

Where is that button again?! – Towards a universal GUI Search Engine

Sven Hertling¹, Markus Schröder¹, Christian Jilek¹ and Andreas Dengel^{1,2}

¹German Research Center for Artificial Intelligence (DFKI) GmbH, Trippstadter Straße 122, 67663 Kaiserslautern, Germany

²Knowledge-Based Systems Group, Department of Computer Science, University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
{sven.hertling, markus.schroeder, christian.jilek, andreas.dengel}@dfki.de

Keywords: information retrieval, GUI automation, accessibility interface

Abstract: In feature-rich software a wide range of functionality is spread across various menus, dialog windows, tool bars etc. Remembering where to find each feature is usually very hard, especially if it is not regularly used. We therefore provide a GUI search engine which is universally applicable to a large number of applications. Besides giving an overview of related approaches, we describe three major problems we had to solve, which are analyzing the GUI, understanding the users' query and executing a suitable solution to find a desired UI element. Based on a user study we evaluated our approach and showed that it is particularly useful if a not regularly used feature is searched for. We already identified much potential for further applications based on our approach.

1 INTRODUCTION

“Where is that button again?!” – this and other complaints are very common among users of feature-rich software. Typically in such software a high amount of features is spread across various menus, dialog windows, tool bars etc. in the graphical user interface (GUI). Remembering where to find each feature is usually very hard especially if it is not regularly used. The GUI, consisting of *graphical elements* like buttons, checkboxes, input fields etc., provides some textual clues of the functionality it offers. For example, a button's label reveals its utility and an additional tooltip may give detailed information about its usage. Depending on the implementation there may be many or just few of such hints. In order to find a certain software function, the user has to browse its graphical front-end as well as read and understand the textual representations. (Cockburn and Gutwin, 2009) show that novice users need more time to retrieve elements within hierarchical menus. Using software having such complex graphical interfaces can lead to despair. One approach to help expert users is CommandMaps (Scarr et al., 2012). Like the name suggests, it unfolds several menus in order to cover the whole screen. Users may then find the demanded menu items with the help of their spatial memory.

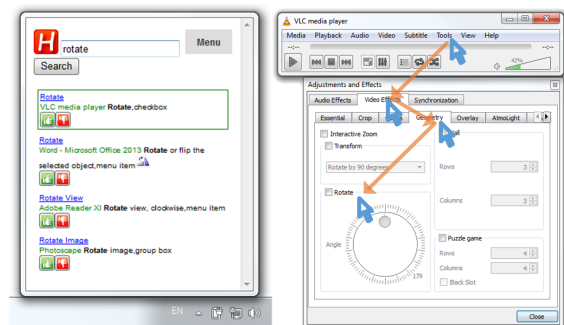


Figure 1: A user wants to rotate an upside down video. Our search engine retrieves a list of graphical elements by the given search term “rotate” (left-hand side). By selecting the first entry, the system performs necessary steps like mouse and keyboard interactions automatically to get the element visible. Therefore it traverses the appropriate menu item as well as several tab controls to finally present the *Rotate* feature (right).

In this paper, we concentrate on helping novice users finding the corresponding software features. This is done by a search engine for widgets within the GUI. After the user clicks on a search result, the appropriate mouse and keyboard actions are performed to make the element visible on screen.

To offer such a support, an approach first has to

grasp information and structure of the user interface. Since a graphical interface is rendered for humans, an obvious solution could be based on image processing (Yeh et al., 2009). However, in order to also extract the texts of the graphical elements, OCR-based approaches have to be applied. Beside the error rate, they also lack the ability to get hidden information like texts in tooltips.

Even if this is solved, we are still facing the challenges of understanding which feature the user is currently looking for (handled by (Adar et al., 2014)) and how to guide them there. Concerning the latter, an obvious solution would be to provide a textual explanation what has to be done in order to navigate to the element in question. For even higher convenience we offer a fully automated execution of the necessary steps. Figure 1 shows an example scenario of our application. The user searches for a software functionality called “rotate”. As a result the search engine provides several user interface elements. They are rendered like typical search results but also contain images and more detailed descriptions of the UI element. The right hand side shows the corresponding application. All necessary steps like mouse and keyboard interactions are performed automatically.

For the proposed approach we identify three main problems: The need for a sufficiently detailed GUI representation, a possibility to derive what the user is currently looking for and a method for searching the GUI model for it. Finally, the system has to perform the necessary steps to reach the desired graphical element. Additionally, we also focus on covering a wide range of software, not depending on any specific implementation details. Therefore 16 of the most frequently used software application are tested with our approach.

This paper is structured as follows: The next section contains an overview of related work. Next, we present our method in more detail according to the various subtasks it solves in order to implement a universal GUI search engine (Section 3). Section 4 is about a user study that was conducted to evaluate our approach. In Section 5 we conclude this paper and give an outlook on possible future work.

2 RELATED WORK

In the past, several approaches of analyzing and providing assistance in the GUI have been proposed. We therefore concentrate on four topics: GUI representation & testing, mapping user queries to commands, GUI assistance and product-specific GUI search engines.

2.1 GUI Representation & Testing

In the area of GUI testing, which verifies specifications in graphical front-ends, there is a need for generating GUI models motivated by model-based testing. As a consequence there are already solutions regarding the extraction of information and structure from the graphical user interface. Writing GUI test cases is an expensive task making their automation a worthwhile goal. For that reason (Memon et al., 2003) show an approach for reverse engineering graphical user interfaces which is called GUI Ripping. Memon et al. use the Windows API to crawl the complete GUI information. In (Memon, 2007) the extracted information is used to build an event-flow model of the GUI for testing purposes.

