18th IEEE International Conference on Program Comprehension

Visualizing the Results of Field Testing

Brian Chan, Ying Zou
Dept. of Elec. and Comp. Engineering
Queen's University
Kingston, Ontario, Canada
{2byc, ying.zou}@queensu.ca

Ahmed E. Hassan
School of Computing
Queen's University
Kingston, Ontario, Canada
ahmed@cs.queensu.ca

Anand Sinha
Handheld Software
Research in Motion (RIM)
Waterloo, Ontario, Canada
asinha@rim.com

Abstract— **Field testing of software is necessary to find potential user problems before market deployment. The large number of users involved in field testing along with the variety of problems reported by them increases the complexity of managing the field testing process. However, most field testing processes are monitored using ad-hoc techniques and simple metrics (e.g., the number of reported problems). Deeper analysis and tracking of field testing results is needed. This paper introduces visualization techniques which provide a global view of the field testing results. The techniques focus on the relation between users and their reported problems. The visualizations help identify general patterns to locate the problems. For example, the technique identifies groups of users with similar problem profiles. Such knowledge helps reduce the number of needed users since we can pick representative users. We demonstrate our proposed techniques using the field testing results for four releases of a large scale enterprise application used by millions of users worldwide.**

Keywords- User Logs; Visualization; Pattern Identification; Automation

I. INTRODUCTION

Software must endure rigorous field testing before market deployment. Field testing helps uncover problems due to unexpected user behaviors and unforeseen usage patterns in a natural setting. Instrumented versions of the application are used during field testing. These versions enable the collection of field data in real-time without developer interference. A problem report is sent and stored in a central repository when an unexpected situation occurs. Common information in the report includes error messages and call stacks for a problem. The report also records the user who had the problem.

Developers must analyze and resolve these reported problems. Problems reported by a larger number of users are a high priority for developers to resolve. To verify the repair of a problem or to understand its peculiarity, developers must often replicate the scenarios which trigger the problem. By understanding the characteristics of the users reporting a problem and its peculiarities, developers should be able to gain insight into ways of resolving it.

In the current state of practice, developers often examine problems individually without a global view on the relation between the different problems and the relation between the users reporting these problems. For instance, it may be the case that:

1. A particular set of problems co-occur frequently together in the same time in a problem report therefore studying and resolving any one of these problems might lead to the resolution of all these problems. With some problems easier to replicate than others, such information is likely to lead to faster resolution of these problems. We developed a **problem graph** to highlight such information.
2. Several users often report the same set of problems (i.e., share the same problem profile); therefore recruiting a few of these users to verify any fix is often a faster option instead of deploying the fix across the field blindly and awaiting problem reports. We developed a **user graph** to identify users reporting common problems.
3. We can cross reference the problems reported and recruit users to replicate them. A large number of problems are reported; however very few of these problems have a wide impact on many users. Fixing problems with high impact is usually a high priority effort. We developed an **interaction graph** to give a global view of the relation between the participants of field testing (i.e., users) and the outcome of the testing (i.e., problems).

Using our visualization, developers can automatically analyze the large amount of data reported during field testing. Our visualization identifies patterns that demonstrate the user and problem interactions. The patterns allow developers to tackle problems with a global view instead of individually. In particular, such patterns help developers categorize, prioritize, and replicate problems, allowing developers to observe new relationships that are often overlooked in practice. By visualizing the field testing results across multiple product releases, developers can compare the progress of field testing efforts for different releases.

Organization of the Paper. The rest of the paper is organized as follows. Section 2 presents our three visualizations: problem graph, user graph and interaction graph. Section 3 presents a case study which demonstrates the use of our visualization to study the field testing results of a large enterprise application. Through our study we note several patterns in the visualization. We developed an automated approach to identify such patterns. We discuss these patterns and report on the accuracy of our automated

978-0-7695-4113-6/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPC.2010.9

approach. Section 4 discusses related work. Finally, Section 5 draws conclusions and discusses future work.

II. Our Three Graphs for Field Data

In this section, we introduce the three types of graphs that are produced from the field data. Our graphs could have as a node: a problem report, a user, or both. The edges in our graphs are based on different types of relations between the nodes. For example, an edge between two nodes in a problem graph indicates that the two nodes (i.e., problems) co-occur frequently together in a report. While an edge between two nodes in a user graph indicates that the two nodes (i.e., user) frequently report the same types of problems. Due to the large number of reported problems and users in field testing, we cannot simply show all the edges and nodes. Instead we must filter them. In the following subsections, we present our three graphs in more detail. We discuss the filtering process after the presentation of the three graph types.

A. *Problem Graph: Visualizing the Interaction among Problems*

Developers often tackle field problems on an individual basis – overlooking the possibility that certain problems may be interconnected. For instance, one problem reports that the disk is full, while another problem reports that the application fails to write to disk. Both problems can occur independently for different users or together for the same user who attempts to write to a full disk and triggers both problems. If the write-failure problem reports were sparse, developers might have a hard time to capture and fix the write-failure problem. However if developers can determine that this rare write-failure problem co-occurs frequently with a disk-full problem, developers can use the disk-full problem to better understand the write-failure problem and resolve it in a timely fashion. This situation occurs often in large complex applications, given the different coding and reporting standards followed by various groups. By flagging relations between problems, we can overcome the fact that some reports might be sparse or that some reports are harder to reproduce or investigate.

In a more complex situation, each occurrence of an error may trigger a different, yet often limited, number of problems. Such an error is often due to the use of an uninitialized field or timing/race conditions. When a user encounters the error several times, various problems associated with the same error can be reported by the same user in different occasions. Identifying the problems reported by the same user helps developers capture the common cause for the problems, and fix all the problems at once.

