Extracting Widget Descriptions from GUIs

Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro

Department of Informatics, Systems and Communications
University of Milano Bicocca
Milano, Italy
{becce,mariani,riganelli,santoro}@disco.unimib.it

Abstract. Graphical User Interfaces (GUIs) are typically designed to simplify data entering, data processing and visualization of results. However, GUIs can also be exploited for other purposes. For instance, automatic tools can analyze GUIs to retrieve information about the data that can be processed by an application. This information can serve many purposes such as ease application integration, augment test case generation, and support reverse engineering techniques.
In the last years, the scientific community provided an increasing attention to the automatic extraction of information from interfaces. For instance, in the domain of Web applications, learning techniques have been used to extract information from Web forms. The knowledge about the data that can be processed by an application is not only relevant for the Web, but it is also extremely useful to support the same techniques when applied to desktop applications.
In this paper we present a technique for the automatic extraction of descriptive information about the data that can be handled by widgets in GUI-based desktop applications. The technique is grounded on mature standards and best practices about the design of GUIs, and exploits the presence of textual descriptions in the GUIs to automatically obtain descriptive data for data widgets. The early empirical results with three desktop applications show that the presented algorithm can extract data with high precision and recall, and can be used to improve generation of GUI test cases.

Keywords: program analysis, graphical user interface, testing GUI applications

1 Introduction

A Graphical User Interface (GUI) can be a valuable source of information for understanding the features implemented by an application. For instance, a GUI typically includes a number of descriptive labels that specify the kind of data that an application processes; a GUI includes menus and buttons that are related to the features an application can execute; a GUI visualizes and processes data, such as descriptions of facts, names of places, and names of people. Unfortunately, the knowledge represented by the content of descriptive labels, menu and data is embedded into the widgets and it is hardly accessible by automatic systems.

In the last years, the scientific community provided an increasing attention to the extraction of information from interfaces with the objective of understanding the data and the features offered by an application under analysis. In particular, techniques for extracting data from Web interfaces, such as [13, 19, 10], early studied this problem limitably to Web forms with the objective of easing data integration of online databases.

Extracting information from interfaces is not only relevant when applied to Web applications, but also when applied to desktop applications. For example, a relevant limitation of automatic test case generation techniques for GUI-based desktop applications is the lack of mechanisms for the identification and generation of data values useful to produce interesting executions in the application under test [20]. Identifying the right data that can be entered in a GUI would overcome this limitation, increasing the effectiveness of test case generation. Similarly, the automatic extraction of the data and features available in a desktop application would enable the possibility to automatically check conformance with respect to requirements [17]. Many other areas could also benefit from the extraction of information from GUIs, for example reverse engineering, and tool integration.

In this paper, we present a technique for the automatic identification of descriptive information about the data that can be entered in data widgets of GUI-based desktop applications. The technique relies on well grounded and widely adopted standards and best practices about the design of GUIs, and exploits the presence of widgets that include textual descriptions to discover the right descriptors of data widgets. The early empirical results with three applications show that the presented algorithm is effective and has the potential of enabling the previously described researches. We also report early empirical results that show how generation of GUI test cases can benefit from our algorithm.

The paper is organized as follows. Section 2 describes the principles underlying the design of our technique. Section 3 presents the algorithm that we use to extract descriptors of data widgets. Section 4 presents early empirical results. Section 5 discusses related work. Finally, Section 6 provides concluding remarks.

2 Design Principles of GUIs

How to design easy-to-use and user-friendly interfaces has been a subject of studies from many years. Nowadays there are a number of mature standards, guidelines and practices that help developers designing good GUIs. Among the many standards, we recall the Java look and feel design guidelines [3], the ISO guidance and specifications [6], and the the laws of the Gestalt about the perception of the space [14]. Our technique exploits these standards and principles to correctly identify relations between widgets.

According to the taxonomy presented in [11], widgets can be classified in three groups: action widgets, static widgets, and data widgets. *Action widgets* are widgets that give access to program functions. A typical example of an action widget is a button. *Static widgets* are widgets used to increase the understand-

ability and usability of a GUI, but not for direct interaction. A typical example of a static widget is a label. *Data widgets* are widgets that display or accept data. A typical example of a data widget is a textarea.