(Aho et al., 2013) and (Aho et al., 2014) transferred GUI testing to the industrial domain. They introduce the platform-independent Murphy tool set for extracting state models of GUI applications. (Grilo et al., 2010) cope with incomplete models and add an additional manual step to complete and validate them.

There is also a .NET-based framework to automate rich client applications which can be based on Win32, WinForms, WPF, Silverlight and SWT platforms (TestStack, 2016). It hides some of the complexity of the used frameworks like Microsoft UI Automation and windows messages. Commercial variants for GUI testing are Ranorex ((Ranorex GmbH, 2016) and (Dubey and Shiwani, 2014)) as well as TestComplete (SmartBear Software, 2016).

All previously described approaches use APIs for user interface accessibility like UI Automation. But there are also others that rely only on the graphical representation of UI widgets. Sikuli (Yeh et al., 2009), for example, retrieves elements by screenshots. If the corresponding position is found, the tool will execute mouse and keyboard events. Moreover they provide a visual scripting API to easily create GUI automation tasks. Another approach called Prefab (Dixon and Fogarty, 2010) reverse engineers the user interface on a pixel-based level. In (Dixon et al., 2011) they extended their solution to also model hierarchical structures of complex widgets.

2.2 Mapping User Queries to Commands

In some cases the user describes their tasks in a different vocabulary than the actual applications domain language. For finding these mappings, (Fourney et al., 2011) introduce query-feature graphs (QF-graphs). Search queries and system features are both represented as vertices. Each edge symbolizes a map-

ping between them. QF-graphs are constructed using search query logs, search engine results, web page content, and localization data from interactive systems.

In addition, the CommandSpace system (Adar et al., 2014) finds these matches based on a large corpus of Web documents about the application. They use deep learning techniques to create them and evaluated their approach on a single application, Adobe Photoshop.

ShowMeHow (Ramesh et al., 2011) translates user interface instructions between similar applications. Since software rapidly changes, tutorials are often out of date or not available at all. Thus, the goal is to locate commands in one application using the interface language of a similar application.

2.3 GUI Assistance

Regarding assistance in the graphical user interface there exist several approaches: *CommandMaps* (Scarr et al., 2012; Scarr et al., 2014) is a command selection interface which makes use of the user’s spatial memory to faster interact with UI elements. They demonstrated that their approach is significantly faster than Ribbon and menu interfaces for experienced users. Another strategy seeks for preventing false feedforward in menus (Lafreniere et al., 2015) and propose a task-centric interface (Lafreniere et al., 2014) which shows a textual workflow with actionable commands included. Both approaches share the intention of getting the user to the desired graphical element faster.

PixelTone (Laput et al., 2013) provides a multimodal interface for photo editing. With their approach it is possible to select a region of an image and label it, e.g. “This is a shirt”. *GEKA* (Hendy et al., 2010) created a graphically enhanced keyboard accelerator which enables users to call software functions by entering commands like in a typical command line interface.

(Ekstrand et al., 2011) developed an approach for finding better software learning resources (documentation, tutorials and the like), in particular by exploiting the user’s current context in a specific application.

The *Process Guide* (IMC Information Multimedia Communication AG, 2016) by IMC directs the user through a graphical front-end by giving practical usage hints in a context sidebar. These hints are generated manually for each workflow.

2.4 Product-specific GUI Search Engines

The following approaches already solved the problem of finding graphical elements in their application by providing an embedded search engine. They are, however, product-specific and do not work in other applications. One example is the settings search of the Chrome web browser (Google Inc., 2016). All content of the settings pages can be searched for, even deeply nested pages (e.g. search for “cookies”). In Microsoft Visual Studio, *QuickLaunch* (Microsoft Corporation, 2016b; Naboulsi, 2012) is embedded for fast retrieval of commonly used functionality. Another similar search engine called *Search Commands* (Microsoft Corporation, 2011) is an add-in for the Microsoft Office product suite. The successor *Tell Me* (Microsoft Corporation, 2016a) is available since Microsoft Office 2016. A more general approach is *Help Menus* in Apple’s OS X operating system (Miser, 2008, p. 62), (Ness, 2011, p. 13), (Pogue, 2012, p. 67). It allows for searching menu items in the menu bar in all OS X applications. Unfortunately, all other graphical elements are not searchable. This is also the case for the settings search in Windows 10 and Apple iOS. Cortana (Microsoft Corporation, 2016c) can search for application names but not texts or names within them.

3 APPROACH

Helping the user in finding the graphical elements they are looking for, several problems have to be tackled. As a basis we first have to grasp the structure and information of the GUI. Then, the appropriate functionality has to be derived from terms entered by the user when conducting a search. Additionally, the system has to perform all mouse and keyboard interactions to navigate to the UI element in question.

3.1 Software Basis

As a basis for the approach, a good sampling of most often used software is necessary. Therefore we analyzed websites which offer commonly used software for downloading. A collection of 35 websites is reduced to the 11 most accessed ones based on web traffic analyzers like Alexa, Compete, Semrush(de), Semrush(en) and PageRank. The resulting set of download websites is

- amazon.com
- cnet.com
- softonic.com
- chip.de
- heise.de
- filehippo.com
- amazon.de
- sourceforge.net
- zdnet.com
- computerbild.de
- netzwelt.de

We chose software which is among the top ten in the ranking of at least two sites. After the removal of programs like “Flash Player” (web-plugin) and “Minecraft” (game) which are irrelevant for our use case the resulting set is:

- Word 2013 (Trial)
- PowerPoint 2013 (Trial)
- Open Office Calc
- VLC media player
- Google Chrome
- 7-Zip
- Skype
- AntiVir 2014 - Avira
- Adobe Reader
- Excel 2013 (Trial)
- Open Office Writer
- Open Office Impress
- CCleaner
- Mozilla Firefox
- WinRAR
- PhotoScape
- NortonInternetSecurity
- avast! Free Antivirus

Unfortunately, “Norton Internet Security” as well as “avast! Free Antivirus” (2 of 18 applications or 11.11 % of the sample set) do neither support the UI Automation framework nor the MSAA framework. The remaining 16 applications are used in this paper to build software models and conduct the evaluation.