Finally, another type of problem co-occurrence happens when a single error leads to a cascade of related problems. For example, a problem, caused by an error in the code that reads data from the network, would often lead to multiple problems triggered in the subsequent execution of the application (due to the network-read errors). In an ideal situation, such problems would have been reported as the same type of problem. However, all too often the errors might manifest themselves as different problems. For example, an error to read from a network might manifest itself as the handling of a NullPointer problem, and the subsequent problems are reported as out of bound processing. The aforementioned situations highlight the importance of studying field problems together instead of individually. By interlinking problems, developers might discover unexpected yet important relations between problems during the field testing process. The problem graph establishes such interlinking among problems.

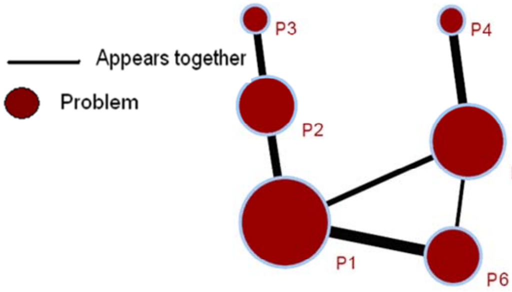


Figure 1. An Example of a Problem Graph

The problem graph is an undirected graph, $G_{Problem} = (V_P, E_P)$. The set of nodes, V_P, of the graph contains a set of problems (i.e., P) reported in the repository. We consider problem reports to be the same if they have the same call stack recorded when the problems are triggered. The set of edges, E_P, contains undirected edges, $e_{ab}=\{P_a, P_b\}$ if problem P_a and problem P_b are reported together by one or more user. Each user reports both problems (i.e., P_a and P_b). If a large number of users report the two problems together, then we believe that such a relation needs to be highlighted to developers for further investigation. The graph is designed so that only nodes with a direct edge have a relationship. For example, the interconnected nodes P_1, P_5 and P_6 indicate that all three problems have been reported although not necessarily all together. Moreover, there could have been three different users reporting each pair of problems (i.e., $\{P_1, P_5\}$, $\{P_5, P_6\}$ and $\{P_1, P_6\}$) or one user reporting the three problems together (i.e., $\{P_1, P_5, P_6\}\}$. A problem node appears in the problem graph only if it is reported together with at least one other problem node by a user.

We add weights to the nodes and edges. The weight of a node indicates the frequency of unique occurrences of that problem among users in the field testing repository. Visually, nodes are shown as circles with the weight of a node depicted as the size of the circle. Looking at Figure 1, P_1 is reported by more users than P_3. We define the weight of edges using statistical filtering techniques. We discuss it in Section II.D. Table I summarizes the definition of nodes and edges for the Problem Graph.

B. *User Graph: Visualizing the Interaction among Users*

Given the complexity of modern enterprise applications, the usage patterns between users tend to vary considerably with users forming clusters based on their usage of the application. For example, given an application for order taking and fulfillment, we would expect that at least two user clusters would arise along the two main uses of the application (i.e., order taking and fulfillment) with many sub-clusters forming based on the peculiarities of use within the two large clusters. Different clusters might also arise from commonalities across the user population. Similar characteristics such as similar hardware configuration or usage patterns (e.g., living in areas with bad wireless coverage or slow internet connections) are likely to lead users to report similar problems.

Flagging users with common problem profiles is of great value to developers. Using this knowledge about user clusters, developers could replicate specific problems and verify their fixes in an easier fashion. For instance, if most users in the order-taking department have the same problem profile, then a developer might be able to recruit a particular user to deploy additional monitoring on their installation. Developers can work closely with that user to solve a problem that affects the whole cluster of users in the order-taking department. Furthermore, using this knowledge about user clusters, managers can optimize their planning of future field testing efforts by picking at least one representative user from each cluster to participate in the field testing for a particular software version. Given the limited number of users that are willing to participate in field testing efforts and the large number of releases that are usually tested in parallel, a field-testing manager can distribute their user base more strategically. For example, if there are three versions of the software under testing and prior field testing shows the existence of two distinct user clusters, then the manager should ensure the participation of one user from each cluster to the three tests instead of randomly picking users.

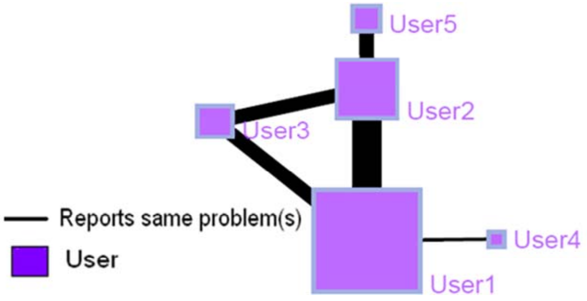


Figure 2. An Example of a User Graph

We developed the user graph to help identify such clusters. The user graph is an undirected graph, i.e., $G_{User} = (V_U, E_U)$. The set of nodes, V_U, of the user graph contains a set of users (i.e., U) who report problems sent to the repository. The set of edges, E_U, contains an undirected edge $e_{ab}=\{u_a, u_b\}$ if users, u_a and u_b, have reported at least one common problem.

The weight of a user node represents the number of unique problems that a user reports with other users. For example a user that reports the same problem with four other users has a greater weight than a user who reports two common problems with one other user. Visually, nodes are shown as squares with the weight of a node depicted as the size of the square. Looking at Figure 2, $User_1$ reports more problems in common with other users than $User_3$. We define the weight of edges using statistical filtering techniques. We discuss it in Section II.D. Table I summarizes the definition of nodes and edges for the User Graph.

C. Interaction Graph: Visualizing the Relations between Problems and Users

The interaction graph gives a global view of the results of field testing process. The graph cross references the relationship between problems and users. The interaction graph helps analyze the visualization produced by the other two graph types. For example, while the problems are not identified within the user graph, the interaction graph could be used to cross-reference the specific problems after locating a particular user base in the graph.

The interaction graph is an undirected, weighted graph, $G_{Inter} =(V_I, E_I)$. The set of nodes, V_I, is the union of the user set (i.e., U) and the problem set (i.e., P). Figure 3 illustrates an example of the interaction graph. The circular nodes represent problems and the square nodes represent users. The set of edges, E_I, contains an undirected edge, e_{ab}={P_a, $User_b$}, if $User_b$ reports a problem, P_a. For the example shown in Figure 3, $User_3$ and $User_7$ both report problems P_2 and P_3. P_1 is reported by $User_1$, $User_2$ and $User_3$.

