Analysis of Computing Open Source Systems

J.C. Silva[1] **and J.L. Silva**[2]

[1] **Dep. Tecnologias, Instituto Politécnico do Cávado e do Ave,**
Barcelos, Portugal
jcsilva@ipca.pt

[2] **Madeira-ITI, Universidade da Madeira**
Funchal, Portugal
jose.l.silva@m-iti.org

Abstract

Graphical user interfaces (GUIs) are critical components of today's open source software. Given their increased relevance, the correctness and usability of GUIs are becoming essential. This paper describes the latest results in the development of our tool to reverse engineer the GUI layer of interactive computing open source systems. We use static analysis techniques to generate models of the user interface behavior from source code. Models help in graphical user interface inspection by allowing designers to concentrate on its more important aspects. One particular type of model that the tool is able to generate is state machines. The paper shows how graph theory can be useful when applied to these models. A number of metrics and algorithms are used in the analysis of aspects of the user interface's quality. The ultimate goal of the tool is to enable analysis of interactive system through GUIs source code inspection.

Keywords: analysis, source code, quality.

1. Introduction

In the user interface of an open source software systems, two interrelated sets of concerns converge. Users interact with the system by performing actions on the graphical user interface (GUI) widgets. These, in turn, generate events at the software level, which are handled by appropriate listener methods. In brief, and from a user's perspective, graphical user interfaces accept as input a pre-defined set of user-generated events, and produce graphical output. The users' interest is in how well the system supports their needs.

From the programmers perspective, typical WIMP-style (Windows, Icon, Mouse, and Pointer) user interfaces consist of a hierarchy of graphical widgets (buttons, menus, text-fields, etc) creating a front-end to the software system.

An event-based programming model is used to link the graphical objects to the rest of the system's implementation. Each widget has a fixed set of properties and at any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI. The programmers interest, besides satisfying the user, is in the intrinsic quality of the implementation, which will impact the system's maintainability.

As user interfaces grow in size and complexity, they become a tangle of object and listener methods, usually all having access to a common global state. Considering that the user interface layer of interactive open source systems is typically the one most prone to suffer changes, due to changed requirements and added features, maintaining the user interface code can become a complex and error prone task. Integrated development environments (IDEs), while helpful in that they enable the graphical definition of the interface, are limited when it comes to the definition of the behavior of the interface.

In this paper we explore an approach for the analysis of open source system's user interfaces. Open-source software is software whose source code is made available, enabling anyone to copy, modify and redistribute the source code without paying royalties or fees. This paper discusses an approach to understand and evaluate an open source system from an interactive perspective. We present a static analysis based framework for GUI-based applications analysis from source code.

In previous papers [1,3] we have explored the applicability of slicing techniques [4] to our reverse engineering needs, and developed the building blocks for the approach. In this paper we explore the integration of analysis techniques into the approach, in order to reason about GUI models.

The paper is organized as follow: Section three discusses the value of inspecting source code from a GUI quality perspective; Section four introduces our framework for GUI reverse engineering from source code; sections five and six presents the analysis of a software system; Section seven discusses the results of the process; the paper end with conclusions in Section eight.

2. Analysis of Open Source Systems

Open source systems are popular both in business and academic communities with products such as Linux, MySQL, OpenOffice or Mozilla. Open source systems are free redistribution with source code accessible and complying several criterions. The program must allow distribution in source code as well as compiled form. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed. The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software [5]. Considering that open source systems are typically prone to suffer changes, due to modifications and derived works, maintaining the system and its usability can become an error prone task [6]. A number of challenges remain to be met, however, many of which are common to all open source projects. This Section discusses open source systems analysis as a way to foster adoption and deployment of open source systems. The objective of the open source analysis is to evaluate the quality of open source systems involving software analysis and engineering methodologies. In the literature, several directions are used for achieving this goal such as testing, light weight verification and heavy weight verification, e.g [7,8]. Testing is a huge area for open source analysis [9]. Different kinds of tests are applied such as functional testing, regression testing, stress testing, load testing, unit testing, integration testing, documentation analysis, source code analysis, reverse engineering. Lightweight verification includes various methods of static analysis and model checking, e.g. [10]. These may include identification of domain specific restrictions and typical bugs for automatic detection, formal representation of the restrictions in terms of the tools used, development of simplified models of target system to be used for automatic analysis, automatic analysis of target source code with verification tools and investigation and classification of results.
Another approach is heavyweight verification providing a more complete analysis of the quality of the source code system. There are different approaches to heavyweight verification. Classical methods of verification requires to formally describe requirements in the form of precondition and post-condition. Then, invariants and variants should be defined for the open source system. After that verification tools automatically generate conditions in high order logic. Proof of the conditions is usually conducted within interactive theorem provers such as PVS or Coq [11,12].
We believe that defining and integrating a methodology into open source systems development processes should be the first priority to certificate open source systems.