Figure 2 shows a more detailed analysis of the download count of cnet.com (depicted as a log-log plot). We see that it best fits a log-normal distribution with the parameters of $x_{min} = 554$, $\mu = 3.656$ and $\sigma = 3.499$. The best fitting parameters for a power law distribution are $x_{min} = 57033$ and $\alpha = 1.813$. They are all calculated using the `powerlaw` package (Gillespie et al., 2015). The log-normal distribution also applies for the `chip.de` website. Unfortunately all other pages don’t provide absolute download counts. This analysis validates that there are only few software tools which are often downloaded and a large number of those that are demanded less frequently. Thus, with an appropriate selection of applications, a huge amount of users could benefit from an approach like ours.

3.2 GUI Analysis

Analyzing the graphical user interface is a difficult task. OCR or other image-based approaches did not fulfill our needs due to the rudimentary amount of information and error-proneness. We therefore utilize accessibility interfaces since they provide precise and detailed information about an application’s content and structure. They are typically used in screen readers which help visually impaired people navigating their applications. In particular, we use “Microsoft

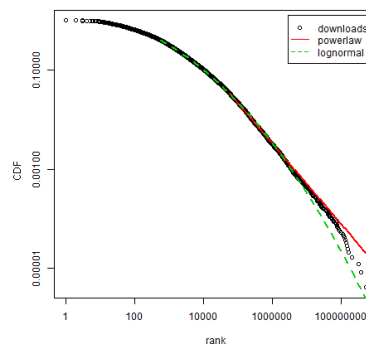


Figure 2: Log-log plot of downloaded software from cnet.com

Active Accessibility” (MSAA) (Microsoft Corporation, 2000) and its successor “Microsoft UI Automation” (UIA) (Microsoft Corporation, 2016d) which are two prominent accessibility APIs. Based on this powerful technology we are able to exploit a lot of software applications without further ado.

The graphical user interface of Microsoft Windows is represented as a tree providing a hierarchical structured list of graphical elements. The APIs allow access to these elements’ structure (i.e. child and parent relations) and properties (i.e. name, position, size, etc). However, they do not offer functionality which extracts a complete software model. In particular, they only allow for *crawling* the currently visible UI tree. Consequently, parts of the graphical front-end have to be made visible in order to capture them. To make further elements visible one has to interact with another element, for example a menu is opened if the corresponding menu item is clicked. That is why we decided to observe GUI interactions (i.e. mouse clicks or keyboard shortcuts) to detect appearing elements and thereby grasp the complete software. As a benefit we recognize which element is responsible to make other elements visible. This insight is later useful to execute appropriate interactions automatically to navigate to a certain element.

We consider two observable interaction sources: On the one hand we can observe users solving daily tasks using the graphical front-end. We can thereby understand GUI usage behavior, but fail in obtaining a complete software model which would be preferable. To solve this we implemented an algorithm called *click monkey* that automatically explores a given application. This is done by clicking all graphical elements in a depth-limited traversal strategy ensuring a high UI element coverage of the software. We implemented an observer which is able to capture UI interactions regardless whether they are performed by the

user or the click monkey. In each record we consider (a) the causing action, (b) an application software’s GUI tree and (c) the graphical element corresponding to the current cursor location. By collecting a lot of records over time we thus receive a detailed interaction log serving as a basis for an expressive software model.

Gathering GUI data is only a preliminary step. Next, we have to understand what the user is currently looking for by interpreting their search query and matching it to known UI elements.

3.3 Interpreting User Queries

To solve the problem of having to browse software manually, we have to construct a searchable GUI model. We therefore make use of information retrieval technologies indexing all observed graphical elements, especially their names and metadata. Using this textual information we try to match given user queries to suitable UI elements. A document in the search index consists of a field which contains concatenated values of the properties Name, LegacyIAccessibleDescription, HelpText and LocalizedControlType of the corresponding element. We call this field “UI text”. Additionally, information like ProductName, CompanyName, FileDescription of the corresponding software is added.

Three approaches for searching UI elements are developed. The baseline approach analyzes the “UI text” field based on the StandardAnalyzer of Lucene which generates tokens based on a Unicode Text Segmentation algorithm (specified in Unicode Standard Annex #29 (Davis and Iancu, 2016)) and lowercases all tokens.

The second approach called “language” further analyzes the “UI text” field resulting in three additional fields in the index:

- (1) standard tokenizer, HTMLStripCharFilter (because HTML is contained in UI elements’ text or description) and the following filters: lowercase, synonyms (English: WordNet (Fellbaum, 1998; Miller, 1995), German: OpenThesaurus (Naber, 2004; Naber, 2005)), ASCIIFolding (Lucene), stopword removal (based on list of elasticsearch (Elasticsearch reference, 2016)), language dependent filter like GermanNormalizationFilter (Lucene), stemming (based on (Savoy, 2006) for French, Portuguese, German and Hungarian languages and (Porter, 1980) for English),
- (2) standard tokenizer and the following filters: lowercase, NGramTokenFilter (min:1, max:20),
- (3) same as (2) but with EdgeNGramTokenFilter.