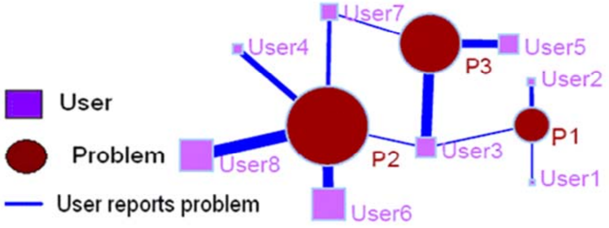


Figure 3. An Example of an Interaction Graph

The weight of a user node captures the number of unique problems reported by the user. Visually, the weight of a user node is shown as the size of the square node. The weight of a problem node reflects the frequency of the problem occurred during the field testing. Visually, the problems are depicted as circles with the weight of a node illustrated as the size of the circle. As depicted in Figure 3, $User_8$ reports more problems than $User_1$. Problem P_2 occurs more frequently than problem P_1. We define the weight of edges using statistical filtering techniques. We discuss it in Section II.D. Table I summarizes the definition of nodes and edges for the Interaction Graph.

D. Statistical Filtering and Weighting

Showing all the users and problems that exist in a field testing repository would lead to very complex graph with

TABLE I. SUMMARY OF THE THREE TYPES OF GRAPHS

Graph	Entity	Entity type	Weight	Filtering
Problem Graph	Node	Problem	The number of different users who report the problem at least once in the repository	Must co-occur with at least one other problem after edge filtering
	Edge	Both problems co-occur	Statistical weight (see section II.D)	Statistical filtering (see section II.D)
User Graph	Node	User	The number of unique problems that a user has in common with other users	Must report a single problem after edge filtering.
	Edge	Both users report the same problems	Statistical weight (see section II.D)	Statistical filtering (see section II.D)
Interaction Graph	Node	Problem, User	**Problem:** The number of times a problem is reported by any user **User:** The number of unique problems reported by a user	Same as above for problem and user nodes
	Edge	User reports the Problem	Statistical weight (see section II.D)	Statistical filtering (see section II.D)

high over plotting. Therefore we chose to filter the edges and nodes in the created graph. Traditionally such filtering is performed by showing edges which are above a particular threshold. For example, we could show edges between nodes which exhibit a co-occurrence relation 70% of the time or based on basic counts (e.g. 70 times). Instead we choose to perform statistical filtering given the large size of the data.

The goal of our **statistical filtering** is to only show edges that do not simply occur due to chance, instead we want to show edges that indicate statistically significant relations (i.e., association). We perform a chi-square (χ^2) test on each edge and filter edges in all graphs based on the results of the test [9]. Table II shows the 2x2 contingency table used to calculate the chi-square value. The same approach is used for all three graphs. For example, in the problem graph, the cell *(P_a, P_b)* counts the frequency (i.e., f_1) where problems P_a and P_b appear together for any users. The cell (P_a, $\neg P_b$) counts the number of times (i.e., f_2) that problem P_a occurred but not problem P_b for a user. The cell *($\neg P_a$, P_b)* counts the number of times (i.e., f_3) that problem P_b occurs but not problem P_a. The cell *($\neg P_a$, $\neg P_b$)* denotes the total number of times (i.e., f_4) that neither problem is reported. χ^2 is calculated using formula (1). Looking at the chi-square (χ^2) statistics distribution tables, we use a chi-square value of 5.99 to filter edges, indicating that we are 95% confident (i.e., p value<0.05) about the statistical validity of a shown edge.

Edges are given a **statistical weight** if it satisfies the conditions of statistical filtering (i.e., are statistically significant enough to be shown). The statistical weight of an edge is calculated using the *phi* measure [5]. The value of *phi* measures the strength of the relationship regardless of the sample size. It was chosen primarily to normalize the edge thickness in a given sample so that no edge would detract from the graph. It is based on the chi-squared test as shown in formula (1). The value of *phi* is computed using formula (2). A value of 0 for *phi* means that there is no association between two nodes (e.g., problems) and a value of 1 indicates a perfect association (i.e. both nodes always co-occur). Therefore, thick edges for any graph display *phi* values close to 1. The *phi* value is statistically significant when the χ^2 value is greater or equal to 5.99 (i.e., p< 0.05). We filter the edges if the weight of the edges is not statistically significant. Thin edges indicate low *phi* values meaning that the minimal statistical significance.

TABLE II. 2X2 CONTINGENGY TABLE

	P_b	$\neg P_b$
P_a	f_1	f_2
$\neg P_a$	f_3	f_4

$$\chi^2 = \frac{(f_1 f_4 - f_2 f_3)^2 (f_1 + f_2 + f_3 + f_4)}{(f_1 + f_2)(f_3 + f_4)(f_2 + f_4)(f_1 + f_3)} \qquad (1)$$

f_1, f_2, f_3, and f_4 are the frequencies of two independent events (e.g., problems) that appear for different cases listed in Table IIF.

$$phi = \sqrt{\frac{(\chi^2)}{(f_1 + f_2 + f_3 + f_4)}} \qquad (2)$$

x^2 is the chi squared value calculated in formula (1). f_1,, f_2, f_3, and f_4 are the frequencies of the two independent events as described in Table II.

III. CASE STUDIES

To demonstrate the benefits of using our proposed graphs, we performed a case study using four test versions of a large scale enterprise software application used by millions of users worldwide for communication. The application is written in the Java programming language. Table III shows descriptive statistics for each version. The data collected for this study was gathered over the course of 30 days within a field test of each version. To ease the comparison of problems of each version, identical problems appearing in any of the four versions are assigned the same identifier.

The objective of this case study is to identify if there are any significant differences or trends among the four versions of the application. We built a prototype tool that can automatically analyze the three types of graphs to identify patterns common in all versions. Due to space constraints, we only display results for versions A and D. Both versions have the highest number of users and reported problems. We also report on the accuracy of our automated pattern identification approach.