In this work, we focus on static and data widgets. We further distinguish static widgets in descriptive widgets and container widgets. *Descriptive widgets* are static widgets that display textual information that help users understanding how to use action and data widgets (e.g., labels). *Container widgets* are static widgets used to group related widgets (e.g., panels and frames).

The key idea exploited in this paper is retrieving descriptions of data widgets by looking in the descriptive widgets. Thus, we designed an algorithm that can identify the descriptive widget associated with a data widget according to three basic principles about the design of GUIs: proximity, homogeneity, and closure (from the laws of the Gestalt). In the following, we describe how these principle affect the algorithm.

Proximity The law of proximity is based on the fact that people tend to logically group together objects that are displayed close each other. This principle is almost applied to the design of every GUI. For instance, the expected content of a data widget (e.g., a person name) is normally specified with a label (e.g., with the text "name") that is placed close to the data widget. In this work we focus on the left-to-right writing convention, we thus consider that labels, and more in general descriptive widgets, are placed at the left/top of the documented widget. The algorithm presented here can be easily adapted to other writing conventions.

Fig. 1. Search space for data widgets.

Our algorithm takes advantage of this convention by restricting the search space that it considers when looking for a descriptive widget associated with a data widget. In particular, when searching a descriptive widget that appropriately describes the content of a data widget the algorithm looks into the rectangular area delimited as follows: the bottom-right corner of the area coincides with the center of the data widget under consideration, and the top-left corner of the area coincides with the top-left corner of the current window. Any

descriptive widget which is entirely or partially displayed in that area is considered as a candidate for being associated with the data widget under analysis. More formally, if (x, y) are the cartesian coordinates of the center of a data widget of interest dw, and (x^*, y^*) are the cartesian coordinates of the upper left corner of a descriptive widget, only the descriptive widgets that satisfy the following relation are candidates for being associated with dw: $x \geq x^* \wedge y \geq y^*$.

Figure 1 shows an example. Any descriptive widget that intersects the dotted area is a candidate descriptor for the textarea.

Homogeneity The principle of homogeneity says that widgets should be distributed with regular patterns and possibly grouped according to their semantic. The regular distribution of descriptive and data widgets often implies that widgets are aligned horizontally and vertically (see for example the GUI in Figure 2). When widgets are dense the likelihood of associating a wrong descriptive widget to a data widget is quite high.

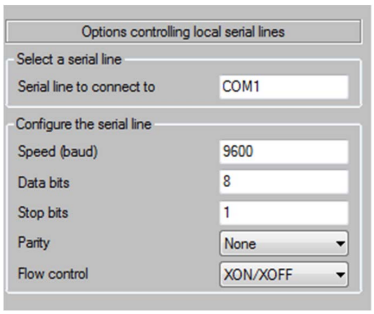

Fig. 2. Aligned widgets.

We took this issue into account when defining how to compute the distance between widgets, as illustrated in Figure 3. Each widget is associated with a representative point. The distance between two widgets is defined as the distance between their representative points. We choose representative points with the purpose to favor descriptive widgets that are aligned horizontally with the data widget under consideration, compared to widgets at different vertical positions. In particular, the representative point of the data widget is always its top-left corner. The representative point of any widget placed below or at the same level of the data widget is its top-right corner (the position of a widget is the position of its center). The representative point of any widget above the data widget is its bottom-left corner. Thus, if there are descriptive widgets placed both above the data widget and aligned with the data widget, the aligned widget is favored because its representative point is closer to the data widget (nevertheless it is still possible to associate a widget placed above the data widget with the data

widget, if the aligned widget is far enough). This strategy is able to well handle the many situations where many widgets in a same window are aligned, like the case shown in Figure 2.