The query is a boolean *should* clause query of “baseline” and three “language” fields. This implies that terms which do not fully match a term in the index are still counted using for example their stemmed version (and thus increase recall). Nevertheless we also consider precision because more matching fields will increase the score.

The “context” approach uses the result of the previous one and re-ranks the documents considering the overall desktop context. In particular, search results corresponding to already opened (i.e. currently in use) applications receive a higher score.

For all approaches TF-IDF is used as a similarity metric. (Okapi BM25 has also been tested but was omitted due to not having a considerable difference compared to TF-IDF).

In the current implementation the mapping between user queries and application vocabulary is based on term matching and synonyms. It would also be possible to include other approaches like QF-graphs (Fourney et al., 2011) or CommandSpace (Adar et al., 2014) as additional resources for synonyms.

The information retrieval system we described so far still lacks the ability of executing mouse and keyboard interactions to get the graphical element visible. That is why the next section covers another software model which is able to derive such information.

3.4 Solution Execution

As already mentioned, one has to interact with an element to make further elements visible. This relation plays a key role in the following graph-based software model. By using a graph of connected UI elements, a navigation is reduced to a shortest path between two elements. For explaining a step by step solution a path is appropriate: Ideally, it starts with an already visible element and describes what has to be done in order to open them successively. The final element will be the one the user is looking for.

In particular, our model is a directed, weighted graph having graphical element vertices and interaction edges. In short, an edge encodes what element causes another one’s occurrence. Because of this property we call this model a *Show New Element Graph* (SNEG). The edge weight is used to store the reliability that the interaction will correctly lead to the promised graphical element. (This is induced by the observation process not being completely reliable.) Based on evidences the model can learn which paths have proven to be correct.

Models like the one just described explain how to reach elements. Users still have to read and under-

stand the explanation and navigate through the GUI on their own. Although it helps them to learn about the software’s structure, we additionally want to provide an automation for convenience. Thus, we implemented an algorithm which executes interactions automatically. To do this, it uses the path acquired from the previous model, which encodes two types of information: the expected graphical elements and the interactions which have to be executed on them. Accordingly, the algorithm has to master two subtasks: a reliable recovery of graphical elements and the execution of the interactions.

UI Element Recovering For recovering a UI element, an evaluation with the 16 selected software tools was conducted (see Section *Software Basis*). All UI elements and all their properties from each start screen are recorded. In total there are 714 elements which should clearly be differentiated. Table 1 shows those properties that best distinguish these elements.

Each property is tested separately and the count of resulting equivalence classes is used to find out which feature combination is best suited. Most of them are volatile and thus change over time like the `BoundingBox`, `ProcessId` etc. The following set can be used to rediscover UI elements:

- Name
- AutomationId
- ClassName
- ControlType
- LegacyIAccessibleChildId
- LegacyIAccessibleDescription
- HelpText
- AccessKey
- LocalizedControlType
- LegacyIAccessibleRole

In GUI testing also UI elements are identified. The identification characteristics are determined at test recording time. Since the playback is executed in the very same environment, this is sufficient. For the proposed approach the environment can change drastically and it is therefore necessary to collect suitable properties for identification.

It is possible that the values of one or more of the properties stated above can change over time. Each element can thus also have dynamic properties which can not be used for identification. Dynamic properties are automatically detected when UI elements with the same `RuntimeId` are compared.

4 EVALUATION

To evaluate our approach we conducted a user study with 10 participants (5 male, 5 female, average age 35.2, std. deviation 16.3). First, the participants were

Table 1: All UIA properties sorted by their resulting equivalence classes (table shows only properties with 10 or more classes). In total there are 714 elements which should be distinguished.

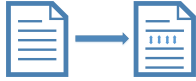

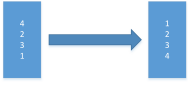

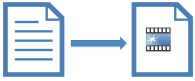







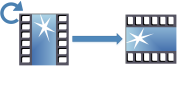












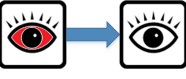

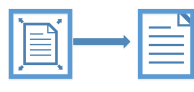



Property name	ID	Count of equivalence classes
RuntimeId	30000	583
BoundingBox	30001	570
Name	30005	299
LegacyIAccessibleName	30092	
ProviderDescription	30107	156
NativeWindowHandle	30020	139
HelpText	30013	100
LegacyIAccessibleHelp	30097	
AutomationId	30011	92
AccessKey	30007	66
LegacyIAccessible-KeyboardShortcut	30098	
ClassName	30012	62
LegacyIAccessible-Description	30094	45
LegacyIAccessibleState	30096	35
LegacyIAccessibleRole	30095	32
LocalizedControlType	30004	31
LegacyIAccessibleValue	30093	28
ControlType	30003	27
ValueValue	30045	25
ProcessId	30002	15
LegacyIAccessible-DefaultAction	30100	13
LegacyIAccessibleChildId	30091	10

asked about their computer usage ranging from “almost no use” (1) to “very frequent use” (4) which resulted in a mean of 3.3 with a standard deviation of 0.67. In order to classify the participants in two classes, they were asked for each application if they frequently use it (expert), or if they rarely use it (novice).

In order to test our GUI search engine under real conditions we extracted 16 most frequently used software applications from the most commonly visited download sites (see Section *Software Basis*). For each of them a fictional task and a corresponding pictogram is defined which illustrates the task at hand without influencing the user’s search terms. They are listed in Table 2.