3. Inspection from source code

The evaluation of an open source software is a multifaceted problem. Besides the intrinsic quality of the implementation, we have to consider the user reaction to the interface (i.e. its usability [13]). This involves issues such as satisfaction, learnability, and efficiency. The first item describes the user's satisfaction with the open source system. Learnability refers to the effort users make to learn how to use the application. Efficiency refers to how efficient the user can be when performing a task using the application.

The analysis of a system's current implementation can provide a means to guide development and to certify software. For that purpose adequate metrics must be specified and calculated [14,15]. Metrics can be divided into two groups: internal and external [16]. External metrics are defined in relation to running software. In what concerns GUIs, external metrics can be used as usability indicators. They are often associated with the following attributes [17]:

- Easy to learn: The user can do desired tasks easily without previous knowledge;
- Efficient to use: The user reaches a high productivity level.
- Easy to remember: The re-utilization of the system is possible without a high level of effort.
- Few errors: Errors are made hardly by the users and the system permits to recover from them.
- Pleasant to use: The users are satisfied with the use of the system.

However, the values for these metrics are not typically obtainable from direct analysis of the implementation, rather through users' feedback to using the system.
Internal metrics are obtained by source code analysis, and provide information to improve software development. Such metrics measure software aspects, such as source lines of code, functions invocations, etc. A number of authors has looked at the relation between internal metrics and GUI quality. Stamelos et al. [18] used the Logiscope[1] tool to calculate values of selected metrics in order to

1http://www-01.ibm.com/software/awdtools/logiscope/

study the quality of Open Source code. Ten different metrics were used. The results enable evaluation of each function against four basic criteria: testability, simplicity, readability and self-descriptiveness. While the GUI layer was not specifically targeted in the analysis, the results indicated a negative correlation between component size and user satisfaction with the software.

Yoon and Yoon [19] developed quantitative metrics to support decision making during the GUI design process. Their goal was to quantify the usability attributes of interaction design. Three internal metrics were proposed and defined as numerical values: complexity, inefficiency and incongruity. The authors expect that these metrics can be used to reduce the development cost of user interaction.

While the above approaches focus on calculating metrics over the code, Thimbleby and Gow [20] calculate them over a model capturing the behavior of the application. Using graph theory they analyze metrics related to the users' ability to use the interface (e.g., strong connectedness ensure no part of the interface ever becomes unreachable), the cost of erroneous actions (e.g., calculating the cost of undoing an action), or the knowledge needed to use the system (e.g., the minimum cut identifies the set of actions that the user must know in order to to be locked out of parts of the interface).

In a sense, by calculating the metrics over a model capturing GUI relevant information instead of over the code, the knowledge gained becomes closer to the type of knowledge obtained from external metrics. While Thimbleby and Gow manually develop their models from inspections of the running software/devices, an analogous approach can be carried out analyzing the models generated directly from source code. We have been developing a tool to reverse engineer models of a user interface from its source code [1,3]. By coupling the type of analysis in [20] with our approach, we are able to obtain the knowledge directly from source code. By calculating metrics over the behavioral models, we aim to acquire relevant knowledge about the dialogue induced by the interface, and, as a consequence, about how users might react to it. In this paper we describe several kinds of inspections making use of metrics.

4. The Tool

The tool's goal is to be able to extract a range of models from source code. In the present context we focus on finite state models that represent GUI behavior. That is, when can a particular GUI event occur, which are the related conditions, which system actions are executed, or which GUI state is generated next. We choose this type of model in order to be able to reason about and test the dialogue supported by a given GUI implementation.

