Win GUI Crawler: A tool prototype for desktop GUI image and metadata collection

Marko Savic, Mika Mäntylä, Maëlick Claes
ITEE, M3S
University of Oulu
Oulu, Finland
Email: marko.savic@oulu.fi, mika.mantyla@oulu.fi,
maelick.claes@oulu.fi

***Abstract*—Despite the widespread of test automation, automatic testing of graphical user interfaces (GUI) remains a challenge. This is partly due to the difficulty of reliably identifying GUI elements over different versions of a given software system. Machine vision techniques could be a potential way of addressing this issue by automatically identifying GUI elements with the help of machine learning. However, developing a GUI testing tool relying on automatic identification of graphical elements first requires to acquire large amount of labeled data. In this paper, we present Win GUI Crawler, a tool for automatically gathering such data from Microsoft Windows GUI applications. The tool is based on Microsoft Windows Application Driver and performs actions on the GUI using a depth-first traversal of the GUI element tree. For each action performed by the crawler, screenshots are taken and metadata is extracted for each of the different screens. Bounding boxes of GUI elements are then filtered in order to identify what GUI elements are actually visible on the screen. Win GUI Crawler is then evaluated on several popular Windows applications and the current limitations are discussed.**

***Keywords*-test automation, machine vision, data collection, gui element identification, graphical user interface.**

I. Introduction

In the past decade, test automation has become the state of the practice in the software industry. Still, automatic testing of graphical user interfaces (GUI) remains a challenge and manual testing of GUI is still a common practice. A recent systematic review article reported on challenges with GUI test automation [1] and found that the most frequently reported challenge was that application changes break test automation execution. That paper postulates that *"Robust identification of GUI widgets ... challenge is possibly related, or even the leading cause of many of the other reported [GUI test automation] challenges."*

Numerous computer vision problems have found novel and improved solutions thanks to the widespread adoption of machine learning [2]. Machine learning approaches lead to higher generalization capacity and thus would make GUI element identification more robust and less susceptible to design changes. An additional area to GUI test automation that would greatly benefit from more robust GUI identification is Robotic process automation (RPA) where automated (robotic) tasks fail when GUI elements are not properly identified and located [3].

However, in order to be able to identify GUI elements automatically, one first has to gather labeled data in order to train a machine learning model. Extensive research has been dedicated to machine vision on Mobile and Web domains, resulting in the availability of several open GUI databases for those domains [4], [5], [6], [7], [8], [9], [10], [11], [12]. To date, there is no publicly available Desktop GUI image database for Windows applications that is still dominant in the desktop world. All existing databases are proprietary to tool vendors and are unlikely to get published due to the commercial advantages they offer. To build and apply machine learning models to the desktop domain, researchers require an extensive database containing GUI images and accurate annotations. To fill this gap and encourage research in the desktop domain, Win GUI Crawler[1], which is presented in this paper, was developed. This prototype tool can automatically traverse a Windows desktop application and collect screenshots alongside XML metadata. The tool can be employed to compile a large-scale desktop GUI database and to aid existing automation methods such as template matching.

II. Related Work

The currently available GUI image databases have been compiled by adopting diverse methods of data collection and annotation. Image data and annotations have been obtained with four main approaches of crowdsourcing, automation, synthetic generation, and manual collection/annotation. However, the approaches are not fully orthogonal to each other as they are often combined and manual collection/annotation can be seen as part of crowdsourcing. Regardless we present prior works under these four approaches.

Crowdsourcing for data collection and annotation has been widely employed in computer vision, due to the need for large-scale labeled data for supervised machine learning tasks. Deka et al. [4] compiled the Rico dataset, one of the largest repositories of mobile app GUI designs (9.7k Android apps with 72k unique GUI screens) by capturing screenshots, view hierarchies and user interactions via crowdsourcing and automated exploration. After recording

[1]https://github.com/M3SOulu/WinGUICrawler

a crowd worker's usage of the app, an automated agent retraced the interactions and continued exploring programmatically to discover new states. Bunian et al. [5] manually selected high-quality screens from the Rico dataset to which they annexed additional Android and iOS application screens. Furthermore, to ensure correct labeling, they implemented a crowdsourcing strategy. Crowd workers were trained and provided with clear instructions on how to produce bounding box annotations for screens. Zhang et al. [6] compiled a dataset of 77k GUI screens (from 4,068 iOS apps) by employing a two-step crowdsourcing approach. Firstly, crowd workers manually traversed apps while collecting screens and metadata. Secondly, crowd workers drew bounding boxes and classified GUI elements.