TABLE III. STATISTICS FOR THE FOUR STUDIED FIELD TESTS

Version	# of Users	# of Reported Problems
A	367	342
B	48	85
C	206	143
D	1,302	883

A. Steps for Identifying Patterns from the Graphs

Our prototype tool automatically generates the three types of graphs and identifies patterns in the following steps:

1. For all versions, we use our prototype tool to automatically analyze each problem report to determine the user who reported the problem and to determine similar problems using the reported call stacks. We also identify similar problems across all the studied versions.
2. We generate the edges between each user and problem. We apply statistical filtering and weighting of the edges as described in Section II.D and summarized in Table I. We filtered nodes which no longer have edges attached to them.
3. We visualize the graphs using a spring-based layout algorithm that is built in the GUESS (Graph Exploration System) [4]. GUESS is a system and language for visualizing and manipulating graph structures. GUESS accepts command scripts, which help highlight identified patterns from the data set. For example, Figures 4a), 6a), and 7a) are the three types of graphs visualized for version A after statistical filtering.
4. Our prototype tool automatically identifies the patterns in the data set. The prototype tool is built using GUESS commands to look for certain characteristics, such as the number of edges attached to a node and the thickness of edges. Such characteristics enable the automatic detection of instances of the patterns in the graphs. For example, to find strongly interconnected problems in the problem graph, the prototype tool isolates problems that have the thickest edges connecting them compared to other adjoining problem nodes. Looking at Figure 4, Figures 4b) and 4c) shows instances of two patterns from the visualization of version A in Figure 4a). Similarly instances of patterns in version D are depicted in Figure 5b) and 5c) based on Figure 5a).

B. Identified Patterns

We automatically identify five patterns common to all four versions using the three types of graphs. We summarize the patterns using the following template:

- **Symptom:** describes the characteristics of the pattern in the graphs which first caught our attention.
- **Examples:** show examples from the studied application.
- **Significance:** explains the benefits of identifying the pattern.

B.1 Interconnected Problems

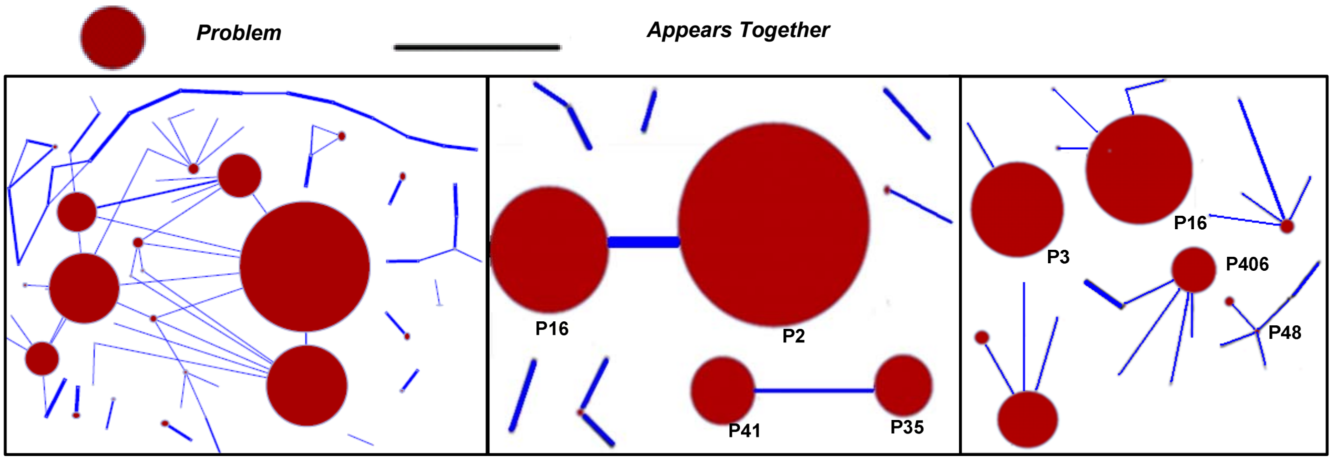
Symptom. This pattern is found in the problem graphs. The pattern is characterized by a thick (i.e., statistically significant) edge connecting adjacent problem nodes. The sizes of connected nodes can vary. This indicates that one problem is reported soon after another problem, suggesting a chain reaction. More than two problem nodes can be interconnected with thick edges. This indicates a longer chain effect among the interconnected problems.

Examples. Figure 4b) shows the identified interconnected problems pattern for version A. In the graph, we found that problem pairs {$P_{2,}$ P_{16}} and {$P_{35,}$ P_{41}} are separate examples of the interconnected problems pattern with their thick edges between each of the two node pairs. Similarly, Figure 5b) visualizes the interconnected problems pattern recognized in version D. For example in Figure 5b), the interconnected problems pattern is exhibited by the thick edges, such as {$P_{365,}$ P_{366}} and {$P_{702,}$ P_{708}}. The node size indicates the frequency of the problem being reported by different users. If the problems are not frequently reported, fixing such interconnected problems might not be a high priority. For example, as illustrated in Figure 5b), the node sizes of problems P_{365} and P_{366} are much bigger than other problems. Therefore, fixing problems P_{365} and P_{366} is a higher priority than fixing problems P_{702} and P_{708} because P_{365} and P_{366} are reported by a larger number of users.

Significance. Examining a particular problem might prove difficult to replicate or fix. However, that problem might co-occur frequently with another problem. For example, due to a chaining effect one problem might always be followed by another problem meaning one problem cannot occur without the other occurring first. Problems that are intertwined in this fashion indicate a series of chain effects that eventually lead to a final problem. It is desirable to find these patterns to eliminate the initial problem so it cannot propagate.

B.2 Near Duplicate Problems

Symptom. This pattern is recognized in the problem graphs. It is characterized by large problem nodes that have many smaller problem nodes attached to them. In effect the large node and its smaller neighboring nodes are essentially the same problem (i.e., they are nearly duplicates of each other). However, when users alter their usage slightly, the two problem reports are recognized as different problems.

