org/10.1023/A:1025671410623
16. Gilly Leshed, Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. CoScripter: Automating and Sharing How-To Knowledge in the Enterprise. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, 789.
17. Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. 2008. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems - CHI '08*, 1719-1728.
18. D.G. Lowe. 1999. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 1150–1157 vol.2. https://doi.org/10.1109/ICCV.1999.790410
19. Richard G. McDaniel and Brad A. Myers. 1999. Getting more out of programming-by-demonstration. In *Proceedings of the SIGCHI conference on Human*

factors in computing systems the CHI is the limit - CHI '99, 442–449. https://doi.org/10.1145/302979.303127

20. B.A. Myers. 1992. Demonstrational interfaces: A step beyond direct manipulation. *Computer* 25, 8: 61–73. https://doi.org/10.1109/2.153286
21. B. A. Myers, B. A., Myers, and B. A. 1986. Visual programming, programming by example, and program visualization: a taxonomy. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '86*, 59–66. https://doi.org/10.1145/22627.22349
22. Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. 2011. Pause-and-play: Automatically Linking Screencast Video Tutorials with Applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, 135–144. https://doi.org/10.1145/2047196.2047213
23. Haider Ramadhan and Benedict du Boulay. 1993. Programming Environments for Novices. . Springer, Berlin, Heidelberg, 125–134. https://doi.org/10.1007/978-3-662-11334-9_12
24. David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs From Examples. *Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1*, 260–267. Retrieved December 4, 2017 from https://dl.acm.org/citation.cfm?id=1624666
25. Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8: 57–69. https://doi.org/10.1109/MC.1983.1654471
26. William Robert Sutherland. 1966. The On-Line Graphical Specification of Computer Procedure. Massachusetts Institute of Technology.
27. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology - UIST '09*, 183–192. https://doi.org/10.1145/1622176.1622213
28. Tom Yeh, Tsung-Hsiang Chang, Bo Xie, Greg Walsh, Ivan Watkins, Krist Wongsuphasawat, Man Huang, Larry S. Davis, and Benjamin B. Bederson. 2011. Creating Contextual Help for GUIs Using Screenshots. In *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, 145. https://doi.org/10.1145/2047196.2047214