The evaluation setup for each participant was realized as follows: In a random sequence a participant processed one task after the other. For each task four subtasks had to be accomplished:

Table 2: Table of programs, tasks and corresponding pictograms used in the user study

 → 	 → 	 → 	 → 	 → 
Word 2013 Change paragraph	Exel 2013 Sort	PowerPoint 2013 Insert video	OpenOffice Writer Double underline	OpenOffice Calc Insert frame
 → 	 → 		 → 	 → 
OpenOffice Impress Change orientation	VLC media player Rotate video	CCleaner Update	Google Chrome Zoom	Mozilla Firefox Print page
 → 	 → 	 → 	 → 	 → 
7Zip File manager Delete file	WinRAR Rename file	Skype Change sound	Photoscape Red eye effect	Adobe Reader Fullscreen
 → 				
Avira Free Antivirus Password				

(Subtask 1) Initially, the task's pictogram as well as the related program's icon and name were displayed. Without influencing the user's search terms, the pictogram helps them to derive the envisioned task. This method wants to simulate an intention to perform a specific task in a certain application.

(Subtask 2) Afterwards, the participant had to formulate a textual query describing the task. Using our proposed information retrieval system a list of retrieved graphical element was shown to the participant. This randomized list consists of the top 15 graphical elements of the three introduced search strategies (pooling strategy of TREC evaluation (Buckley and Voorhees, 2005)). Based on the participant's relevance feedback we later evaluated the strategies' precision and recall with the trec_eval evaluation package (Buckley, 2004).

(Subtask 3) Then, the participant is supposed to search for the graphical element on their own in the graphical front-end. We encouraged them to use any form of external assistance (e.g. internet, manual, etc.). While doing so, time and click count is captured. The user had to indicate if the element was found.

(Subtask 4) Finally, our GUI Search Engine must be used to find the correct graphical element. Again, time and click count was recorded while using the

tool. In particular, this also includes the formulation of the query, the selection of an element and the execution of the tool. Once more, the user has to state if the tool found the graphical element in question.

Finally each participant had to fill out a user experience questionnaire (UEQ) (Laugwitz et al., 2008).

Based on the user study's acquired data, we analyzed the usefulness of our approach and the usage experience.

Users can mainly benefit in two ways: The GUI search engine may find the UI elements faster and it may retrieve results when the user has already given up manual search. In particular, a confusion matrix describing the dependencies is shown in Table 4(b). It distinguishes whether the tool and/or the user found the desired UI element or not. In case both found it, the matrix further shows in how many cases using the tool was faster. We differentiate 160 observations – 10 participants performing 16 tasks each with another application.

The measured number of clicks and the time to complete the tasks are visualized in Figure 3. We ordered the tasks according to their difficulty, i.e. the average user time to fulfill the task. We assume that easier tasks are solved faster while more difficult ones need more time. Figure 4(a) especially focuses on those cases in which the tool was helpful. It shows

that the search engine helped users coping with the most difficult and moderately difficult tasks. The green bars indicate the fraction of participants per task for whom the tool was particularly helpful, since they could either find elements faster (green) or they were able to find the elements after manual search failed (dark green). If the tool was not helpful, we differentiate the following three cases: The user was faster when not using the tool (orange), tasks could not be completed at all – whether the tool was used or not – (red) and only participants not using the tool could find the element (dark red). Unexpectedly, the tool could not efficiently support participants in Tasks 5, 4 and 1. In case of Task 5 the GUI search engine could not find any graphical element at all due to an incorrect observed software model. Although the tool found the elements in Tasks 4 and 1, it took more time than a manual search. Regardless of the necessary time, our GUI search engine could find the desired elements in 55.6% of all 160 cases.

In order to evaluate the users' perceptions of the search engine, we additionally conducted a user experience questionnaire (UEQ) (Laugwitz et al., 2008). Please note that there might be a slight bias since the participants are personally known to the authors. Based on the collected data it derives the following six factors: attractiveness, perspicuity, efficiency, dependability, stimulation, and novelty. The results are depicted in Figure 5 and are discussed in the following. The excellent average value in *novelty* shows that the participants consider the approach as creative, innovative and leading-edge. Although there are product-specific search engines, presented in related work section, they seem to be unknown to the users. Ergonomic quality aspects like efficiency, perspicuity and dependability are rated above average, whereas the latter two achieve a slightly higher score. We assume that the slightly lower rating of efficiency is due to the technical immaturity of the current prototype. However, there is still potential for improvement.

Another minor criterion of the evaluation is the performance of the graphical element retrieval system which lists suitable UI elements based on a user's search term. Since this is information retrieval technology, we will investigate its recall and precision.

Based on relevance feedback in the user study, we created recall-precision-curves for each of the three search strategies that are depicted in Figure 6. We see that more analytical effort results in higher recall and precision values.

In the next section we conclude this paper and give an outlook on possible future work.

5 CONCLUSION AND OUTLOOK

In this paper we presented the first universal GUI search engine, i.e. one that works for a wide range of applications.

For 16 of 18 very frequently used programs we showed that our approach supports users in easily finding software functionality, especially if it is a feature that is not used regularly. We provide the automatic execution of required steps like mouse and keyboard actions. To obtain the necessary software models we implemented the click monkey, a mechanism to automatically explore the user interfaces of given applications. Concerning our information retrieval system we implemented a first version and identified some potential for improvement. We also showed that rather minor enhancements already resulted in noticeable positive effects.

In the two remaining cases our solution could not be used, since the applications did not support the Windows accessibility API. Using the Java Access Bridge in these cases seems promising and will be implemented in future versions.

We also intend to build an in-application tutoring system based on our approach. Additionally, the possibility of deriving usability implications from the given software models could be investigated. Porting our approach to mobile platforms like Android is another future task.