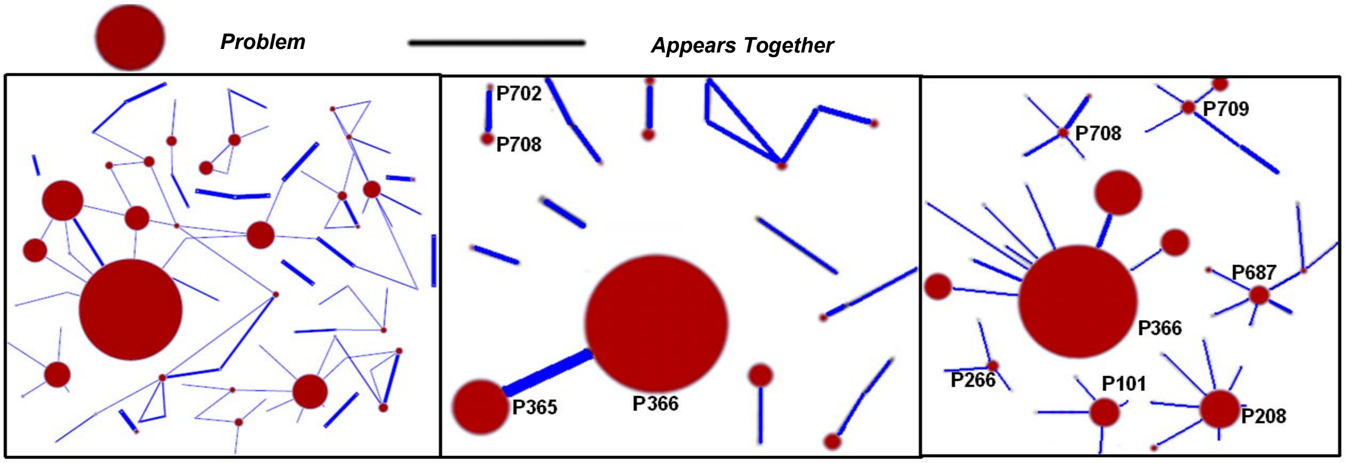


a) Filtered Problem Graph Before Pattern Identification

b) Interconnected Problem Pattern Extracted

c) Near Duplicate Pattern Extracted

Figure 4. Visualization of the Problem Graphs for Version A.



a) Filtered Problem Graph Before Pattern Identification

b) Interconnected Problem Pattern Extracted

c) Near Duplicate Pattern Extracted

Figure 5. Visualization of the Problem Graphs for Version D.

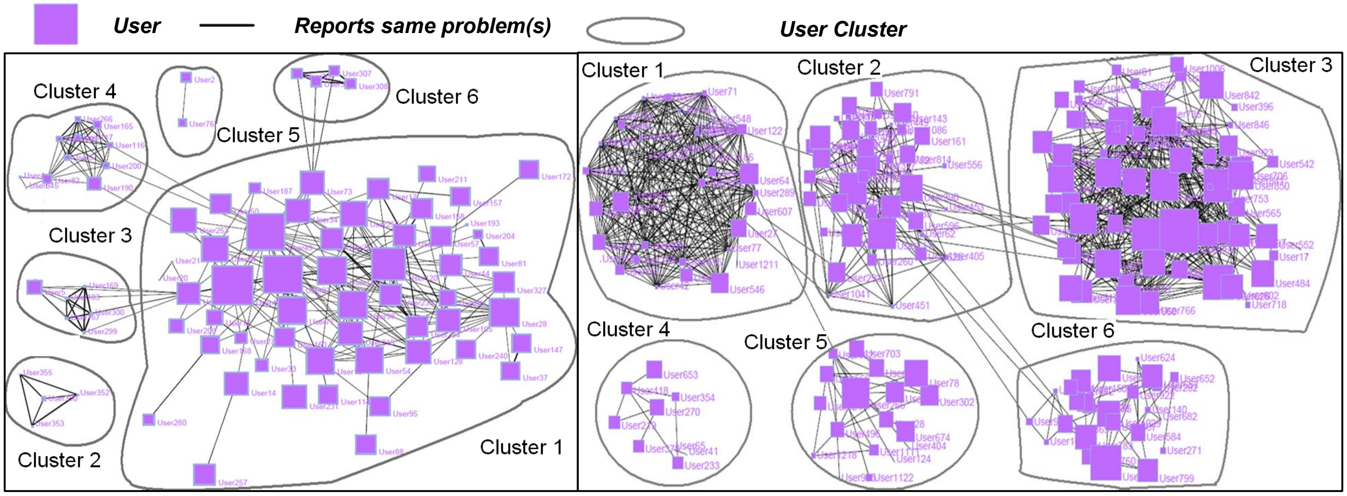
Examples. Figures 4c) and 5c) show the near duplicate problem patterns identified from both versions. For example, looking at Figure 4c), $P_{3,}$ $P_{16,}$ P_{48} and P_{406} represent nodes with the duplicate problems. Such problem nodes have smaller nodes that branch off them. Similarly, the problem graph of version D, shown in Figure 5c), has instances of this pattern which centers around the problem nodes, such as $P_{101,}$ P_{208} and P_{366} and P_{687}.

Significance. Often during field testing, a set of problems are reported infrequently. Through close investigation, we might determine that the set of problems is a special case of a frequently-occurring problem (i.e., the near duplicates of each other). In this case, it is desirable to assign the investigation of both problems to the same developer.

B.3 Distinct User Cluster

Symptom. This pattern is found in the user graph. It is characterized with large clusters of user nodes that are tightly connected. The clusters are formed by applying the spring-based layout algorithm which groups the user nodes with stronger association closer. The statistical filtering and weighting described in Section II.D ensure that the clusters are statistically significant and not simply due to chance. The clusters are visually identified from the user graph.

