Third International Conference on Software Testing, Verification, and Validation Workshops

# Using Methods & Measures from Network Analysis for GUI Testing

Ethar Elsaka, Walaa Eldin Moustafa, Bao Nguyen, Atif Memon
*Department of Computer Science*
*University of Maryland*
*College Park, MD, USA*
*Email: (ethar|walaa|baonn|atif)@cs.umd.edu*

***Abstract*—Graphical user interfaces (GUIs) for today's applications are extremely large. Moreover, they provide many degrees of freedom to the end-user, thus allowing the user to perform a very large number of event sequences on the GUI. The large sizes and degrees of freedom create severe problems for GUI quality assurance, including GUI testing. In this paper, we leverage methods and measures from network analysis to analyze and study GUIs, with the goal of aiding GUI testing activities. We apply these methods and measures on the *event-flow graph* model of GUIs. Results of a case study show that "network centrality measures" are able to identify the most important events in the GUI as well as the most important sequences of events. These events and sequences are good candidates for test prioritization. In addition, the "betweenness clustering" method is able to partition the GUI into regions that can be tested separately.**

***Keywords*-GUI testing, event-flow graphs, network analysis, test prioritization, software testing, network centrality, betweenness clustering.**

## I. Introduction

Testing is widely recognized as a key quality assurance (QA) activity in the software development process. Research in testing has received considerable attention in the last two decades [5]. This paper focuses on system testing of a software that has a *graphical user interface* (GUI) front-end [10]. A software with a GUI front-end consists of two parts: (1) the underlying code that implements the "business logic" and (2) the GUI front-end that facilitates user interaction with the underlying code. A software user interacts with the GUI by performing *events*, such as button clicks, menu selections, and text inputs. The GUI uses the input events to interact with the underlying code via messages and method invocations.

During GUI testing, test cases, modeled as sequences of events are executed on the GUI and its output is compared to an "expected output." The goal of GUI testing is to reveal *GUI faults* (defined as one that manifests itself on the visible GUI at some point of time during the software's execution). Although the test cases interact only with the GUI interface, and the expected output consists only of GUI elements, Brooks *et al.* [2] have shown that a large proportion of faults detected by this type of testing are in the underlying business logic of the application, rather than in the GUI code itself.

Several techniques have been developed for GUI testing. One of the most successful model-based techniques is based on *Event-Flow Graphs* (EFGs) [9]. Although EFGs may be used to generate test cases that detect many GUI faults, these graphs are very large, and they yield an extremely large test suite. They also make it difficult to target testing to select parts of the GUI, and perform operations such as test selection and prioritization.

In this paper, we leverage methods and measures from the network analysis domain to analyze and study EFGs, with the goal of aiding GUI testing activities. In particular, we use the "network centrality measures" and "betweenness clustering" method. Adopting a network analysis- based approach for GUI testing is a new area of research. We illustrate our approach on a small GUI and then conduct a case study on a large GUI subject called *TerpPresent*, which is part of the *TerpOffice* [15] application suite developed by the University of Maryland students. Our results show that "network centrality measures" are able to identify the most important events in the GUI as well as the most important sequences of events. One application of this identification is that these events and sequences may be used for test prioritization; another is that maintenance of these events can be done more carefully than that of others. In addition, the "betweenness clustering" method is able to partition the GUI into independent regions; events in each region can, for example, be tested separately.

The rest of the paper is structured as follows. In the next section, we present an overview of network measures and methods that we use in this paper. In Section III, we discuss their application to a small GUI example. In Section IV, we provide our case study along with the results and their discussion. In Section V we discuss related work. Finally, we conclude the paper and discuss our plans for future extensions in Section VI.

## II. Overview of Some Networking Methods & Measures

In network analysis, ranking measures are widely used to reveal information regarding important entities in networks, such as nodes and edges. Ranking methods include PageRank [1], and Hubs and Authorities [7], which are more common in the hyperlinked WWW ranking domain.

978-0-7695-4050-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSTW.2010.61

Other importance measures include degree centrality and betweenness centrality [3].