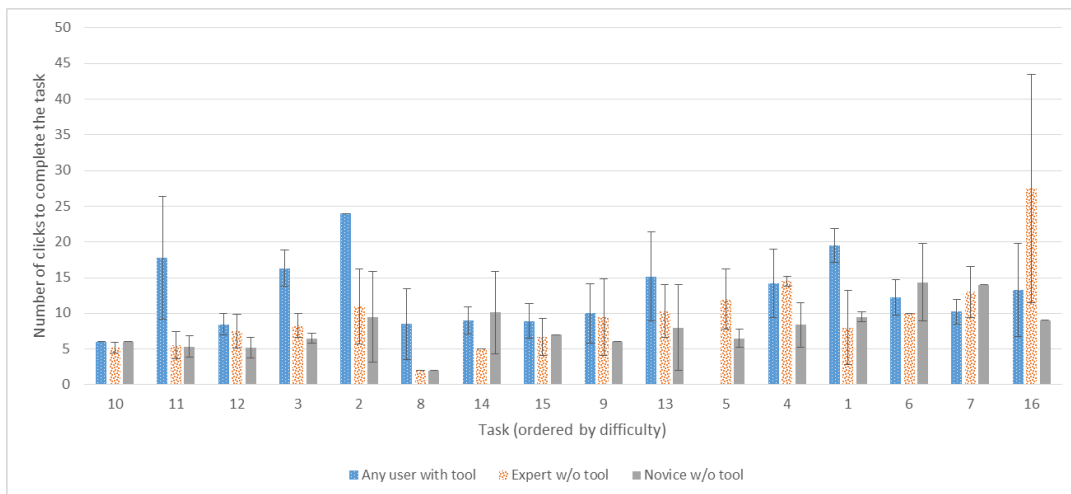
The GUI usage behavior can further be exploited in order to share expert knowledge about software which is contained therein.

ACKNOWLEDGEMENTS

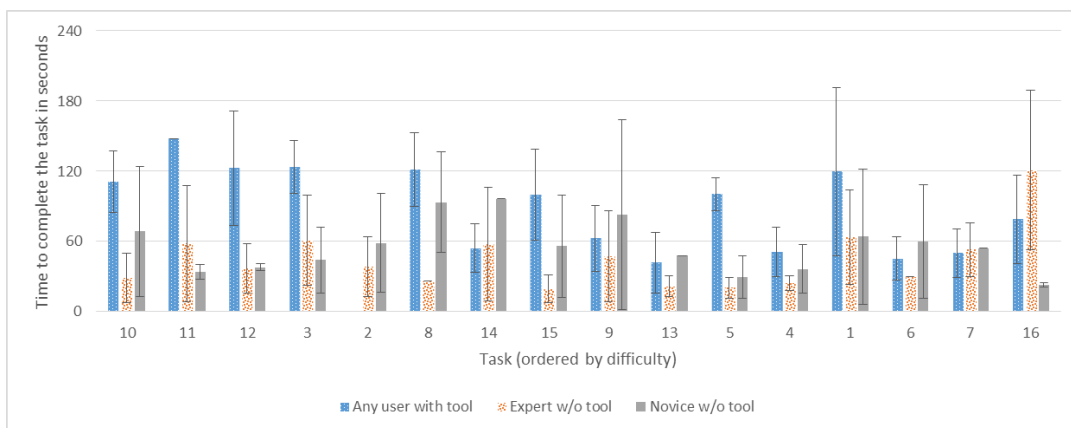
This research was funded in part by the German Federal Ministry of Education and Research under grant no. 01IS12050 (project SuGraBo). The responsibility for this publication lies with the authors.

REFERENCES

- Adar, E., Dontcheva, M., and Laput, G. (2014). Commandspace: modeling the relationships between tasks, descriptions and features. In *Proceedings of the 27th annual ACM symposium on User interface software and technology*, pages 167–176. ACM.
- Aho, P., Suarez, M., Kanstren, T., and Memon, A. (2013). Industrial adoption of automatically extracted GUI models for testing. In *Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modelling*. Springer Inc.



(a) Average number of clicks to complete each task differentiating between tool, expert and novice users.



(b) Average time in seconds to complete each task differentiating between tool, expert and novice users.

Figure 3: Analysis of the average click count and time to successfully execute a task (ordered by the tasks' difficulties)

Aho, P., Suarez, M., Kanstren, T., and Memon, A. (2014). Murphy tools: Utilizing extracted GUI models for industrial software testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 343–348.

Buckley, C. (2004). Trec eval ir evaluation package. http://trec.nist.gov/trec_eval/. accessed on 2016-10-25.

Buckley, C. and Voorhees, E. M. (2005). Retrieval system evaluation. *TREC: Experiment and evaluation in information retrieval*, pages 53–75.

Cockburn, A. and Gutwin, C. (2009). A predictive model of human performance with scrolling and hierarchical lists. *Human Computer Interaction*, 24(3):273–314.

Davis, M. and Iancu, L. (2016). Unicode text segmentation. <http://unicode.org/reports/tr29/>. accessed on 2016-10-25.