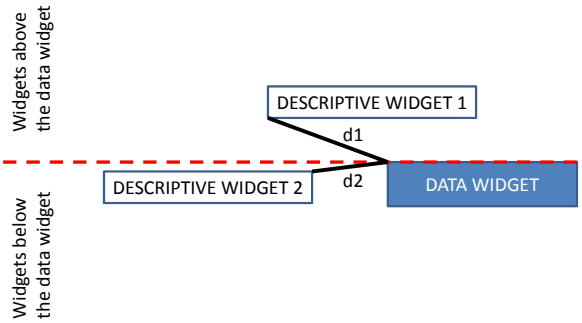

Fig. 3. Distance between widgets.

Closure The principle of closure says that persons tend to see complete figures even when part of the information is missing. This principle is typically exploited in the design of windows that contain many widgets. In fact, it is common practice to use container widgets for separating into multiple groups the widgets in a same window. Groups include semantically correlated widgets. For instance, two containers can separate the widgets designed for entering personal information from the widgets designed for entering credit card data, in a window dedicated to the handling of a payment process.

Our algorithm takes into account this practice implementing the possibility to limit the search space of a data widget to the widgets included in its same container.

3 Extraction of Widget Descriptions

In this section we present the algorithm for guessing associations between descriptive widgets and data widgets. The behavior of the algorithm is influenced by multiple parameters, which are empirically investigated in Section 4.

The types of widgets supported by the algorithm are specified in Table 1 column `Widgets`. Note that the column includes not only data widgets, but also container widgets. Container widgets are included because the label used to describe a container can be often used to describe the data widgets in the container as well.

The widgets listed in column `Widgets` cover the majority of data widgets that are used in practice. Each type of widget, specified in column `Widget Type`, can

Widget Type	Widgets	Descriptor Widgets
Text	TextField, FormattedTextField, PasswordField, TextArea, EditorPane, TextPane	Label, CheckBox, RadioButton
Multichoice	CheckBox, RadioButton, ToggleButton, ComboBox, List	Label
Container	Panel, ScrollPane, TabbedPane, SplitPane	Label

Table 1. Association between widgets and their descriptors.

be associated with a different set of descriptors. Column `Descriptor Widgets` indicate the descriptors that the algorithm considers for each type of widget. For instance, our algorithm uses Labels, CheckBoxes and RadioButtons as possible descriptors for TextField; while it uses only Labels as descriptors for ComboBoxes. These associations are defined according to common practices in design of GUIs[1].

Associations can be discovered statically (i.e., by analyzing the source code) or dynamically (i.e., by analyzing the windows of the application at run-time). Since static analysis techniques can only be applied if specific development strategies are adopted, such as the use of Rapid Application Development environments [15], our algorithm discovers associations dynamically. In particular, it extracts the data necessary for the analysis from the widgets displayed by the application at run-time.

Algorithm 1 reports the pseudocode of the main algorithm, while Algorithm 2 reports the pseudocode of the `isCandidate()` auxiliary function. The pseudocode is a simplified version of the implemented algorithm that does not consider performance optimizations.

Algorithm 1 takes as input a data widget, the set of widgets in the same window and four parameters, and returns the descriptor widget that passes the selection criteria implemented by the algorithm and is closest to the input data widget. Algorithm 2 implements all the checks that a widget has to pass to be considered as candidate descriptor for another widget. These checks include the ones inherited from the Proximity principle (see lines 6-8 in Algorithm 2); the Homogeneity principle (see computation of the distance at line 16 in Algorithm 1); the Closure principle, which is mapped into the local search strategy described afterward; and associations in Table 1, which are exploited for the check at line 2 in Algorithm 2. Table 2 specifies the values that can be assigned to parameters taken as input by Algorithm 1. We now describe in detail the role of the parameters.

Since a GUI can include noisy descriptor widgets, that is descriptor widgets with no information that are incorrectly used to layout the widgets in a window (e.g., empty labels invisible to users), the algorithm includes a noise reduction step. Noise reduction can be done at two different times: before starting the analysis of a window (opt_{noise} =Begin), or while analyzing a window (opt_{noise} =Incremental). When opt_{noise} =Begin, the algorithm removes every

[1] note that for widgets like checkboxes the items that can be checked are reported at the right of small rectangles, but the items that should be checked are anyway described with a label placed in the area at the top-left of the checkboxes.