On the other hand, clustering approaches divide the network into partitions such that entities in each partition are similar or close to each others with respect to a certain property. This property can be graph-structural (e.g., based on graph connectivity) or a local property of each entity that does not have to do with the graph structure. For example, in social networks, clustering based on topological aspects is known as *finding community structure* as it finds communities that have relatively dense interlinks inside them, with low number of crossing links between them. Examples of algorithms with that goal are Girvan-Newman algorithm [4] which is based on edge betweenness centrality, and algorithms that are based on modularity such as [11].

### A. Betweenness Centrality

Betweenness centrality is an importance measure that can be defined for both nodes and edges. Node betweenness centrality indicates that a node lies on many shortest paths between pairs of nodes, and hence, using short paths, it connects together nodes from the graph that would be otherwise distant. Formally, node betweenness is calculated using the following formula: $C(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where $\sigma_{st}$ is the number of shortest paths from $s$ to $t$, and $\sigma_{st}(v)$ is the number of shortest paths from $s$ to $t$ that pass through a vertex $v$. Similarly, edge betweenness centrality is a measure of edge importance, and is calculated by considering the shortest paths that run through that edge.

### B. Edge Betweenness Clustering

Edge betweenness clustering finds partitions by finding *crossing edges*, i.e., edges that are least central inside communities but are important in linking *between* communities. Therefore, this method starts with calculating the betweenness centrality of all the graph edges, and removes the edge with the highest centrality. The new centralities of the edges are recomputed and the operation of removing the highest-centrality edge is repeated, until the graph is disconnected to the desired number of partitions.

In this paper, we argue that betweenness centrality measure is appropriate for GUI testing and can be used effectively for finding important widgets, events, and event sequences for testing, and additionally we argue that edge betweenness clustering is a suitable method for partitioning the GUI EFG for smaller parts that can, for example, be tested individually.

## III. Applying Betweenness Ranking and Clustering to EFGs

By applying the measures discussed above on a GUI event flow graph, insights can be made to uncover hidden important GUI elements or event sequences. Since nodes correspond to events and edges correspond to event sequences in EFGs, ranking the events by their betweenness centrality results in the most important events in the sense that they are *central* to the GUI, and will be used often to reach from one part of the GUI to the other. Similarly, by ranking the EFG edges according to their edge betweenness centrality, not only single event importance is highlighted, but also importance of pairwise event sequences, which in turn can be generalized to higher length-sequences, so that a group of subsequent events are identified to be the main pipe through which the state of the application is transferred from one GUI part to another. Such event sequences are important to stress on during software development and testing, as they will likely hold most of the GUI state information and its transformation.

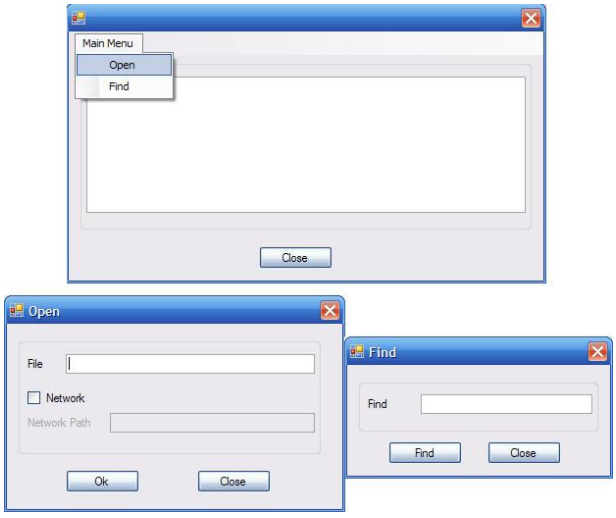


Figure 1. Example application

On the other hand, betweenness-clustering the event flow graphs is a means to break down the GUI into smaller units that are relatively cheaper to process and analyze, without worrying much about the interaction of these smaller units with each others, as the interaction is minimal, as given from the definition of the betweenness clustering. Another advantage of using betweenness clustering is that using it does not impose much overhead on the overall process, as the edge centralities will be calculated in order to rank the event sequences as we discussed above. Therefore, this computation can be utilized in finding the initial set of crossing edge candidates. For subsequent iterations, an incremental method can be used so that it takes into consideration changing only the centralities of the edges that have been affected by the removal of the edge selected in the previous iteration, keeping the re-computation overhead to a minimum.