The tool performs the parsing of the source code. A module executes this step. To implement this first module, a parser for the programming language being considered is used. The tool has been used to reverse engineer Java and Haskell [21] programs written using the (Java) Swing, GWT, and (Haskell) WxHaskell GUI toolkits. For the Java/Swing and GWT toolkits, the SGLR parser has been applied whose implementation can be accessible via the Strafunski tool [22]. For the WxHaskell toolkit the Haskell parser that is included on the Haskell standard libraries was used. Whatever the parser, it generates an Abstract Syntax Tree (AST). The AST is a formal representation of the abstract syntactical structure of the source code.

The full AST represents the entire code of the application. However, the tool's objective is to process the GUI layer of interactive open source systems, not the entire source code.

To this end, an another module implements a GUI code slicing process using strategic programming. The module is used to slice the AST produced by the compiler, in order to extract its graphical user interface layer. The module is composed of a slicing library, containing a generic set of traversal functions that traverse any AST.

Once the AST has been created and the GUI layer has been extracted, GUI behavioral modeling can be processed. A module implements a GUI abstraction step. The module is language independent. It generates a model of user interface behavior. The relevant abstractions used in the model are user inputs, user selections, user actions and output to user.

More specifically, the modules generates GUI-related metadata files with information on possible GUI events, associated conditions and actions, and states resulting from these events. Each of these items of data are related to a particular fragment from the AST. These are GUI specifications written in the Haskell programming language. These specifications define the GUI layer by mapping pairs of event/condition to actions.

5. HMS Case Study: A Larger Interactive System

In previous Section, we have presented the implemented tool. In this Section, we present the application of the tool to a complex/large real interactive system: a Healthcare Management System (HMS) available from Planet-source-code[2], one of the largest public source code database on the Internet. The goal of this Section is twofold: Firstly, it is a proof of concept for the tool. Secondly, we wish to

2 http://www.planet-source-code.com/

analyze the interactive parts of a real application.
The HMS system is implemented in Java/Swing and supports patients, doctors and bills management. The implementation contains 66 classes, 29 windows forms (message box included) and 3588 lines of code.
The login window is the first window that appears to HMS users. This window gives authorized users access to the system and the HMS main form, through the introduction of a user name and password pair. This window is composed of two text box (i.e. username and password input) and two buttons (i.e. Login and Exit buttons).
If the user introduces a valid user name/password and presses the Login button, then the window closes and the main window of the application is displayed. On the contrary, if the user introduces invalid data, then a warning message is produced and the login window continues to be displayed. By pressing the Exit button, the user exits the application.
Applying the tool to the source code of the application, and focusing on the login window, enables the generation of several models. Figure 1, for example, shows the graph generated to capture the login window's behavior. Associated to each edge there is a triplet representing the event that triggers the transition, a guard on that event (here represented by a label identifying the condition being used), and a list of interactive actions executed when the event is selected (each action is represented by a unique identifier which is related to the respective source code).

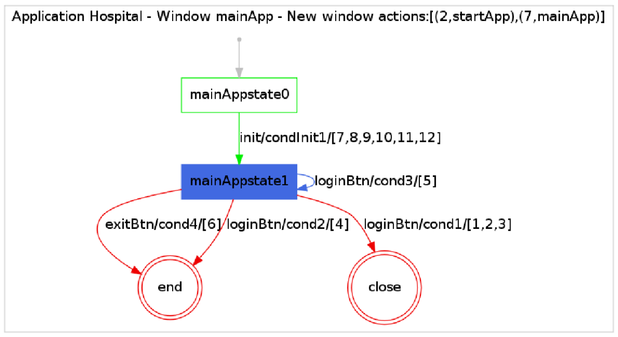
Application Hospital - Window mainApp - New window actions:[(2,startApp),(7,mainApp)]
mainAppstate0
init/condInit1/[7,8,9,10,11,12]
mainAppstate1
loginBtn/cond3/[5]
exitBtn/cond4/[6]
loginBtn/cond2/[4]
loginBtn/cond1/[1,2,3]
end
close