Dixon, M. and Fogarty, J. (2010). Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the*

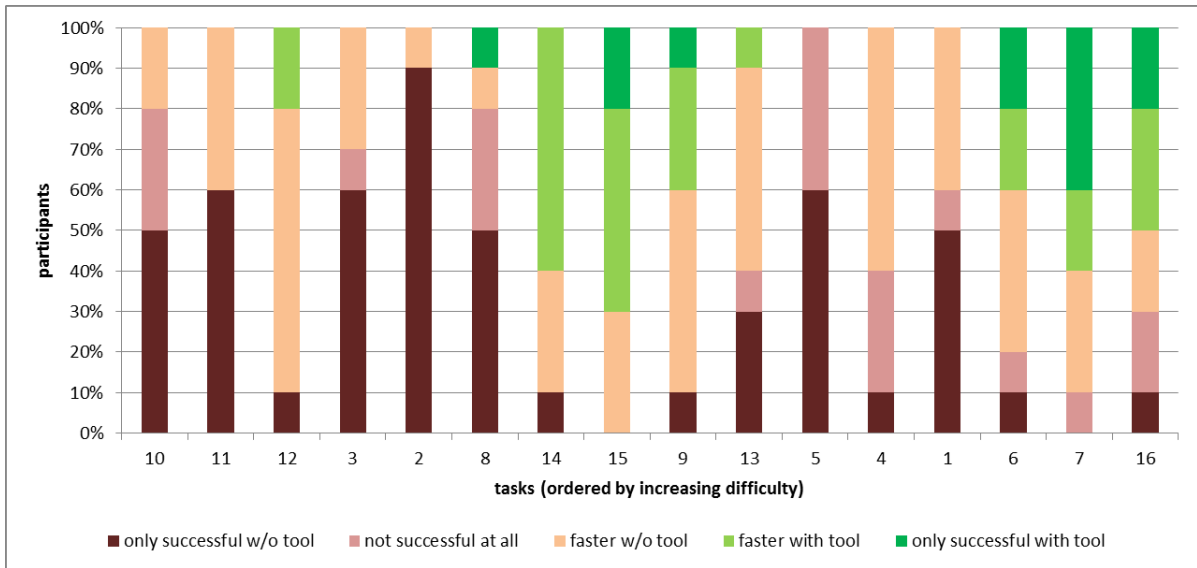
SIGCHI Conference on Human Factors in Computing Systems, pages 1525–1534. ACM.

Dixon, M., Leventhal, D., and Fogarty, J. (2011). Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 969–978. ACM.

Dubey, N. and Shiwani, M. S. (2014). Studying and comparing automated testing tools; ranorex and test-complete. *International Journal Of Engineering And Computer Science*, 3:5916–5923.

Ekstrand, M., Li, W., Grossman, T., Matejka, J., and Fitzmaurice, G. (2011). Searching for software learning resources using application context. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 195–204. ACM.

Elasticsearch reference (2016). Stop token filter. <http://www.elasticsearch.org/guide/en/elasticsearch/reference/current/analysis-stop-tokenfilter.html>. accessed on 2016-10-25.



(a) The colored bars indicate the fraction of participants per task for five aspects: The tool helped the users because they could not find the element on their own (dark green) or because using the tool was faster (green). In cases the tool was not helpful we differentiate three cases: Not using the tool was faster (orange), element could not be found at all (whether the tool was used or not, red) and only participants not using the tool found the element (dark red).

		User with tool			Total
		successful	failed	failed	
User w/o tool	successful	33,1%	15,0% (tool faster)	31,9%	80%
	failed		7,5%	12,5%	20%
Total		55,6%		44,4%	100%

(b) Confusion Matrix depicting in how many cases the tool and/or user found the UI element in question. Please note that we split up the case of both groups being successful: we additionally record whether using the tool was faster or not.

Figure 4: Analysis of the approach's usefulness

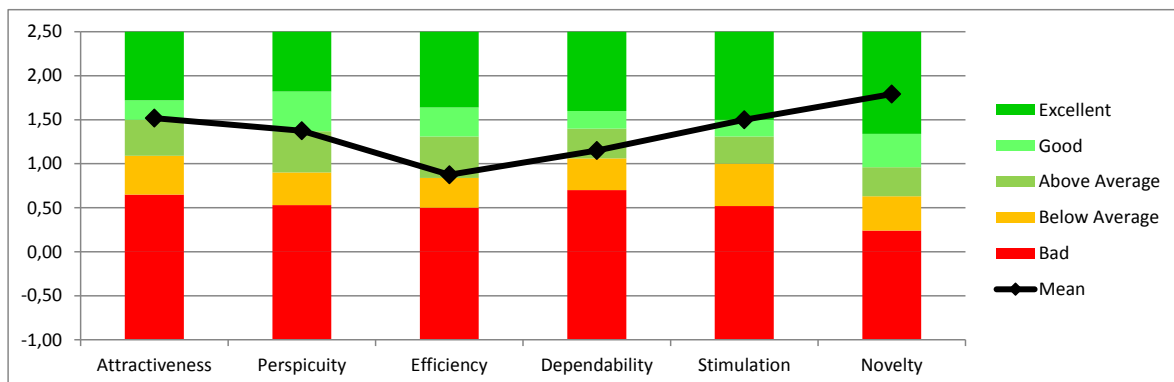


Figure 5: User experience questionnaire results

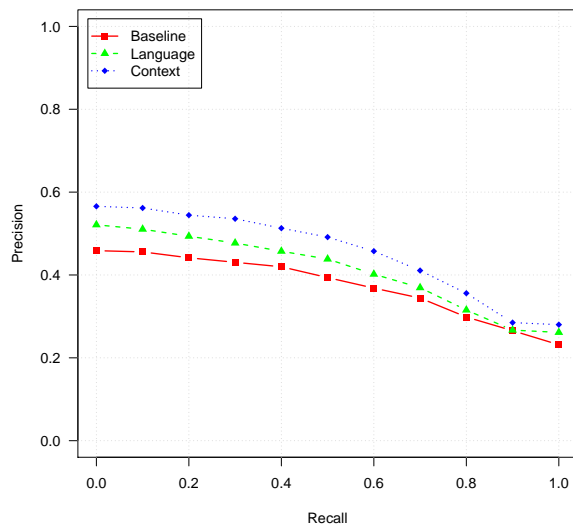


Figure 6: Recall/precision curves of the three search strategies based on relevance feedback in the user study