Algorithm 1 guessDescription()

Require: $dw = (x, y, width, height)$ a data-widget with upper left corner at (x, y), width $width$ and height $height$
Require: $W = \{w_1, \ldots, w_n\}$ a window with n widgets, where $dw \in W$
Require: opt_{noise}, opt_{search}, $opt_{visible}$, $opt_{hierarchical}$
Ensure: returns either $w \in W$ descriptive widget associated with dw or $\varnothing$

```
1:
2: pos_dw = ((x+width)/2, (y+height)/2)
3: min = MAXINT
4: bestWidget = ∅
5:
6: if opt_noise = Begin then
7:    W = removeNoisyWidgets(W)
8: end if
9:
10: for each i=1 to |W| do
11:    if not isCandidate(dw, W, w_i, opt_noise, opt_search, opt_visible) then
12:       continue //skip to next widget
13:    end if
14:
15:    //select the closest descriptive widget
16:    dist = computeDistance(w_i, dw)
17:    if dist < min then
18:       min = dist
19:       bestWidget = w_i
20:    end if
21: end for
22:
23: if min=MAXINT then
24:    if opt_search = Local and opt_hierarchical then
25:       return guessDescription(container(dw), W, opt_noise, opt_search, opt_visible,
          opt_hierarchical)
26:    else
27:       return ∅
28:    end if
29: else
30:    return bestWidget
31: end if
```

useless descriptor widget from the set considered by the analysis and then proceeds normally with the updated set (see lines 6-8 in Algorithm 1). When opt_{noise} =Incremental, the algorithm removes useless descriptor widgets incrementally while considering them (see lines 10-12 in Algorithm 2).

The algorithm can look for descriptor widgets globally in the current window (opt_{search} =Global) or within the current container only, following the closure principle (opt_{search} =Local). If opt_{search} =Local, Algorithm 2 discards every widget that is not in the same container than in the data widget under consider-

Algorithm 2 isCandidate()

Require: $dw = (x, y, width, height)$ a data-widget with upper left corner at (x, y), width $width$ and height $height$
Require: $W = \{w_1, \ldots, w_n\}$ a window with n widgets, where $dw \in W$
Require: $w_i = (x_i, y_i, width_i, height_i) \in W$
Require: opt_{noise}, opt_{search}, $opt_{visible}$
Ensure: returns True if w is a candidate descriptor for dw, False otherwise

```
1:
2: if not compatible(type(w), type(dw)) then
3:    return False //skip widgets that cannot be associated with dw according to Table 1
4: end if
5:
6: if x < x_i or y < y_i then
7:    return False //skip widgets that are outside the interesting area of dw
8: end if
9:
10: if opt_noise = Incremental and noisy(w_i) then
11:    return False //incrementally ignore noisy widgets
12: end if
13:
14: if opt_search = Local and container(dw) ≠ container(w_i) then
15:    return False //ignore descriptive widgets in other containers
16: end if
17:
18: if opt_search = Global and opt_visible = VisibleOnly and not visible(w_i) then
19:    return False //skip widgets that are not visible
20: end if
21: return True
```

ation (see lines 14-16 in Algorithm 2). If opt_{search} =Local, when no descriptor is found for a data widget in a container, the algorithm can associate the descriptor of the container to the data widget ($opt_{hierarchical}$ =Yes), instead of using no descriptor ($opt_{hierarchical}$ =No). This case is covered by the recursive call at line 25 in Algorithm 1. If opt_{search} =Global, the algorithm can be further tuned to ignore widgets that are not visible to users ($opt_{visible}$ =VisibleOnly) or to also consider widgets invisible to users ($opt_{visible}$ =All). The check is implemented from line 18 to line 20 in Algorithm 2. A typical case influenced by this parameter is the analysis of a window with a ScrollPane that includes many elements, but only some of them are visualized at time.