Fig. 1 HMS: Login behavioral graph

Analyzing this model, one can infer that there is an event/condition pair (edge loginBtn / cond1, with action list [1,2,3]) which closes the window (cf. edge moving to close node). Investigating action reference 2, it can be further concluded that another window (startApp) is subsequently opened. Furthermore, one can also infer that there are two event/condition pairs (edge exitBtn / cond4 with action list [6], and edge loginBtn / cond2 with action list [4]) which exit the system. These events can be executed by clicking the Exit or Login buttons, respectively. The informal description of login window behavior provided at the start of the Section did not included the possibility of exiting the system by pressing the Login button. The extracted behavioral graph however defines that possibility, which can occur if condition cond2 is verified (cf. pair loginBtn/cond2 with action list [4]). Analysing condition cond2 (*source.equals(exitBtn)*), dead code was encountered. The source code executed when pressing the Login button uses a condition to test whether the clicked button is the Login button or not. This is done through the boolean expression *source.equals(loginBtn)*. However, the above action source code is only performed when pressing the Login button. Thus, the condition will always be verified and the following *else* component of the conditional statement will never be executed.
Summarizing the results obtained for the login window, one can say that the generated behavioral graph contains an event/condition/actions triplet that does not much the informal description of the system. Furthermore, this triplet cannot be executed despite being defined on the behavioral model. This example demonstrates how comparing expected application behavior against the models generated by the tool can help understand (and detect problems in) the applications' source code.

6. GUI Inspection through Graph Theory

This Section describes some examples of analysis performed on the application's behavioral graph from the previous section. We make use of the implemented tool for the manipulation and statistical analysis of the graph.

6.1 Graph-tool

Graph-tool[3] is an efficient python module for the manipulation and statistical analysis of graphs. It allows for the easy creation and manipulation of both directed or undirected graphs. Arbitrary information can be associated to the nodes, edges or even the graph itself, by means of property maps. Graph-tool implements all sorts of algorithms, statistics and metrics over graphs, such as degree/property histogram, combined degree/property histogram, vertex-vertex correlations, average vertex-vertex shortest distance, isomorphism, minimum spanning tree, connected components, maximum flow, clustering coefficients, motif statistics, communities, or centrality measures.

3 http://projects.forked.de/graph-tool/

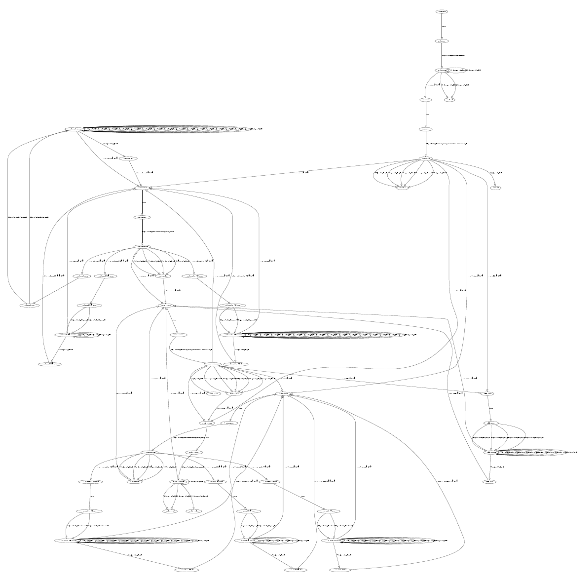
Fig. 2 HSM: The overall behavior

Now we will consider the graph described in Figure 2 where all vertices and edges are labeled with unique identifiers. Figure 2 provides the overall behavior of the HMS system. This model can be seen in more detail in the electronic version of this paper. Basically, this model aggregates the state machines of all HMS forms. The right top corner node specifies the HMS entry point, i.e. the *mainAppstate0* creation state from the login's state machine (cf. Figure 1).

6.2 GUI Metrics

As discussed in this paper, one of our goals is to show how the implemented tool supports the use of metrics such as those used by Thimbleby and Gow [20] to reason about the quality of a user interface. To illustrate the analysis, we will consider three metrics: Shortest distance between vertices, Pagerank and Betweeness.