### A. Example

To illustrate our proposed method, we use a small example application and walk through the analysis using its EFG. The example application is for a simple editing program that consists of three windows, one *Main* window, a *Open File* window, and a *Find* window. *Open File* is a modal window, i.e., it restricts the user to the set of events in the window, until it is explicitly terminated. *Find*, on the other hand, is a modeless window, i.e., it allows the user to perform events outside the window, even when it is open. The three windows are shown in Figure 1. The main window has a main menu with two menu items, each opens one of the other two sub windows. The main window also has an edit box, and a close button. The first sub-window, the *Open File* window, is loaded when the *Open* menu item is selected from the main menu. It has two text boxes, and a third box that is enabled only if its corresponding option *Network path* is selected. Additionally, this window has an *OK* button. The second sub-window, or the *Find* window, has a text box and an *OK* button. All of the three windows include a close button.

An EFG is a specific model of the GUI for a particular application, representing all possible sequences of events that a user can execute on that GUI. Nodes in the EFG represent events, and directed edges represent the *event-flow* relationship between two events. That is, an edge in the graph from event $e_1$ to $e_2$ indicates that event $e_2$ may be invoked *immediately after* event $e_1$. The predicate $follows(e_2, e_1)$ represents this relationship and denotes that $e_2$ *follows* $e_1$. EFGs are potentially cyclic, since events can typically be executed more than once during a session with an application.

The EFG of our application is shown in Figure 2. Note that events in the *Open* dialog cannot be preceded or followed by events from other widows because it is a modal window. On the other hand, most of the events in the other windows can interact, i.e., they have edges between them, because the *Find* window is modeless.

By applying the above measures on this example network, a set of interesting results can be found:

- **Event ranking:** The ranking of events according to their betweenness centrality score is shown, in sorted order, in Table I, where each widget (Column 2) is shown along with its rank (Column 1) and score (Column 3).
  By inspecting the rankings we make the following observations:
  - *Open-window-Close-button* has the highest rank because it is the crossing point between two main windows in the application, and one of them is modal. Therefore, the user cannot gain control over the parent window except by pressing that button first.
  - *Open-window-OK-button* has the second highest rank after the *Open-window-close-button*. It has similar properties to the close button but cannot be reached directly after the *Open-window* is opened, because there is a pre-condition that a file name is entered first. Therefore, the paths that are passing through it are not as many as the paths that pass through the close buttons.
  - *Main-menu* is the third because it is the cross point from the application's main windows to its sub-windows. Its score is lower than the above two because most of the paths that go through it are not shortest paths. For example the user can go directly from the *Main-menu-Edit-box* to the application's *Close-button* without having to go through the *Main-menu*.

Table I
EVENT RANKING OF THE EXAMPLE APPLICATION

| Rank | Event | Score |
|---|---|---|
| 1 | *Open-window-Close-button* | 33.5 |
| 2 | *Open-window-Ok-button* | 27.2 |
| 3 | *Main-menu* | 10.1 |
| 4 | *File-name-textbox* | 2.2 |
| 5 | *Enable-Network-Path-checkbox* | 2.2 |
| 6 | *Find-window-Close-button* | 2.1 |
| 7 | *Main-window-Edit-box* | 2.1 |
| 8 | *Find-textbox* | 2.1 |
| 9 | *Main-window-Close-button* | 2.1 |
| 10 | *Find-button* | 2.1 |
| 11 | *Open-menu-item* | 1.6 |

- **Event sequence ranking:** The ranking of events sequences according to their betweenness centrality is as follows. We show only the top 20 in Table II. These ranks show us that:
  - The first rank is for going from *Main-menu* to *Open-menu-item*. This is natural, because this action makes the application transition to a modal window, so it is the crossing line between two major parts of the application.
  - The sequences ranked from 2 to 10 are due to the importance of their last events *Open-window-Ok-button* and *Open-window-Close-button*.
  - The sequences ranked 11 to 20 are due to the importance of their first event, which is either *Open-window-Close-button* and *Open-window-Ok-button*.
- **Clustering:** We used betweenness clustering algorithm as described in Section II-B, where we kept iterating and removing edges until the graph was disconnected to two partitions. The result was two clusters; one that corresponds to the modal *File-window*, and another cluster that corresponds to the two other windows, *Main-window* and *Find-window*. This is useful because the clustering was able to determine that these two separate high level modules are, in fact, two different