- Fellbaum, C. (1998). *WordNet – An Electronic Lexical Database*. MIT Press.
- Fourney, A., Mann, R., and Terry, M. (2011). Query-feature graphs: bridging user vocabulary and system functionality. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 207–216. ACM.
- Gillespie, C. S. et al. (2015). Fitting heavy tailed distributions: The powerlaw package. *Journal of Statistical Software*, 64(i02).
- Google Inc. (2016). Chrome-browser features. <https://www.google.com/chrome/browser/features.html>. accessed on 2016-10-25.
- Grilo, A. M., Paiva, A. C., and Faria, J. P. (2010). Reverse engineering of gui models for testing. In *5th Iberian Conference on Information Systems and Technologies*, pages 1–6. IEEE.
- Hendy, J., Booth, K. S., and McGrenere, J. (2010). Graphically enhanced keyboard accelerators for guis. In *Proceedings of Graphics Interface 2010*, pages 3–10. Canadian Information Processing Society.
- IMC Information Multimedia Communication AG (2016). Electronic performance support system imc process guide. <https://www.im-c.de/en/learning-technologies/performance-support>. accessed on 2016-10-25.
- Lafreniere, B., Bunt, A., and Terry, M. (2014). Task-centric interfaces for feature-rich software. In *Proceedings of the 26th Australian Computer-Human Interaction Conference on Designing Futures: the Future of Design*, pages 49–58. ACM.
- Lafreniere, B., Chilana, P. K., Fourney, A., and Terry, M. A. (2015). These aren't the commands you're looking for: Addressing false feedforward in feature-rich software. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 619–628. ACM.

- Laput, G. P., Dontcheva, M., Wilensky, G., Chang, W., Agarwala, A., Linder, J., and Adar, E. (2013). Pixel-tone: a multimodal interface for image editing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2185–2194. ACM.
- Laugwitz, B., Held, T., and Schrepp, M. (2008). *Construction and evaluation of a user experience questionnaire*. Springer.
- Memon, A. M. (2007). An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157.
- Memon, A. M., Banerjee, I., and Nagarajan, A. (2003). Gui ripping: Reverse engineering of graphical user interfaces for testing. In *WCRE*, volume 3, page 260.
- Microsoft Corporation (2000). Microsoft active accessibility: Architecture. <http://msdn.microsoft.com/en-us/library/ms971310.aspx>. accessed on 2016-10-25.
- Microsoft Corporation (2011). Office labs: Search commands. <http://www.microsoft.com/en-us/download/details.aspx?id=28559>. accessed on 2016-10-25.
- Microsoft Corporation (2016a). Do things quickly with tell me. <https://support.office.com/en-us/article/Do-things-quickly-with-Tell-Me-f20d2198-17b8-4b09-a3e5-007a337f1e4e>. accessed on 2016-10-25.
- Microsoft Corporation (2016b). Quick launch. <http://msdn.microsoft.com/en-us/library/hh417697.aspx>. accessed on 2016-10-25.
- Microsoft Corporation (2016c). What is cortana? <https://support.microsoft.com/en-us/help/17214/windows-10-what-is>. accessed on 2016-10-25.
- Microsoft Corporation (2016d). Windows automation api. [http://msdn.microsoft.com/en-us/library/windows/desktop/ff486375\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ff486375(v=vs.85).aspx). accessed on 2016-10-25.
- Miller, G. A. (1995). Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.
- Miser, B. (2008). *Mac OS X Leopard in Depth*. Que Publishing.
- Naber, D. (2004). Openthesaurus: Building a thesaurus with a web community. *Citeseer*, 3:2005.
- Naber, D. (2005). OpenThesaurus: ein offenes deutsches Wortnetz. *Beiträge zur GLDV-Tagung*, pages 422–433.
- Naboulsi, Z. (2012). Visual studio 2012 new features: Quick launch. <http://blogs.msdn.com/b/zainnab/archive/2012/06/26/visual-studio-2012-new-features-quick-launch.aspx>. accessed on 2016-10-25.
- Ness, R. (2011). *Mac OS X Lion in Depth*. Que Publishing.
- Pogue, D. (2012). *OS X Mountain Lion: The Missing Manual*. O'Reilly Media.
- Porter, M. F. (1980). An algorithm for suffix stripping. *Program*, 14(3):130–137.
- Ramesh, V., Hsu, C., Agrawala, M., and Hartmann, B. (2011). Showmehow: translating user interface instructions between applications. In *Proceedings of the*

- 24th annual ACM symposium on User interface software and technology*, pages 127–134. ACM.
- Ranorex GmbH (2016). Ranorex homepage. <http://www.ranorex.com>. accessed on 2016-10-25.
- Savoy, J. (2006). Light stemming approaches for the french, portuguese, german and hungarian languages. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1031–1035. ACM.
- Scarr, J., Cockburn, A., Gutwin, C., and Bunt, A. (2012). Improving command selection with commandmaps. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 257–266. ACM.
- Scarr, J., Cockburn, A., Gutwin, C., Bunt, A., and Cechanowicz, J. E. (2014). The usability of commandmaps in realistic tasks. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2241–2250. ACM.
- SmartBear Software (2016). TestComplete homepage. <https://smartbear.com/product/testcomplete/overview/>. accessed on 2016-10-25.
- TestStack (2016). White. <https://github.com/TestStack/White>. accessed on 2016-10-25.
- Yeh, T., Chang, T.-H., and Miller, R. C. (2009). Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192. ACM.