The Graph-Tool enables us to calculate the shortest path between two vertices. This is useful to calculate the number of steps to execute a particular task. These results can be used to analyze the complexity of an interactive application's user interface. Higher numbers of steps represent complex tasks while lower values are applications with simple tasks. It can also be applied to calculate the center of a graph. The center of a graph is the set of all vertices *A* where the greatest distance to other vertices *B* is minimal. The vertices in the center are called central points. Thus vertices in the center minimize the maximal distance from other points in the graph. Finding the center of a graph is useful in GUI applications where the goal is to minimize the steps to execute a particular task (i.e. edges between two points). For example, placing the main window of an interactive system at a central point reduces the number of steps a user has to execute to accomplish tasks.

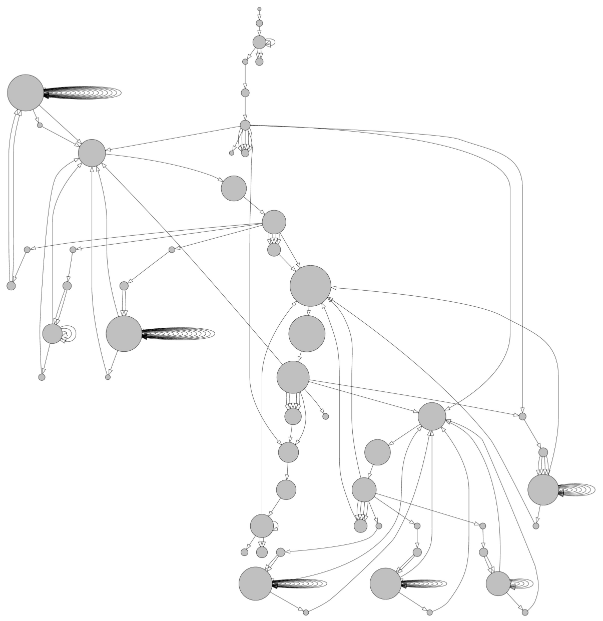
Fig. 3 HSM's pagerank results

PageRank is a link analysis algorithm, used by the Google Internet search engine that assigns a numerical weighting to each element of a hyperlinked set of documents. The main objective is to measure their relative importance. The wight assigned to each element represents the probability that a person randomly clicking on links will arrive at any particular page [23]. A probability is expressed as a numeric value between 0 and 1. This same algorithm can be applied to our GUI's behavioral graphs. Figure 3 provides the result obtained when applying the pagerank algorithm to graph of Figure 2. The size of a vertex corresponds to its importance within the overall application behavior. This metric can have several applications, for example, to analyze whether complexity is well distributed along the application behavior. In this case, there are no particularly salient vertices, which is an indication that interaction complexity is well distributed considering the overall application. It is also worth noticing that according to this criteria, the Main window is clearly a central point in the interaction.

Betweenness is a centrality measure of a vertex or an edge within a graph [24]. Vertices that occur on many shortest paths between other vertices have higher betweenness than those that do not. Similar to vertices betweenness centrality, edge betweenness centrality is related to shortest path between two vertices. Edges that occur on many shortest paths between vertices have higher edge betweenness.