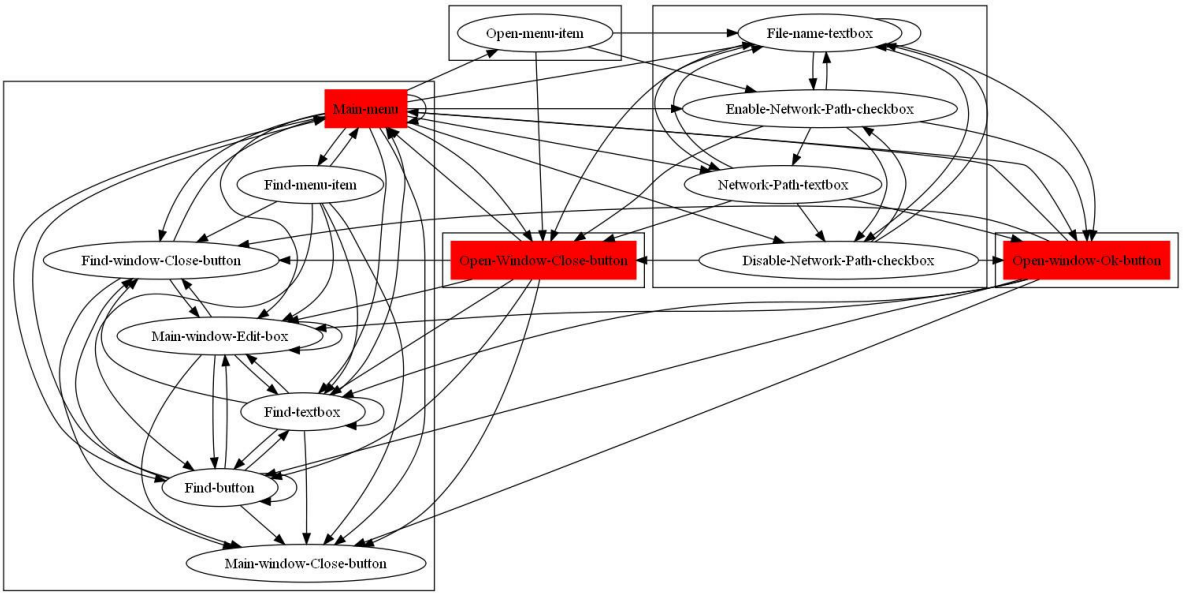


Figure 2. Example application EFG

Table II
EVENT SEQUENCE RANKING OF THE EXAMPLE APPLICATION

| Rank | Sequence | Score |
|---|---|---|
| 1 | *Main-menu* , *Open-menu-item* | 11.3 |
| 2 | *Network-Path-textbox* , *Open-window-Close-button* | 9.9 |
| 3 | *Disable-Network-Path-checkbox* , *Open-window-Close-button* | 9.9 |
| 4 | *Enable-Network-Path-checkbox* , *Open-window-Ok-button* | 9.5 |
| 5 | *File-name-textbox* , *Open-window-Ok-button* | 9.5 |
| 6 | *Network-Path-textbox* , *Open-window-Ok-button* | 9.2 |
| 7 | *Disable-Network-Path-checkbox* , *Open-window-Ok-button* | 9.2 |
| 8 | *File-name-textbox* , *Open-window-Close-button* | 8.8 |
| 9 | *Enable-Network-Path-checkbox* , *Open-window-Close-button* | 8.8 |
| 10 | *Open-menu-item* , *Open-window-Close-button* | 8.3 |
| 11 | *Open-window-Close-button* , *Main-window-Edit-box* | 8.2 |
| 12 | *Open-window-Close-button* , *Find-button* | 8.2 |
| 13 | *Open-window-Close-button* , *Find-window-Close-button* | 8.2 |
| 14 | *Open-window-Close-button* , *Main-window-Close-button* | 8.2 |
| 15 | *Open-window-Close-button* , *Find-textbox* | 8.2 |
| 16 | *Open-window-Ok-button* , *Main-window-Edit-box* | 7.2 |
| 17 | *Open-window-Ok-button* , *Find-button* | 7.2 |
| 18 | *Open-window-Ok-button* , *Find-window-Close-button* | 7.2 |
| 19 | *Open-window-Ok-button* , *Main-window-Close-button* | 7.2 |
| 20 | *Open-window-Ok-button* , *Find-textbox* | 7.2 |