Automated annotation can be leveraged to obtain large amounts of labeled data. Nonetheless, it comes at the cost of quality, as the labels can prove less accurate than those of human annotators. Humans possess highly accurate internal recognition and learning mechanisms that outperform computer vision applications in a large class of problems [13]. Liu et al. [7] automated the annotation of mobile GUI elements and computed annotations for the screens from the Rico dataset. Their approach consisted in a deep classification comprised of both functional and structural semantics, done via code-based classification (GUI elements and text buttons) and Convolutional Neural Networks (icons). Moran et al. [8] used a completely automated approach and created an exploration engine that navigated a target app in Depth-First-Search (DFS) manner and extracted annotations using a GUI automation framework. Chen et al. [9] also adopted a similar approach, with the key difference lying in a weight strategy. During traversal actions probable to lead to more unique data were prioritised.

A project that resembles Win GUI Crawler is openui-dataset [14], which uses automated GUI exploration (with TESTAR tool [15]) for extracting screenshots and JSON files that describe GUI elements, their location and other properties. However, when attempting to use TESTAR tool for this purpose, we initially encountered technical difficulties. Later with TESTAR authors we found workarounds. Regardless, TESTAR is another well suited tool for GUI element extraction.

Synthetic data generation has also been proposed. Beltramelli et al. [10] created a small dataset by utilising a stochastic user interface generator to synthesize images and code.

Manual collection/annotation is the simplest approach. Hu et al. [11] selected a subset of Rico screens to which they annexed additional images taken from app stores, subsequently reclassifying and annotating the elements manually. Rahmadi et al. [12] collected a small dataset by manually acquiring GUI component images from CSS framework kitchen sink pages. The manual approach has poor scalability and is not well suited to efficiently collecting a large-scale database.

III. Methodology

In this section, we describe the methodology followed to extract screenshots and metadata of GUI elements. For this we designed Win GUI Crawler, which is built on top of Microsoft's Windows Application Driver (WinAppDriver) [16], to extract metadata from each screen and perform actions. WinAppDriver is a service to support Selenium-like UI Test Automation on Windows applications. It is derived from Selenium WebDriver and supports Universal Windows Platform (UWP), Windows Forms (WinForms), Windows Presentation Foundation (WPF) and Classic Windows (Win32) apps on Windows 10. In the remainder of this section, we describe how Win GUI Crawler collects GUI element images and filters irrelevant elements.

A. Crawler

The main objective of Win GUI Crawler is to automatically traverse a Windows application and collect image screenshots and related metadata. Crawlers can take either a depth first or breadth first approach. Depth first search is more appropriate for Windows applications since traversal happens only in one direction as there is not always an action that permits to reach the previous state. Each time the search reaches a dead end the application is restarted. In a breadth first approach, it would need to be restarted for each new screen, causing much longer traversal times. The traversal is recorded within a tree structure where nodes correspond to application screens and edges correspond to actions that the crawler takes to navigate between screens.