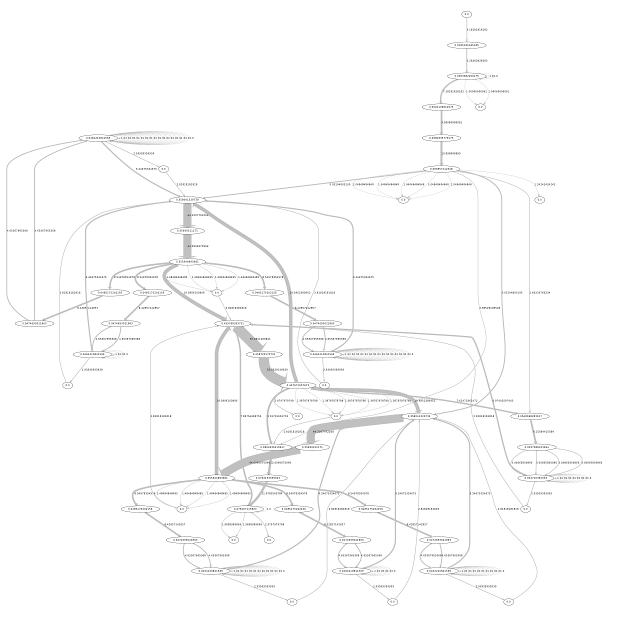
Fig. 4 HSM's betweenness values

Figure 4 provides the obtained result when applying the betweenness algorithm. Betweenness values are expressed numerically for each vertices and edges. Highest betweenness edges values are represented by thicker edges. Some states and edges have the highest betweenness, meaning they act as a hub from where different parts of the interface can be reached. Clearly they represent a central axis in the interaction between users and the system. In a top down order, this axis traverses the following states *patStartstate0*, *patStartstate1*, *startAppstate0*, *startAppstate1*, *docStartstate0* and *docStartstate1*. States *startAppstate0* and startAppstate1 are the main states of the startApp window's state machine.
The Main window has the highest betweenness, meaning it acts as a hub from where different parts of the interface can be reached. Clearly it will be a central point in the interaction.

6.3 GUI Testing

The reverse engineering approach described in this paper allows us to extract an abstract GUI behavior specification.
Our next goal is to perform model-based GUI testing. To this end, we make use of the QuickCheck Haskell library tool. QuickCheck is a tool for testing Haskell programs automatically. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators. Considering the application described in the previous section and its abstract GUI model-based we could now write some rules and test them through the QuickCheck tool. To illustrate the approach, we will test if the application satisfies the following rule: users need to execute less than three actions to access the main window.
The rule is specified in the Haskell language. From the windows set we automatically generate randomly cases. We extract valid GUI sentences from a GUI behavioral model. Then the rule is tested in a large number of cases (10000 in this GUI testing process!). The number of random cases and event lengths are specified by the user. Each random case is a sequence of valid events associated with their conditions, actions and the respective window. In other words, each case is a sequence of possible events, so all respective conditions are true in this context.
This approach enables to analyze a GUI model using a model-based testing technique. Though our approach is non-exhaustive, this is a technique which allows us to test the quality of models at a lower cost than other exhaustive techniques such as model checking. This section's focus is on GUI testing. Coverage criteria for GUIs are important rules that provide an objective measure of test quality. We plan to include coverage criteria to help determine whether a GUI has been adequately tested. These coverage criteria use event sequences to specify a measure of test adequacy. Since the total number of permutations of event and condition sequences in any GUI is extremely large, the GUI's hierarchical structure must be exploited to identify the important event sequences to be tested.

6.4 Conclusions

This Section described the results obtained with the implemented tool when applying it to a larger interactive system. The chosen interactive system case study is related to a healthcare management system (HMS). The HMS system is implemented in Java/Swing programming language and implement operations to allow for patients, doctors and bills management. A description of main HMS windows has been provided, and tool results have been described. The tool enabled the extraction of different behavioral models. Methodologies have been also applied automating the activities involved in GUI model-based reasoning, such as, pagerank and betweenness algorithms. GUI behavioral metrics have been used as a way to analyze GUI quality. This case study demonstrated that the tool enables the analysis of real interactive applications written by third parties.