4 Empirical Evaluation

The empirical evaluation presented in this section investigates the quality of the results produced by the algorithm presented in this paper, with particular emphasis on the tradeoffs between the different configurations. We also analyze

Parameters	**Values**	
opt_{noise}	Begin	Incremental
opt_{search}	Global	Local
Global Search Parameters	**Values**	
$opt_{visible}$	All	VisibleOnly
Local Search Parameters	**Values**	
$opt_{hierarchical}$	Yes	No

Table 2. Parameters.

the performance of the algorithm, and we report early results about the benefits introduced in the AutoBlackTest GUI testing technique by the presented algorithm.

Application	**Windows**	**Widgets**		**Containers**	
	Number	**avg**	**max**	**avg**	**max**
JPass	4	20.75	32	1.5	4
PDFSaM	7	23.63	32	3.13	5
jAOLT	12	52.58	169	4.42	12

Table 3. Case Studies

Case Studies In order to evaluate the technique presented in this paper we looked for applications from different domains with GUIs of increasing size and structure. Table 3 summarizes the applications we selected from Sourceforge.

JPass [4] is a personal password manager. PDFSaM [5] is an application for splitting and merging PDF files. jAOLT [2] is a desktop client for eBay. Column `Windows Number` indicates the number of analyzed windows. We measured the size of the analyzed windows reporting the average and maximum number of widgets per window (columns `Widgets avg` and `max` respectively). We counted both visible and invisible widgets. To approximatively derive a measure of the structure of the GUI of an application, we measured the average and maximum number of container classes per window (columns `Containers avg` and `max` respectively), assuming that more containers per window intuitively suggests that developers provided greater effort into suitably grouping widgets according to their semantics.

Configuration Name	opt_{search}	$opt_{visible}$	$opt_{hierarchical}$	opt_{noise}
GLOBAL(All) + Incremental	Global	All	-	Incremental
GLOBAL(All) + Begin	Global	All	-	Begin
GLOBAL(VisibleOnly) + Incremental	Global	VisibleOnly	-	Incremental
LOCAL(No) + Incremental	Local	-	No	Incremental
LOCAL(Yes) + Incremental	Local	-	Yes	Incremental

Table 4. Configurations

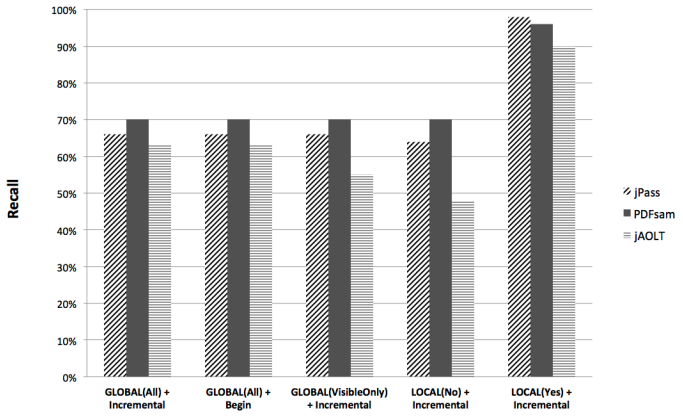

Fig. 4. Recall

Empirical Process In the validation, we studied the configurations reported in Table 4. Note that the option $opt_{noise} = Begin$ is studied only for the Global search strategy. We made this choice because it was clear already from the initial experiments that in the practice the Begin and Incremental strategies produce results with the same quality, but Incremental is faster. We thus kept $opt_{noise} = Incremental$ for the rest of the experiments.

We measure the quality of the results produced by the algorithm using the standard metrics of precision, recall and F-measure. Precision indicates the fraction of correct associations extracted by the algorithm with respect to the overall number of extracted associations. Recall indicates the fraction of correct associations extracted by the algorithm with respect to the overall number of associations that could be extracted from the applications. F-measure is an index that combines and balances precision and recall. Formally,

$$precision = \frac{CA}{WA + CA}, recall = \frac{CA}{TA}, F\text{-}measure = \frac{2 * precision * recall}{precision + recall} \quad (1)$$

where CA is the number of correct associations extracted by the algorithm, WA is the number of wrong associations extracted by the algorithm and TA is the total number of correct associations that the algorithm should have retrieved. The value of CA, WA and TA are measured by checking one by one each association retrieved by the algorithm and every widget in every window of the case studies. To mitigate the risk of computing imprecise data we repeated the counting multiple times.