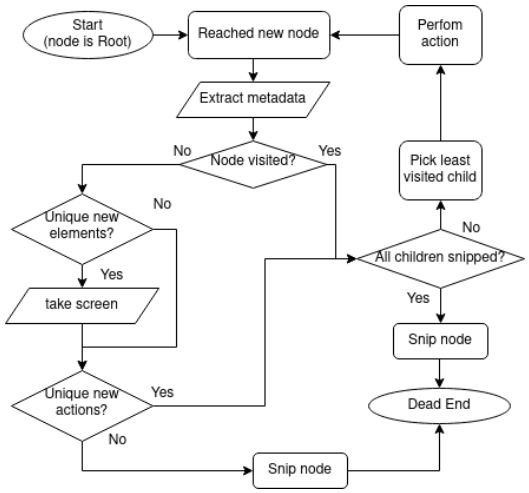


Figure 1: Flowchart of tree traversal loop

The traversal algorithm is shown in Fig. 1. Starting from the root node, which corresponds to the main application screen, the crawler will progressively move through the application and update the tree. The crawler keeps moving

down the tree until it reaches a dead end, this will denote one pass. The metadata that WinAppDriver extracts from each screen contains its GUI hierarchy and information on each element. There is no specific metadata tag to uniquely identify GUI elements. Thus, we use the XPath expression representing the element in the XML metadata of the screen containing it to identify the same elements across different screens. If two elements share the same XPath expression (containing the type and name of each element in the path), it means that they are the same GUI element.

At the beginning of the loop, metadata is extracted from the application screen. If the current node hasn't been previously visited and contains new unique elements that haven't been seen during the pass, then a screen image is taken. If the screen doesn't contain any new actions, then the crawler reaches a dead end and the node is snipped.

A dead end is also reached if all current node's children are snipped, meaning there are no more paths to traverse downstream.

If there are child nodes that have not been snipped, then one of them is picked as the next node. To ensure that traversal happens in a uniformly distributed manner, when picking the next node priority is given to nodes that have a lower visitation counter (which is incremented each time the crawler goes through a node). This type of distributed strategy is advantageous because it compels the crawler to gather more unique screens during the first iterations. Additionally, it prevents the crawler from getting stuck due to unforeseen behaviour in parts of the tree.

To reach the next node, the crawler clicks the GUI element. This loop is repeated until all nodes have been visited and the application has been fully explored (from the crawlers point of view). As more passes lead to more redundant data, a more optimal choice is to set a finite number of passes the crawler should complete. The crawler can run a finite number of passes or until the whole application is explored.

B. Filtering to improve bounding boxes

Bounding box labels that are extracted from the metadata might correspond to elements that are not visible in the image and thus result in inaccurate annotations. A filtering scheme is applied to improve accuracy, transform the metadata into clean annotations and crop GUI element images. The filter algorithm re-traverses the recorded interaction tree and uses information extracted from the metadata to remove inaccurate bounding boxes. Furthermore, it detects duplicate screens by comparing GUI element metadata of each screen.

Firstly, the elements that have the "IsOffscreen" attribute set to True are removed, and elements that have a very small area (less than 5 pixels in height and width).

The bounding box filter algorithm detects when two elements overlap and decides if and which ones to remove. To avoid detecting false overlaps or cases where the overlap area is too small to be noticeable, two elements are defined as overlapping if:

$$\frac{A \cap B}{min(A, B)} > OTh \tag{1}$$

A and B are the two bounding box areas and $OTh \in [0, 1]$ is an arbitrary threshold, which defines how much area overlap is acceptable (optimally the value should be close to 0 to only permit small overlap). After detecting all the overlapping element pairs, the filter decides which ones are acceptable. Elements can overlap and still be both visible, this happens when they are in a child/parent relationship or also if they are direct siblings of the same parent, these cases are deemed acceptable by the filtering algorithm and should not produce wrong labels.