7. Discussion

The previous section has illustrated how the implemented tool makes possible high-level graphical representation of GUI behavior from thousand of lines of code. The process is mostly automatic, and enables reasoning over the interactive layer of open source systems. Examples of some of the analysis that can be carried out were provided. Other uses of the models include, for example, the generation of test cases, and/or support for model-based testing. During the development of the framework, a particular emphasis was placed on developing tools that are, as much as possible, language independent. Through the use of generic programming techniques, the developed tools aim at being retargetable to different user interface programming toolkits and languages. At this time, the framework supports (to varying degrees) the reverse engineering of Java code, either with the Swing or the GWT (Google Web Toolkit) toolkits, and of Haskell code, using the WxHaskell GUI library. Originally the tool was developed for Java/Swing. The WxHaskell and GWT retargets have highlighted successes and problems with the initial approach. The amount adaptation and the time it took to code are distinct. The adaptation to GWT was easier because it exploits the same parser. The adaptation to WxHaskell was more complex as the programming paradigm is different, i.e. functional. Using the tool, programmers are able to reason about the interaction between users and a system at a higher level of abstraction than that of code. A range of techniques can be applied on the generated models. They are amenable, for example, to analysis via model checking [25]. Here however, we have explored alternative, lighter weight approaches.
Considering that the graphs generated by the reverse engineering process are representations of the interaction between users and system, we have shown how metrics defined over those graphs can be used to obtain relevant information about the interaction. This means that we are able to analyze the quality of the user interface, from the users perspective, without having to resort to external metrics which would imply testing the system with real users, with all the costs that process carries.
Additionally, we have explored the possibility of analyzing the graphs via a testing approach, and how best to generate test cases. It must be noted that, while the approach enables us to analyze aspects of user interface quality without resorting to human test subjects, the goal is not to replace user testing. Ultimately, only user testing will provide factual evidence of the usability of a user interface. The possibility of performing the type of analysis we are describing, however, will help in gaining a deeper understanding of a given user interface. This will promote the identification of potential problems in the interface, and support the comparison of different interfaces, complementing and minimizing the need to resort to user testing. Similarly, while the proposed metrics and analysis relate to the user interface that can be inferred from the code, the approach is not proposed as an alternative to actual code analysis.
Metrics related to the quality of the code are relevant, and indeed the tool is also able to generate models that capture information about the code itself. Again, we see the proposed approach as complementary to that style of analysis. Results show the reverse engineering approach adopted is useful but there are still some limitations. One relates to the focus on event listeners for discrete events. This means the approach is not able to deal with continuous media and synchronization/timing constraints among objects. Another has to due with layout management issues. The tool cannot extract, for example, information about overlapping windows since this must be determined at run time. Thus, we cannot find out in a static way whether important information for the user might be obscured by other parts of the interface. A third issue relates to the fact that generated models reflect what was programmed as opposed to what was designed. Hence, if the source code does the wrong thing, static analysis alone is unlikely to help because it is unable to know what the intended outcome was. For example, if an action is intended to insert a result into a text box, but input is sent to another instead. However, if the design model is available, the tool can be used to extract a model of the implemented system, and a comparison between the two can be carried out.
Additionally, using graph operations, models from different implementations can be compared in order to assess whether two systems correspond to the same design, or to identify differences between versions of the same system.

8. Conclusions

In what concerns interactive open source software development, two perspectives on quality can be considered. Users, on the one hand, are typically interested on what can be called external quality: the quality of the interaction between users and system. Programmers, on the other hand, are typically more focused on the quality attributes of the code being produced. This work is an approach to bridging this gap by allowing us to reason about GUI models from source code. We described GUI models extracted automatically from the code, and presented a methodology to reason about the user interface model. A number of metrics over the graphs representing the user interface were investigated. An approach to testing the graph against desirable properties of the interface was also put forward. A number of issues still needs addressing. In the example used throughout the paper, only one windows could be active at any given time (i.e., windows were modal). The tool is also able to work with non-model windows (i.e., with GUIs where users are able to freely move between open application windows). In that case, however, nodes in the graph come to represents sets of open windows instead of a single active window. While all analysis techniques are still available, this new interpretation of nodes creates problems in the interpretation of some metrics that need further consideration. The problem is exacerbated when multiple windows of a given type are allowed (e.g., multiple editing windows). Coverage criteria provide an objective measure of test quality. We plan to include coverage criteria to help determine whether a GUI has been adequately tested. These coverage criteria use events and event sequences to specify a measure of test adequacy. Since the total number of permutations of event and condition sequences in any GUI is extremely large, the GUI's hierarchical structure must be exploited to identify the important event sequences to be tested.
This work presents an approach to the analysis of interactive open source systems through reverse engineering process. Models enable us to reason about both metrics of the design, and the quality of the implementation of that design. Our objective has been to investigate the feasibility of the approach. We believe this style of approach can feel a gap between the analysis of code quality via the use of metrics or other techniques, and usability analysis performed on a running system with actual users.