Examples. Figure 6 shows the user graph for both versions. We note that each version manifests its own distinct clusters of user nodes. Version A has a large central cluster (i.e., cluster 1) with several splinter clusters. It indicates that the majority of users report the same set of problems and are indistinguishable by the problems they



a) User graph for Version A

b) User graph for Version B

Figure 6. Visualization of the User Graphs for Both Versions.

report. Version D has six more distinctive clusters. It shows that the problems among users are more distinctive.

Significance. Identifying user clusters based on common problem profiles helps practitioners in the management of problems through interviews and additional instrumentation. Furthermore, the representative users could be used to replicate problems and verify fixes before they are deployed more widely during the field testing. The identification of user clusters could also help managers in planning future field testing efforts. Managers can ensure that each field test run has representative users from each cluster to guarantee a wider and more general testing of an application.

B.4 Distributed Problem Coverage

Symptom. This pattern is uncovered by analyzing the interaction graph. It is characterized with a problem node having a large number of adjoining user nodes. To prioritize the fixing of problems, we identify the problems reported by a large number of users. If a problem node is attached to a large number of user nodes, it presents a wide spread problem. The thicknesses of the edges show the distribution of the problem occurrence across the users. If we can identify a very thick edge to a particular user, this indicates that this problem primarily occurs for that user. If a problem node is attached to a small number of user nodes, the problem is not widely spread. Part of the analysis also involves examining the users who report the problem to determine if they report a large number of problems.

Examples. Figures 7b) and 8b) show the distributed problem coverage patterns identified from the interaction graph for versions A and D respectively. Looking at the graph for version A depicted in 7b), we note several instances of this pattern, such as the clusters around the problems, P_{16}, P_{406}, and P_{408}. For example, problem, P_{16} and P_{408} are reported by almost the same set of users. The interaction graph of version D illustrated in Figure 8b)

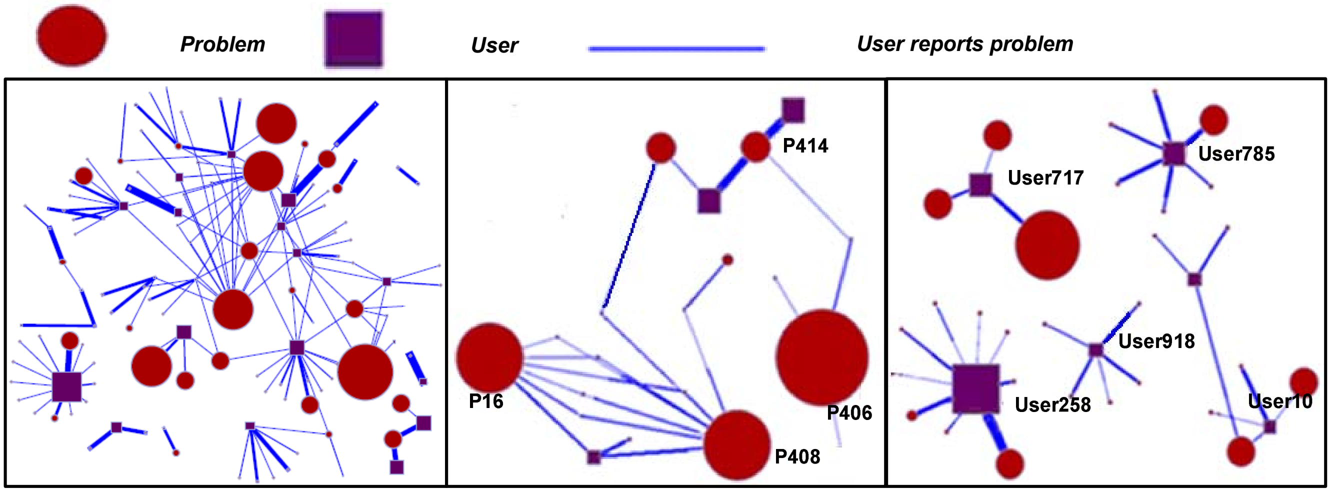
current field testing efforts and the planning of future efforts. Developers can work closely with representative users of clusters to gain a better understanding of their

contains many instances of this pattern, such as the clusters around the problem nodes, $P_{81,}$ P_{172}, $P_{183,}$ $P_{232,}$ $P_{422,}$ P_{676}, and P_{678}. As a result, these problems are good candidates to fix first.

Significance. Prioritizing which problems to address first is often a complex decision involving the criticality of the problem and its widespread impact on the user base. Given two problems that occur frequently, it is often desirable to fix problems that impact a larger number of users evenly. Counting the number of users that encounter the problem is a good metric. However, the metric fails to capture the frequency of the occurrence of the problem. For example, given a problem which occurs 100 times, it could be the case that the problem occurs 80 times for a single user and very infrequently for another 20 users. Or it could be that the problem occurs evenly across the 21 users. A visualization that shows this distribution help in the prioritization of problems. Moreover we must consider each user that has this problem in the context of the other problems. Referring back to our previous example of a user with a problem being reported 80 times, if we notice that this user is reporting a large number of other problems, it might be the case that this user's installation or environment has issues and that the problem is not as frequently occurring as we thought.

B.5 Distributed User Coverage

Symptom. This pattern shows up in the interaction graph with a large user node having many adjoining problem nodes. This pattern appears visually as a star-like shape with a center user node and problems radiating out of that user node.