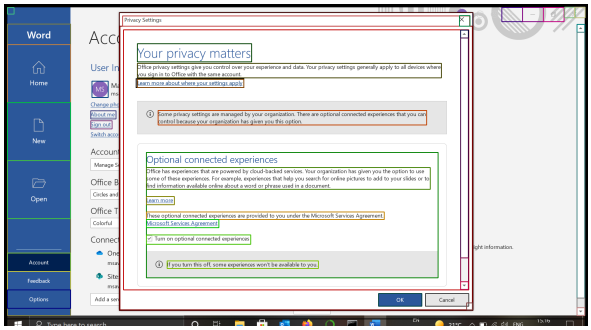
The unacceptable pairs are further analysed by exploiting the traversal order information. If one of the overlapping elements is not present in the previous screen, but present in the current one, it is kept and the other is removed. This way priority is given to newer elements that in most cases visually obstruct older ones. There can be cases where this is not true, but the element removed has already been correctly annotated previously (since it was present in the previous screen) and has at least one correct label in the collected data. If both elements have the same age, priority is given to certain types of elements that are more likely to be visible. For example, "Button" elements are more probable to have visual content than "Pane" elements that are containers. Since each application has their own peculiarities, the filtering cannot result in 100% accuracy. Nonetheless, it can be useful to assist data collection considering that the elements kept by the filter are much more likely to have an accurate label.

In Fig. 2 an example of the output of the bounding box filter is shown, as can be seen, the non visible bounding boxes are suppressed in the final result. This algorithm also cleans up the metadata and provides more accessible data by transforming it into a JSON file containing relevant attributes and binary outcome of the filter. Furthermore, individual cropped images of each GUI element are saved for each screen along with a file listing containing information on each element.

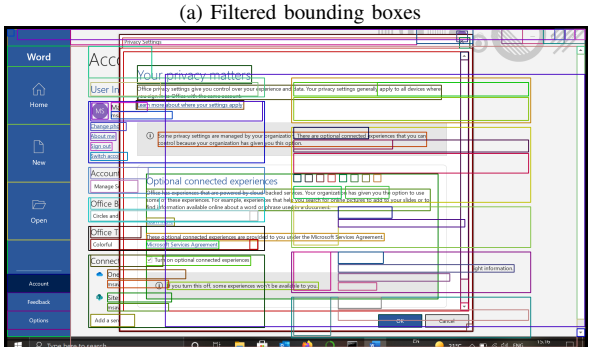
IV. Discussion

A. Evaluated Applications

The tool was tested on several commonly used Windows 10 applications. Since the tool is still a prototype the testing was limited to exploring a small set of applications to debug, improve the tool and investigate potential benefits and limitations. An extensive validation will be performed when the tool becomes more mature and ready for data collection. The applications tested were: Calculator, MS Paint, MS Word, Teamviewer, XnConvert, Virtualbox, Adobe Acrobat Reader, GIMP.



(a) Filtered bounding boxes



(b) Unfiltered bounding boxes

Figure 2: Filter output example of Microsoft Word screen with bounding boxes

Except for GIMP and Adobe Reader, the other applications were successfully explored by the crawler that managed to extract metadata for each element on screen. In the case of GIMP, as can be seen in Fig. 3, the only metadata accessible relates to the window, its content cannot be accessed. This is due to GIMP being designed with the GTK framework. In the case of Adobe Reader, as can be seen in Fig. 4, a subset of GUI element metadata is inaccessible, due to part of the user interface being designed with a HTML-based framework.

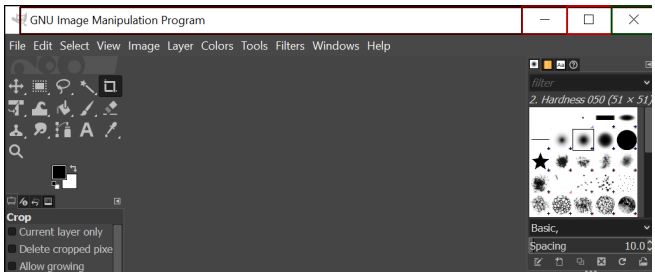


Figure 3: GIMP screenshot with bounding boxes

It was also observed that in case of dynamic elements that alter the state of the application (e.g. logout buttons), the crawler is cut off from part of the application after clicking them. Another challenge that makes it difficult to extract high-quality labels for every GUI element, is the fact that the coordinates in the metadata do not always