References

[1] J. C. Silva, J. C. Campos, and J. Saraiva, “Combining formal methods and functional strategies regarding the reverse engineering of interactive applications,” in Interactive Systems, Design, Specifications and Verification, Lecture Notes in Computer Science. DSV-IS 2006, the XIII International Workshop on Design, Specification and Verification of Interactive System, Dublin, Ireland, pp. 137–150, Springer Berlin / Heidelberg, July 2006.
[2] J. C. Silva, J. C. Campos, and J. Saraiva, “Models for the reverse engineering of java/swing applications,” ATEM 2006, 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies for Reverse Engineering, Genova, Italy, October 2006.
[3] J. C. Silva, J. C. Campos, and J. Saraiva, “A generic library for gui reasoning and testing,” in In ACM Symposium on Applied Computing, pp. 121–128, March 2009.
[4] F. Tip, “A survey of program slicing techniques,” Journal of Programming Languages, september 1995.
[5] D. Cerri, A. Fuggetta, D. Cerri, A. Fuggetta, and C. P. D. Milano, “Open standards, open formats, and open source,” 2007.
[6] C. Benson, “Professional usability in open source projects,” in Conference on Human Factors in Computing Systems, pp. 1083–1084, ACM Press, 2004.
[7] H. Nakajima, T. Masuda, and I. Takahashi, “Gui ferret: Gui test tool to analyze complex behavior of multi-window applications,” in Proceedings of the 2013 18th International Conference on Engineering of Complex Computer Systems, ICECCS ’13, (Washington, DC, USA), pp. 163–166, IEEE Computer Society, 2013.
[8] C. E. Silva and J. C. Campos, “Combining static and dynamic analysis for the reverse engineering of web applications,” in Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS ’13, (New York, NY, USA), pp. 107–112, ACM, 2013.
[9] D. H. Department and D. Horton, “Software testing.”
[10] C. D. Roover, I. Michiels, K. Gybels, and K. Gybels, “An approach to high-level behavioral program documentation allowing lightweight verification,” in In Proc. of the 14th IEEE Int. Conf. on Program Comprehension, pp. 202–211, 2006.
[11] G. Sala¨un, G. Salan, G. Salan, C. Attiogb, C. Attiogb, C. Attiogb, M. Allem, M. Allem, and M. Allem, “Verification of integrated specifications using pvs.”
[12] J.-C. Filliatre, “Program verification using coq introduction to the why tool,” 2005.
[13] ISO, ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on Usability. International Organization for Standardization, 1998.
[14] F. A. Fontana and S. Spinelli, “Impact of refactoring on quality code evaluation,” in In Proceeding of the 4th workshop on Refactoring toolsi, Honolulu, HI, USA, 2011. ACM. Workshop held in conjunction with ICSE 2011, 2011.
[15] J. Al Dallal, “Object-oriented class maintainability prediction using internal quality attributes,” Inf. Softw. Technol., vol. 55, pp. 2028–2048, Nov. 2013.
[16] ISO/IEC, “Software products evaluation,” 1999. DIS 14598-1.
[17] J. Nielsen, Usability Engineering. San Diego, CA: Academic Press, 1993.
[18] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, “Code quality analysis in open source software development,”

Information Systems Journal, vol. 12, pp. 43–60, 2002.
[19] Y. S. Yoon and W. C. Yoon, "Development of quantitative metrics to support ui designer decision-making in the design process," in Human-Computer Interaction. Interaction Design and Usability, pp. 316–324, Springer Berlin / Heidelberg, 2007.
[20] H. Thimbleby and J. Gow, "Applying graph theory to interaction design," pp. 501–519, 2008.
[21] S. P. Jones, J. Hughes, L. Augustsson, et al., "Report on the programming language haskell 98," tech. rep., Yale University, Feb. 1999.
[22] R. Lammel and J. Visser, "A STRAFUNSKI application letter," tech. rep., CWI, Vrije Universiteit, Software Improvement Group, Kruislaan, Amsterdam, 2003.
[23] P. Berkhin, "A survey on pagerank computing," Internet Mathematics, vol. 2, pp. 73–120, 2005.
[24] S. Y. Shan and et al., "Fast centrality approximation in modular networks," 2009.
[25] J. C. Campos and M. D. Harrison, "Interaction engineering using the ivy tool," in ACM Symposium on Engineering Interactive Computing Systems (EICS 2009), (New York, NY, USA), pp. 35–44, ACM, 2009.