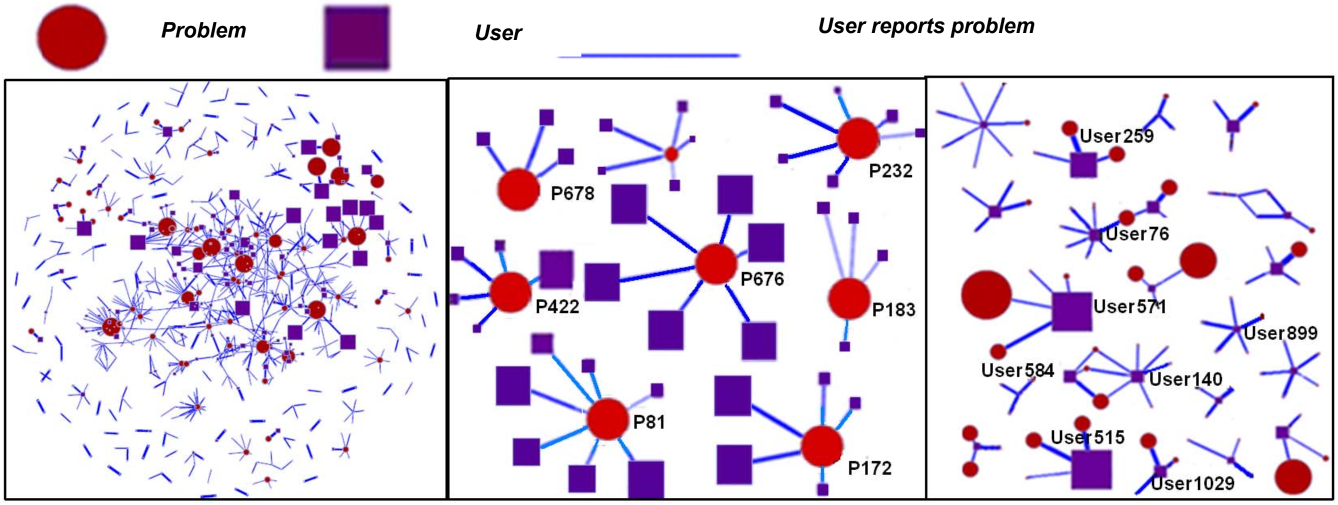


a) Filtered Interaction Graph before Pattern Identification

b) Distributed Problem Coverage Pattern Extracted

c) Distributed User Coverage Pattern Extracted

Figure 7. Visualization of the Interaction Graph for Version A.



a) Filtered Interaction Graph before Pattern Identification

b) Distributed Problem Coverage Pattern Extracted

c) Distributed User Coverage Pattern Extracted

Figure 8. Visualization of the Interaction Graph for Version D

Examples. Figures 7c) and 8c) show the distributed user coverage patterns extracted from both versions. As illustrated in Figure 7c) for version A, the clusters around user nodes, *User*$_{10}$, *User*$_{258}$, *User*$_{717}$, *User*$_{785}$, and *User*$_{918}$, are examples of this pattern. Similarity, Figure 8c) for version D contains many instances, such as the clusters around user nodes, *User*$_{76}$, *User*$_{140}$, *User*$_{259}$, *User*$_{515}$ and *User*$_{571}$.

Significance. Users reporting a large number of problems are ones that developers should examine closer. The user graph shows such users as large nodes. However, the mapping from the user to specific problems is not visible in the user graph. We want to better understand the distribution of problems generated by a particular user. In the same way we examine the distribution of a particular problem across different users. By exploring these users, we can then study if their applications are misconfigured and whether their results should be ignored. Or if they are facing a large number of problems and are ideal users to work with closely. In essence, this pattern helps reduce the time of developers in finding good testing candidates.

C. Accuracy of our Pattern Identification Approach

We manually verified the accuracy of the identified patterns for all four versions. We hope to achieve high accuracy so we can reduce the time wasted by developers as they

TABLE IV. STATISTICS VALIDATION OF PATTERNS FORM PROBLEM AND USER GRAPHS

Version	Interconnected Problem		Near duplicate problem		Distinct User Clusters		Distributed Problem Coverage	
	Accuracy	# of instances	Accuracy	# of instances	Probability of locating alternatives	# of instances	Accuracy	# of instances
A	70%	12	70%	10	92%	6	100%	6
B	69%	13	N/A	N/A	100%	2	100%	5
C	57%	7	50%	2	98%	3	100%	6
D	68%	22	60%	15	89%	6	100%	14

explore the identified patterns. For each problem report (i.e., node), we check its stack trace, source code, bug reports, change logs and severity reports to identify the logical relations among problems, among users and between users and problems. For example, to verify the interconnected problems pattern, we manually check the call stack associated with each of the problems (i.e. nodes shown in the problem graphs), to see if the problems are considered to be interconnected problems. For each type of patterns except the distinct user cluster pattern, the accuracy measures the percent of correctly identified instances of a pattern over the total number of identified instances. Due to the large number of users appearing in some of the user clusters, it is not feasible to manually review the logs for each user. We randomly select a percentage of the users from a cluster to verify if there exists another user which could serve as a replacement for problem replication. For clusters larger than 60 users, we test 10% of users, 30% for clusters with the size between 30 and 60, and 50% for clusters less than 30 users. We then calculate the average probability of locating alternatives in the same cluster. Table IV summarizes the results of the evaluation. We do not measure the recall for the identified patterns since measuring the recall requires knowing the total number of instances for each pattern. We would need to manually check every pair of nodes for each pattern. This is unfeasible task given the large size of the data.

Table IV shows that approximately 66% of the instances of the interconnected problems pattern are verified to be related. Some interconnected problems were found not to be related after manual verification. For example, examining the stack trace we find that the problem was due to two different unrelated modules, but coincidentally, the problems occurred with comparative frequency and time.

We found that on average 60% of identified near duplicate problems were manually verified to be duplicates. The misidentified duplicates problems are often connected to a central problem without sufficient similarity. For example, the misidentified duplicate problems are connected to the central problem, but have very different functionality.

For the distinct user cluster pattern, the average probability of locating alternative users in the same cluster is high for all four versions. Nevertheless, users in the same cluster do not always report the exact same set of problems to each other. Therefore, a given problem reported by one user may not be reported by a user who is randomly picked from the set of the connected users in the cluster.

Our prototype tool can accurately locate the distributed problem coverage pattern with 100% accuracy. To verify the accuracy of the identified instances of this pattern, we compare the identified problems with the high severity reports that are filed during the development of these versions. We found that all problem nodes map to high severity reports.

To verify the accuracy of the identified distributed user coverage pattern, we would need detailed information about each user's configuration so we can determine the rationale for the user reporting such large variety of problems. Unfortunately, such information is not available in our data set. Therefore we are not able to verify the accuracy of the distributed user coverage pattern.