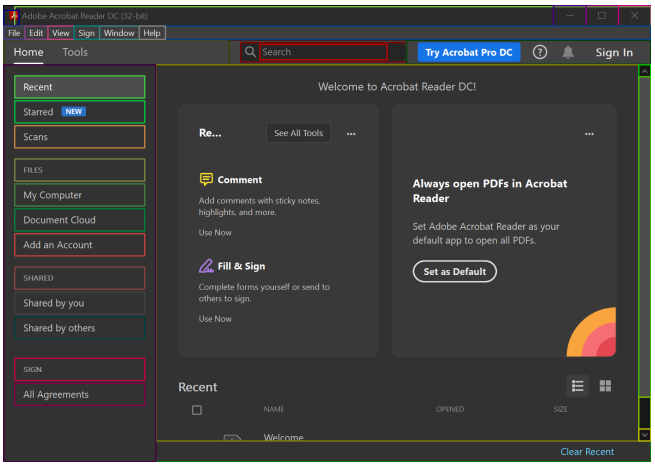
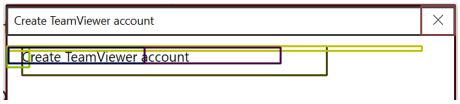


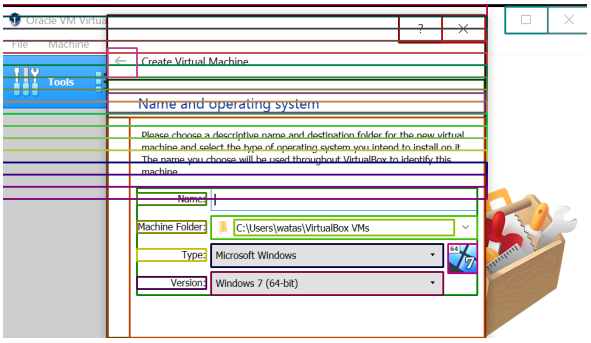
Figure 4: Adobe Acrobat Reader screenshot with bounding boxes

correspond to a bounding box that fits the element well. They may also correspond to non-visible elements that cannot be distinguished from visible ones from the metadata alone. Some examples of this type of behaviour are shown in Fig. 5.

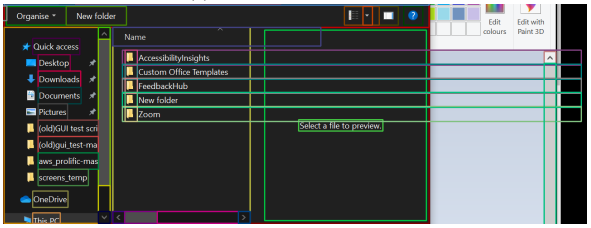
(a) Bad fit, bounding boxes too small



(b) Non-visible elements



(c) Non-visible elements



(d) Bad fit, bounding boxes too wide

Figure 5: Examples of inaccurate bounding boxes obtained from metadata

B. Data collection and benefits

The main purpose of this tool is to enable large-scale data collection. At its current state, the crawler can be used to traverse applications and extract preliminary annotations. The advantage in having an automated approach is that large amounts of data can be collected by employing less resources, with the drawback being that the resulting annotations are noisy. A way to overcome this is to have human annotators inspect the labels and modify them if necessary, thus producing higher quality annotations. To render this process more efficient, the elements singled out by the filtering algorithm should be given more attention as they are much more probable to correspond to inaccurate labels.

In addition to data collection, another practical application was identified, the crawler could aid existing template matching automation methods by automatically providing cropped images of the GUI elements. Automated cropping of elements can significantly reduce set-up times and make automation workflows more efficient compared to tedious manual cropping.

C. Limitations

There are several limitations to the tool. First, it is incompatible with non-native Win32 API applications (e.g. GTK or HTML-based) where access to metadata is limited. This is due to WinAppDriver not supporting these frameworks.

Second, there is no access to metadata from GUI elements that are linked to a different Windows process from the main application process, and that do not belong to the main application window. This limitation is caused by the fact that WinAppDriver attaches the WebDriver interface to a specific Windows process and window.

Third, interacting with dynamic GUI elements can result in the crawler being cut-off from some traversal paths, as it cannot reach nodes that are not present in the new application state.