We evaluated the performance of the technique by measuring the total time required for analyzing the GUI of the applications.

We finally integrated the algorithm in the AutoBlackTest test case generation technique [12]. AutoBlackTest generates GUI test cases randomly choosing

concrete input values from a pre-defined set of values. The integration of the algorithm presented in this paper augmented AutoBlackTest with the capability of selecting test inputs according to the kind of data widgets that must be filled in. We measure the benefit of the augmented approach measuring code coverage.

Effectiveness Figure 4 shows the results about recall. We can notice that every configuration based on a Global search performed similarly. The restriction of the search to visible widgets only causes a small reduction of recall, which means that the option causes the lost of some relevant associations. The local search with $opt_{hierarchical}$ =No surprisingly behaves worst than any global search. On the contrary, when $opt_{hierarchical}$ =Yes the recall raises to values between 90% and 100%. These results suggest that in the practice the labels associated with containers are frequently used to also describe the data expected by data widgets.

Figure 5 shows data about precision. We can notice that every configuration worked well, which means that regardless the selected configuration the algorithm seldom extract wrong associations. We can also notice that the local search worked slightly better than the global search. Intuitively this confirms that the closure principle is applied in the practice, it is thus better to restrict the search within containers, otherwise the risk of extracting wrong associations increases.

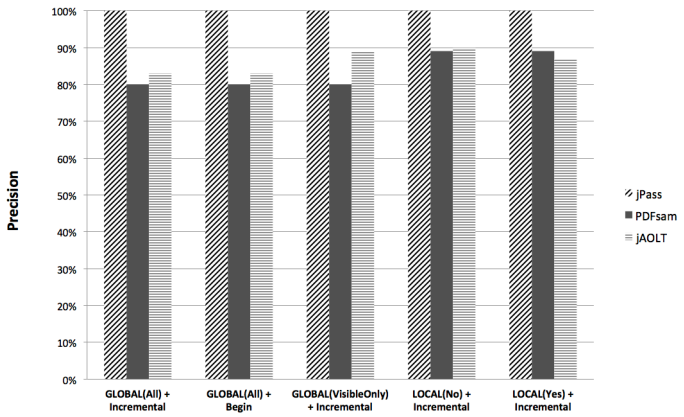

Fig. 5. Precision

Figure 6 shows data about F-measure. We can notice that global search and local search with $opt_{hierarchical}$ =No perform similarly. On the contrary the last configuration outperformed the others producing the best compromise in terms of precision and recall (note that F-measure varies between 89% and 99%). It is interesting to notice that local search with $opt_{hierarchical}$ =No is worse than global search, while the local search with $opt_{hierarchical}$ =Yes is better than

global search, regardless the amount of containers and structure in the interface. Considering the presence of containers is thus important only if considering also the labels associated with containers, even for interfaces with little structure.

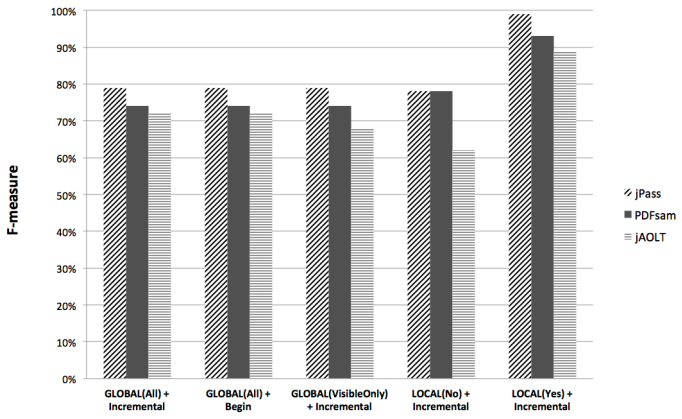

Fig. 6. F-measure

Performance To evaluate the performance of the configurations we measured the amount of time required to complete the analysis of the windows implemented in the three case studies that we selected. Figure 7 visualizes the performance of each configuration.