entities. If we try to break down the clustering to more regions, by removing 25 edges, we get five clusters as shown in Figure 2, where the clusters are surrounded by rectangles. In this figure, we will find that each of the *Open-window-Ok-button*, the *Open-window-Close-button*, and the *Open-menu-item* are separated in their own clusters because they are the cross points between the first two big clusters.

With the above discussion, we can see the advantages of using betweenness centrality measure in ranking events or sequences of events, where these measures are useful for two reasons: 1) they shed the light on important actions in the GUI interaction graph that need more attention, and 2) these point out actions that form a crossing point between largely different application parts. Furthermore, betweenness clustering provides a means to enable the study the sub-parts themselves, because it can find out these somehow independent parts, given the fact that the results of how they are connected were already covered by node and edge betweenness.

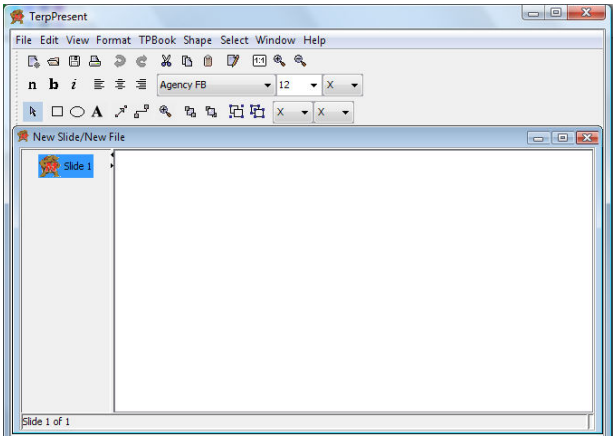


Figure 3. TerpPresent Screenshot

## IV. Case Study

To evaluate our approach, we now present a case study using *TerpPresent*, a presentation software developed by the students at the University of Maryland. We begin by describing the application, then describe the tools used to perform the study, and finally report the results.

### A. Application

A screenshot of the application is shown in Figure 3. TerpPresent has a main window with two main frames: a frame for the main menu along with the toolbar and a frame for the slide design workspace. The user accesses the software functionality through the menu and the toolbar, and uses the workspace frame to draw the presentation components, which can be text, pictures, backgrounds, and shapes. The user is allowed to select colors for text, pictures, and shapes using a color window.

### B. Tools

We reverse engineered TerpPresent's GUI using the GUITAR (http://guitar.sourceforge.net) software. The output is two XML files. The first file describes the EFG of the application, and the second file describes the widgets of the application in terms of their name, type (class), their hierarchical level and other attributes. Using this file we can relate the events in the EFG file to actual widgets that the user sees when running the application. We read the graph structure from the EFG and use JUNG API (jung.sourceforge.net), which is a scientific Java API for analyzing and visualizing different kinds of graphs, to analyze the TerpPresent EFG and output our three measures of interest.

### C. Results and Discussion

In this subsection we discuss the results we got by ranking the events, sequence of events, and performing clustering. EFG of TerpPresent contains 459 nodes and 2262 edges. Analyzing the EFG using JUNG on a machine with Intel Core 2 Duo CPU @2.26 GHz and 4 GB of RAM takes 10 seconds for performing each of event ranking, sequence ranking and edge betweenness clustering.