Fourth, the crawler is limited to clicking only. Performing additional actions such as editing fields and inputting user credentials would take further development.

D. Future work

The tool presented in this paper is a first prototype and its further development will be the direction of future efforts. For the sake of improving the crawler range, the inclusion of more sophisticated actions will be implemented. Additionally, to avoid having the crawler cut-off from part of the application due to dynamic elements, a strategy will be devised that allows the crawler to be unsusceptible to state changes. To further refine the filtering algorithm, a pixel based analysis will be added. A more accurate filtering would significantly reduce the amount of elements that need to be inspected by human annotators. Once the tool has been developed to sufficient maturity, it will be utilised to collect a large-scale database on which an initial validation will be performed. Subsequently, a fine-combed crowdsourced annotation step will be carried out to reduce the labeling noise.

V. Conclusion

Win GUI Crawler was designed in order to overcome the challenge of non-robust GUI identification, which limits the applicability of automatic GUI testing. Machine vision based GUI testing could greatly improve robustness given that a large amount of data is available for training, which Win GUI Crawler can supply. The tool performed well with native Win 32 API applications, it was able to traverse the applications and extract annotations for each GUI element present. A filtering algorithm was also implemented to clean up noisy annotations by removing non-visible elements. The limitations that were reported are incompatibility of non-native Win 32 API applications, dynamic traversal issues and limited actions. Future work will include improving the tool by implementing more sophisticated filtering and traversal. When Win GUI Crawler reaches sufficient maturity, it will be employed in large-scale data collection.

References

[1] M. Nass, E. Alégroth, and R. Feldt, "Why many challenges with gui test automation (will) remain," *Information and Software Technology*, vol. 138, p. 106625, 2021.

[2] A. I. Khan and S. Al-Habsi, "Machine learning in computer vision," *Procedia Computer Science*, vol. 167, pp. 1444–1451, 2020.

[3] T. Chakraborti, V. Isahagian, R. Khalaf, Y. Khazaeni, V. Muthusamy, Y. Rizk, and M. Unuvar, "From robotic process automation to intelligent process automation," in *International Conference on Business Process Management*. Springer, 2020, pp. 215–228.

[4] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017, pp. 845–854.

[5] S. Bunian, K. Li, C. Jemmali, C. Harteveld, Y. Fu, and M. S. Seif El-Nasr, "Vins: Visual search for mobile user interface design," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–14.

[6] X. Zhang, L. de Greef, A. Swearngin, S. White, K. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach *et al.*, "Screen recognition: Creating accessibility metadata for mobile applications from pixels," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–15.

[7] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning design semantics for mobile apps," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 569–579.

[8] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 196–221, 2018.

[9] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 665–676.

[10] T. Beltramelli, "pix2code: Generating code from a graphical user interface screenshot," in *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, 2018, pp. 1–6.

[11] R. Hu, M. Chen, L. Cai, and W. Chen, "Detection and segmentation of graphical elements on guis for mobile apps based on deep learning," in *International Conference on Mobile Computing, Applications, and Services*. Springer, 2020, pp. 187–197.

[12] A. A. Rahmadi and A. Sudaryanto, "Visual recognition of graphical user interface components using deep learning technique," *Jurnal Ilmu Komputer dan Informasi*, vol. 13, no. 1, pp. 35–45, 2020.

[13] W. J. Scheirer, S. E. Anthony, K. Nakayama, and D. D. Cox, "Perceptual annotation: Measuring human vision to improve computer vision," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 8, pp. 1679–1686, 2014.

[14] H. Terho, "openuidataset," accessed: 2021-06-02. [Online]. Available: https://github.com/openuidata/openuidataset/

[15] T. E. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders, "testar–scriptless testing through graphical user interface," *Software Testing, Verification and Reliability*, vol. 31, no. 3, p. e1771, 2021.

[16] Microsoft, "Windows application driver," accessed: 2021-06-13. [Online]. Available: https://github.com/microsoft/WinAppDriver