IV. RELATED WORK

Studying the operational distribution of a program is an important and practical area of research proposed by Musa [11]. Different users exercise different subsets of the features provided by an application. User profiles can be built by instrumenting the in-field use of an application. These profiles can be used to compare different users using various similarity and dissimilarity metrics (e.g. [2] and [10]). Using this knowledge one can reduce the number of test cases and determine similar user groups. When studying large distributed software applications, detailed instrumentation is not feasible due to the high overhead. Such detailed instrumentation consumes extensive resources and produces a large amount of data which is challenging to transmit back to central repositories for further analysis. In our work we analyze the already-collected data, the in-field problem reports, to derive an approximation of the operational distribution of an application (i.e., its problem profile). In contrast to an operational profile, users with different usage patterns might not have the same problems.

Various approaches are used to study and compare different profiles. Sarbu *et al.* [15] use techniques to profile device driver behavior using temporal metrics obtained for I/O traffic characterization. They aim to improve testing on real-life workloads and test their technique on actual Windows drivers. Dickinson et al. [3] use clustering analysis techniques to study the relation between different operational profiles (e.g., users) using mathematical techniques. Orso *et al.* [13] and Jones *et al.* [8] use visualization techniques to study the similarity of profiles by mapping them to the lines of code and visualizing the relation between the different lines to identify the location of faults. Our approach visualizes relations at a higher level of abstraction (i.e., at the level of field problems and users) instead of lines of code. Our analysis could be also performed at the level of lines of code; however this would require a more detailed instrumentation of the application.

Christmansson *et al.* inject software errors directly into an application and observe the outcome [1]. Such fault injections approaches could be used to verify the soundness of the clusters in our visualization by checking if a representative user of a cluster would exhibit similar problems to other users in the clusters.

In our work, we use spring-based layout algorithms to produce clusterings of field reports and users. There is a large and active area of research in the use of clustering techniques to understand software artifacts [7, 12, 14, 16, 17, 18]. In contrast, our work is primarily visual whereas the aforementioned techniques produce textual output representing the different clusters.

Prior work which visualizes relations between software artifacts (e.g., Harald *et al.* [6]) employs basic threshold filtering techniques. For example, only edges above a specific threshold are shown to reduce clutter and avoid over-plotting. Our approach uses a statistical filtering algorithm. The statistical filtering is possible due to the large size of the data used in our case study.

V. Conclusion and Future Work

We present three types of graphs for visualizing the field testing results. The graphs provide a high level view of the large amount of field testing results. We describe five patterns which help managers and developers make best use of the results of the field testing and improve future field testing efforts. We automate the identification of patterns to enable the analysis of large scale data for practitioners. As all visualizations, the graphs lack the information needed for detailed analysis. However, the visualization flags important and interesting patterns out of the large data at hand.

In the future, we plan to conduct an empirical study to investigate the improvement in field testing effort using the proposed approach. We expect the need to extend our catalog of patterns as we study multiple versions of different applications.

Acknowledgments

We would like to thank Dr. Bram Adams, from Queen's University, for his valuable feedback on our work and his GUESS expertise.

We are grateful to Research In Motion (RIM) for providing access to the data used in the case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of RIM and/or its subsidiaries and affiliates. Moreover, our results do not in any way reflect the quality of RIM's software or hardware products.

References

[1] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. International Symposium on Fault Tolerant Computing, 1996, pp. 304-313.

[2] W. Dickinson, The Application of Cluster Filtering to operational testing of Software. Doctoral dissertation. Case Western Reserve University. May 2001.

[3] W. Dickinson, D. Leon, and A. Podgurski, Finding failures by cluster analysis of execution profiles. The International Conference on Software Engineering 2001, pp.339- 348.

[4] GUESS: The Graph Exploration System. http://graphexploration.cond.org/. Last access on June 2009.

[5] J. P. Guilford. The phi coefficient and chi square as indices of item validity. Springer. 1941.

[6] G. Harald, J. Mehdi, R. Claudio: Visualizing Software Release Histories: The Use of Color and Third Dimension. International Conference on Software Maintenance, 1999, pp.99-108

[7] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. IEEE Transactions on Software Engineering, Aug. 1985, pp. 947-757.

[8] J.A. Jones, M.J. Harrold and Stasko, J. Visualization of test information to assist fault localization. The International Conference on Software Engineering 2001, pp. 467-477, 2001.

[9] H.O. Lancaster. Chi Squared Distribution (Probability & Mathematical Statistics). Wiley. 1969.

[10] D. Leon, A. Podgurski and White, L. J. 2000. Multivariate visualization in observation-based testing. The International Conference on Software Engineering, 2000, pp. 116-125.

[11] J. D. Musa, Operational Profiles in Software-Reliability Engineering. IEEE Software. 10 (2), March 1993, pp.14-32.

[12] M. G.H. Omran, Andries P. Engelbrecht, A. Salman. An overview of clustering methods. Intelligent Data Analysis Vol 11 Issue 6, December 2007, pp. 583-605.

[13] A. Orso, D. Liang, M.J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. The International Symposium on Software Testing and Analysis, 2002, pp. 65-69.

[14] P. Pantel, D. Lin. Document clustering with committees. The Annual International ACM SIGIR Conference on Research and development in information retrieval. 2002, pp.199-206.

[15] C. Sarbu, A. Johansson, N. Suri, N. Nagappan. Profiling the Operational Perhavior of OS Device Drivers. The International Symposium on Software Reliability Engineering. 2008, pp.127-136.

[16] V. Tzerpos, R.C. Holt. Software Botryology: Automatic Clustering of Software Systems. The International Workshop on Program Comprehension, 1998, pp. 811-818.

[17] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. The Working Conference on Reverse Engineering, 1997, pp. 33-43.

[18] S. Zhong, J. Ghosh. A unified framework for model based clustering. The Journal of Machine Learning Research, Vol 4, 2003, pp. 1001-1037.