Table III
Event ranking of TerpPresent

| Rank | Event Widget | Score |
|---|---|---|
| 1 | *Slide-workspace* | 61220.1 |
| 2 | *Main-menu-bar* | 61220.1 |
| 3 | *Font-menu-item* | 21741.3 |
| 4 | *Help-menu* | 14994 |
| 5 | *About-menu-item* | 14994 |
| 6 | *Color-window-Ok-button* | 10769.1 |
| 7 | *Color-window-Close-button* | 10769.1 |
| 8 | *File-menu* | 1836.5 |
| 9 | *Text-menu-item* | 356.1 |
| 9 | *Scale-menu-item* | 309.2 |
| 9 | *Background-menu-item* | 295.3 |

Ranking events results are shown in Table III. We discuss them next:

- The first rank is for the main frame where the slides are drawn and designed. This coincides with the fact TerpPresent is a presentation software where most of its widgets and options are designed to serve this main frame and interact with it.
- The second rank is for the *Main-menu-bar*. This also coincides with the fact that the main menu bar is the widget through which all of the program options and functionalities can be reached.
- Thirdly comes the *Font-menu-item*. Naturally, selecting and setting fonts is one of the most important functionalities in a presentation software.
- Fourth comes the *Help-menu-item*. This menu item is not central to a presentation software in terms of functionality. However, it is important from the point of view that it leads to many help pages that cannot be reached otherwise. Similar is the case of the *About-menu-item* that opens another modal window with three tabs and several widgets.
- Afterwards comes the *OK* and *Cancel* buttons of the *Color-window*. This is again a crucial feature of a presentation software.
- Afterwards comes the menu *File* then the menu items *Text*, *Scale*, and *Background*. Again these menus are the hubs through which most of the application functionality can be reached.

The rest of the rankings have a very low score and therefore we do not discuss them.

Sequence rankings where affected by the existence of *Help* and *About* menus as one of the top important widgets. Therefore we disregarded the *Help-and-About-related* event sequences from the top ranking and found that the top event sequences are as shown in Table IV. We discuss them in the following bullets:

- Clicking *OK* or *Cancel* in the *Color-window* then

Table IV
EVENT SEQUENCE RANKING OF TERPPRESENT

| Rank | Sequence | Score |
|---|---|---|
| 1 | *Color-window-Ok-button , Slide-workspace* | 3835.0 |
| 2 | *Color-window-Cancel-button , Slide-workspace* | 3835.0 |
| 3 | *Color-window-Ok-button , Main-menu-bar* | 3835.0 |
| 4 | *Color-window-Cancel-button , Main-menu-bar* | 3835.0 |

working inside the main slide work frame, which stems from the fact that this sequence is the switch between two major application components, which are the color window and the slide work space.

- The next couple of sequences is clicking *OK* or *Cancel* in the *Color-window* then selecting a menu in the menu bar, which again connects two of the most important application components.

For the clustering results, we got three main clusters: the cluster of the color window widgets, the cluster with the *"About TerpPresent"* options, as it includes three tabs and other widgets such as text areas and scroll bars that are only reachable from the *About-menu-item*, and the last cluster was with the rest of the application's widgets.

As we mentioned earlier, the results suggest that there are components of the TerpPresent application that are more sensitive and important, such as the slide work space, the main menu bar, the color window and the font selection window. The results show that transitioning scenarios between these components should be taken more into account in testing and development, and furthermore, as we will discuss in Section VI for future work, according to these components relative importance, test case priortization can be accomplished using their relative rankings.

## V. RELATED WORK

There is a growing interest among software engineering researchers in applying network analysis techniques and their metrics for software engineering tasks. There are many forms where different information available from the software engineering process can modeled as a social network, where relationships take place between developers, developers and software, or otherwise. In this section we discuss some of the related work in the literature, where network analysis played a crucial role in promoting software development, reliability and management.

In [16], the authors study the collaborative nature of open software development from a network analysis perspective. They use SourceForge as a case study where they model developers as actors in the network, and a link between two developers denotes that they collaborated on at least one project. They analysis results show that the graph degree distribution is heavily skewed, which suggests that collaborations do not occur in random; however, there is a preferential attachment pattern, which can be explained by the fact that developers tend to join successful projects which in turn attracts more developers.

Relationship between developers and source code have not only been considered on the project level, but also on the code artifacts level. In [8], the authors utilize information obtained from CVS repositories to get insights about the internal structure of software projects. They use two types of networks. The first type is committer networks, where nodes denote developers and links between two developers denote the fact that they contributed to at least one common module. And the second type of networks is the module networks, where the nodes denote modules and links denote the existence of at least one committer who contributed to both of them.