We can notice that the configuration that generated the best performance is also the slowest one, while the other configurations have similar performance. Since the time difference is small both in absolute and relative terms, unless performance is of crucial importance, the LOCAL(Yes) + Incremental is the configuration that should be selected. On the contrary, if performance is a crucial aspect, the fastest configuration that still provides good results is GLOBAL(All) + Incremental.

Augmenting AutoBlackTest The technique presented in this paper can be used to augment the capabilities of many techniques in many domains. Here we report early empirical data about the improvement that this technique produced in AutoBlackTest [12]. The study focuses on two applications that we already tested with AutoBlackTest and that we now tested gain with the augmented version of AutoBlackTest: PDFSam [5] and Buddi [1].

AutoBlackTest automatically generates GUI test cases that contain simple concrete values selected from a predefined data pool with many generic strings and numbers. This clearly limits the testing capability of AutoBlackTest. We

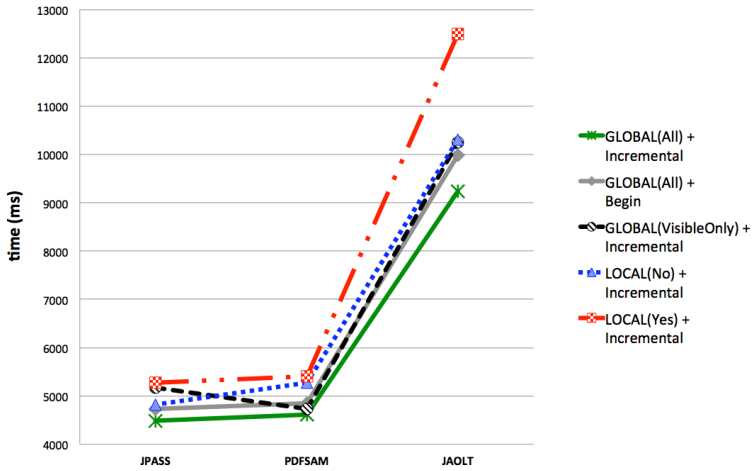

Fig. 7. Performance

extended AutoBlackTest with both the capability of associating descriptions to data widgets, according to the algorithm presented in this paper, and with the capability of using a datapool with concrete values organized according their kind. For instance, the datapool distinguishes dates, quantities, person names, and city names. The datapool is manually populated with concrete values following the boundary testing principle, that is it includes legal values, boundary values, illegal values and special values. As a result, the augmented AutoBlackTest can generate test cases that make a better use of data widgets by selecting proper concrete values from the datapool, exploiting the semantic information associated with the widgets.

The non-extended version of AutoBlackTest generated GUI test cases that cover 64% and 59% of the code in PDFSam and Buddi, respectively. The extended version of AutoBlackTest increased code coverage to 70%(+6%) and 64%(+5%), respectively. Considering that the computations implemented by these applications make a limited use of the data entered in data widgets (they mostly store and retrieve values from a database), the obtained increment is encouraging. In the future, we will study stronger ways of integrating this algorithm with test case generation techniques.

5 Related Work

Research specifically targeting the extraction of information from GUIs appeared in [13, 19, 10]. All these works addressed the integration of Web data sources. The contribution most relevant to our work is the one by Nguyen et al. [13], which uses Naive Bayes and Decision Trees classifiers for automatically associating labels with data widgets in Web forms. Our contribution complements this work

according to multiple aspects. First, the work by Nguyen et al. targets Web applications while we target desktop applications, which are designed partially following different principles. Second, the technique by Nguyen et al. learns how to retrieve associations from a training set. Unfortunately, training sets are notably hard to retrieve, especially for desktop applications. In addition, learning from a training set works properly only if it is large enough and well represents the GUIs that need to be analyzed. On the contrary, our approach does not rely on any training set, but it is defined according to standards and best practices about the design of GUIs. Thus, even if the effectiveness of our solution is in principle correlated to the quality of the interface under analysis, our algorithm is always applicable, and even with imperfect interfaces like the ones we analyzed, it provided high quality results.