In [12], the authors use networks of developers and projects as a tool for helping developers who want to seek experience and direct questions to other developers. The tool visualizes developer-developer networks, and project-project networks, focusing on the similarities between the vertices (similarity is defined as the number of common projects in developer-developer networks, and the number of common developers in project-project networks). Using visualization with the similarity metric helps finding most related projects and developers who can answer questions from similar projects or domains.

In [14] the authors empirically investigate the relationship between the fragmentation of developer contributions and the number of post-release failures, by representing the contributions as relationships between developers and modules in developer-module networks. They find that central binaries are more likely to be failure-prone than binaries located in surrounding areas of the network. They employ this correlation to show that error prone modules can be predicted by making use of advanced network centrality measures.

In [17], the authors compare the usage of code complexity measures versus network measures as an indicator of future software defects. Network measures are applied on the module dependency graph, while code complexity measures are obtained from modules separately and include measures such as the number of lines, methods, or classes in the source code. They found that network measures correlate positively with the number of post release defects, and can be used to predict their number.

To our knowledge no previous work models the GUI structure as a network or performs analysis on such a network to obtain insights about the GUI.

## VI. CONCLUSIONS AND FUTURE WORK

In this paper we described a novel technique for utilizing well established network measures for GUI testing. We exploit the fact the GUI interactions can be modeled as a graph, and used network analysis to reveal interesting insights from this graph. The measures we used are node

betweenness centrality, edge betweenness centrality and betweenness clustering. Future work involves expanding the set of network measures we employ to study the performance of other ranking methods such as PageRank, or Hubs and Authorities. We also plan to perform a comparative study to compare the performance of various network measures with existing techniques for ranking GUI events, enriching the input EFG with more information that can lead to more interesting results. Methods of enriching the input graph involve giving weights to the events or edges between events to indicate how likely they occur in an actual scenario, or how crucial they are to the application functionality. A motivating example for this is what happened with the Help menu. It ranked highly in our results because it leads to a big group of pages that cannot be reached otherwise. By decreasing the weight of the Help menu in the input graph, we can get results where this menu does not rank so highly. Other directions for future work include performing test case generation and prioritization using the ranking results, by combining sequences of highly ranked events so that they form a complete test case. We will compare the performance of employing network analysis with other systems that perform test case generation and prioritization. For test case generation, we will compare with other model-based approaches such as [13], and for test case prioritization, we will compare with weight-based approaches such as [6].

## ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation under NSF grants CNS-0855055 and CCF-0447864 and the Office of Naval Research grant N00014-05-1-0421.

## REFERENCES

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[2] P. Brooks, B. Robinson, and A. M. Memon. An initial characterization of industrial graphical user interface systems. In *International Conference on Software Testing, Verification and Validation*, pages 11–20, 2009.

[3] L. C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1):35–41, 1977.

[4] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America*, 99(12):7821–7826, June 2002.

[5] M. J. Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM Press.

[6] C.-Y. Huang, J.-R. Chang, and Y.-H. Chang. Design and analysis of gui test-case prioritization using weight-based methods. *Journal of Systems and Software*, 83(4):646 – 659, 2010.

[7] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, 1999.

[8] Luis, G. Barahona, and G. Robles. Applying social network analysis to the information in cvs repositories. In *Proceedings of the Mining Software Repositories Workshop. 26th International Conference on Software Engineering*, 2004.

[9] A. M. Memon and Q. Xie. Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, 2005.

[10] B. A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):64–103, 1995.

[11] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, June 2006.

[12] M. Ohira, N. Ohsugi, T. Ohoka, and K.-i. Matsumoto. Accelerating cross-project knowledge collaboration using collaborative filtering and social networks. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[13] A. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal. Modeling and testing hierarchical guis. In *Abstract State Machines*, pages 329–344, 2005.

[14] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12, New York, NY, USA, 2008. ACM.

[15] Terpoffice. http://www.cs.umd.edu/ atif/newsite/terpoffice.htm, 2010.

[16] G. von Krogh and S. Spaeth. The open source software phenomenon: Characteristics that promote research. *J. Strateg. Inf. Syst.*, 16(3):236–253, 2007.

[17] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 531–540, New York, NY, USA, 2008. ACM.