Test data generation is an area that is gaining increasing attention and can be well targeted by our algorithm. The generation of test data for GUI test cases is a particularly hard problem where little automation is available. Nowadays the generation of test data for GUI test cases is a difficult and laborious process, in which test designers have to manually produce the inputs for data widgets. Automated GUI testing techniques either ignore generation of test data [16] or rely on fixed datapools of values [12, 20]. As a consequence many behaviors cannot be automatically tested.

In this paper we early investigated the use of our algorithm to improve generation of test data for GUI test cases. We integrated the algorithm in AutoBlackTest, but the algorithm can be potentially integrated in any other GUI test case generation technique. To the best of our knowledge the work presented in [9] is the only one that infers the values that can be entered in data widgets with the objective of using this information in the scope of testing, even if with a different purpose. In fact they used this mapping to assure that GUI test scripts could be reused after GUI modifications.

Other people addressed generation of test data for GUI test cases from specifications, such as augmented use case descriptions [8, 7] and enriched UML Activity Diagrams [18], but these descriptions are seldom available in the practice.

6 Conclusions

The GUI is a useful source of information that software analysis techniques should better exploit to increase their effectiveness.

In this paper we addressed the issue of automatically extracting the associations between descriptive and data widgets. We presented an algorithm that bases the extraction strategy on standards and common principles in the design of GUIs. Early empirical data collected with three case studies suggest that the algorithm can extract associations with high precision and recall.

The algorithm can be useful in many domains. In this paper we investigated the integration of the algorithm in the AutoBlackTest technique. Early results show that AutoBlackTest augmented with this algorithm produces GUI test cases that achieve greater coverage than the non-augmented version.

References

1. Buddi. http://buddi.digitalcave.ca/.
2. jAOLT. http://code.google.com/p/jaolt/.
3. Java look and feel design guidelines. http://java.sun.com/products/jlf/ed2/book/.
4. JPass. http://metis.freebase.hu/jpass.html.
5. PDFSAM. http://sourceforge.net/projects/pdfsam/.
6. *ISO 9241-12:1998 Ergonomic requirements for office work with visual display terminals (VDTs) - Part 12: Presentation of information.* 1998.
7. A. Bertolino and S. Gnesi. Use case-based testing of product lines. *SIGSOFT Softw. Eng. Notes*, 28:355–358, September 2003.
8. P. Fröhlich and J. Link. Automated test case generation from dynamic models. In *Proceedings of the European Conference on Object-Oriented Programming*, 2000.
9. C. Fu, M. Grechanik, and Q. Xie. Inferring types of references to gui objects in test scripts. In *Proceedings of the International Conference on Software Testing Verification and Validation*, 2009.
10. B. He and K. C.-C. Chang. Statistical schema matching across web query interfaces. In *Proceedings of the International Conference on Management of Data*, 2003.
11. R. Lo, R. Webby, and R. Jeffery. Sizing and estimating the coding and unit testing effort for gui systems. In *proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, 1996.
12. L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: a tool for automatic black-box testing. In *Proceeding of the international conference on Software engineering*, 2011.
13. H. Nguyen, T. Nguyen, and J. Freire. Learning to extract form labels. *Proceedings of the VLDB Endowment*, 1, August 2008.
14. J. Preece, Y. Rogers, H. Sharp, D. Benyon, S. Holland, and T. Carey. *Human-Computer Interaction.* Addison Wesley, 1994.
15. O. Sánchez Ramón, J. Sánchez Cuadrado, and J. García Molina. Model-driven reverse engineering of legacy graphical user interfaces. In *Proceedings of the International Conference on Automated Software Engineering*, 2010.
16. R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *proceedings of the International Symposium on Fault-Tolerant Computing*, 1997.
17. W. F. Tichy and S. J. Koerner. Text to software: developing tools to close the gaps in software engineering. In *proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010.
18. M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier. Automation of gui testing using a model-driven approach. In *Proceedings of the 2006 international workshop on Automation of software test*, 2006.
19. W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep Web. In *Proceedings of the International Conference on Management of Data*, 2004.
20. X. Yuan and A. M. Memon. Generating event sequence-based test cases using GUI run-time state feedback. *IEEE Transactions on Software Engineering*, 36(1):81–95, 2010.