Improving Bug Triage with Bug Tossing Graphs

Gaeul Jeong *
Seoul National University
gejeong@ropas.snu.ac.kr

Sunghun Kim †
Hong Kong University of Science and Technology
hunkim@cse.ust.hk

Thomas Zimmermann
Microsoft Research
tz@acm.org

ABSTRACT

A bug report is typically assigned to a single developer who is then responsible for fixing the bug. In Mozilla and Eclipse, between 37%-44% of bug reports are "tossed" (reassigned) to other developers, for example because the bug has been assigned by accident or another developer with additional expertise is needed. In any case, tossing increases the time-to-correction for a bug.

In this paper, we introduce a graph model based on Markov chains, which captures bug tossing history. This model has several desirable qualities. First, it reveals developer networks which can be used to discover team structures and to find suitable experts for a new task. Second, it helps to better assign developers to bug reports. In our experiments with 445,000 bug reports, our model reduced tossing events, by up to 72%. In addition, the model increased the prediction accuracy by up to 23 percentage points compared to traditional bug triaging approaches.

Categories and Subject Descriptors

D2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; K6.3 [**Management of Computing and Information Systems**]: Software Management – Software maintenance

General Terms

Measurement, Reliability, Experimentation, Human Factors.

Keywords

Bug report assignment, Bug triage, Bug tossing, Issue tracking, Machine learning, Problem tracking

*She is supported by the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2009 and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / Korea Science and Engineering Foundation (KOSEF), grant number R11-2008-007-01002-0.

†This project was initiated when he was at Seoul National University as a postdoc.

ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.

1. INTRODUCTION

The timely identification and correction of bugs are very important software engineering practices. To handle a large number of bugs, bug tracking systems such as Bugzilla [9] are widely used. However, most bugs are assigned manually to developers, which is a labor-intensive task, especially for large software projects. For example, the Eclipse and Mozilla projects receive several hundred bug reports per day and assign each of them to one of the several thousand developers. This is not an easy task and is often error-prone.

Once a bug report has been assigned, developers can reassign the bug to other developers; we call this process *bug tossing*. For this paper, we studied the assignment and tossing activities for 450,000 bug reports from Eclipse and Mozilla (Section 2). We found that 37%-44% of bugs have been tossed at least once to another developer. One of the common reasons for bug tossing is that bugs are sometimes assigned to developers by mistake. For example, a developer might not own the defective code ("not mine") or might not have the expertise to fix the bug. When tossing a bug report, it is often unclear who is the correct person to fix the bug ("who should the bug go to next?"). Another common reason for bug tossing is to include developers with additional expertise in the discussion of the bug report. In any case, many tossing events generally slow down progress when fixing bugs because, as we found in our study, a bug tossing event takes an average of 50 days.

This paper introduces a tossing graph model which is based on the Markov property (Section 3). The proposed model captures tossing probabilities between developers from the tossing history available in bug tracking systems. This graph model has two desirable qualities:

1. *Discovers developer networks and team structures.*

   We showed the tossing graphs to the Mozilla and Eclipse developers and received positive feedback. They confirmed the graphs are useful for bug processing tasks.

2. *Helps to better assign developers to bug reports.*

   In our experiments with the 450,000 bug reports from Mozilla and Eclipse, our model could reduce the amount of bug tossing substantially. In addition, our graphs increased automatic bug assignment accuracy by up to 23 percentage points.

We believe the proposed tossing graph model provides useful and actionable information to improve bug report processes.

This paper makes the following contributions:

Empirical study of bug report assignment and tossing. We analyzed detailed activities of bug reports. Based on the analysis, this paper provides detailed statistics about bug assignment and tossing.

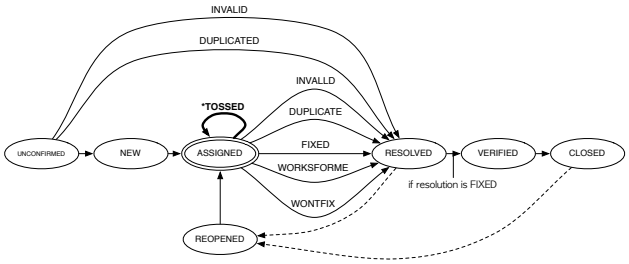


Figure 1: The life cycle of a bug report [30]. The **TOSSED* cycle is added and the main focus of this paper.

Novel graph model for bug tossing. To our knowledge, this is the first time that bug tossing is used to reveal team structures and to improve bug report processes.

Evaluation of the bug tossing graphs. We present results from the analysis of 450,000 bug reports, which show that our model is useful for discovering team structures, reducing the number of tossing events, and bug assignment tasks.

The remainder of this paper is organized as follows. Section 2 shows statistics related to bug assignment and tossing. Section 3 presents our tossing graph model and Section 4 shows the usefulness of our model by presenting the results from the experiments. Section 5 discusses the results and the limitations of our study. Section 6 surveys related work and Section 7 concludes this paper.

2. BUG REPORT ANALYSIS

This section presents the bug assignment and tossing analysis results to provide understanding of bug report processes and properties.

2.1 Subject Systems and Bug Reports

We analyzed the first 145,000 bug reports from Eclipse (starting with bug id 5,001 and ending with id 150,000) and the first 300,000 bug reports from Mozilla. At the time of this writing, there are about 267,000 bugs in Eclipse and 482,000 bugs in Mozilla. Since our analysis is observing bug assignment and tossing processes, we only use bugs which are old enough to be assigned and processed.

We excluded Eclipse bug reports from bug id 1 to 5,000. Their creation dates are all the same. These bug reports have been migrated from an OTI legacy system and their creation dates are perhaps not properly set. As a result, the developer assignment date of some reports is earlier than the bug creation date. In our research, the time spent between bug report creation and assignment is a critical factor, so we excluded these suspicious bug reports.

Most bugs have a common lifecycle identified by Zeller [30] as depicted in Figure 1. A user or developer finds a problem and then reports it using a bug tracking system. Then a manager or developer of the corresponding project reads the report description and assigns the bug to the proper developer. Then the bug will be fixed by the developer and verified by other developers or managers. Finally, the bug report will be closed.

Bug tracking systems record these detailed activities. Table 1 shows the activities of an Eclipse bug. The activity information includes time of bug assignment, assigned bug developer, the time of bug status changes, and actors of each activity. By analyzing these activities, we identify interesting properties about bug report processes. For example, Eclipse has more than 1,200 developers and Mozilla has more than 2,400 developers, who are assigned to more than one bug report.

Table 1: Example of activities of an Eclipse bug report.

Who	When	Action
akiezun@mit.edu	2001-10-12	Assigned to Mike_Wilson@oti.com
Mike_Wilson@ca.ibm.com	2002-05-23	Status to RESOLVED
		Resolution to LATER (deprecated)
veronika_irvine@ca.ibm.com	2002-09-11	Assigned to Steve_Northover@oti.com
		Status to NEW
steve_northover@ca.ibm.com	2004-04-09	Status to RESOLVED

`https://bugs.eclipse.org/bugs/show_activity.cgi?id=4425`

Bug tossing events are also recorded in the activity information. As shown in Table 1, after the bug is assigned to Mike_Wilson, the developer reassigns the bug to another developer (Steve_Northover). We found this is a common practice in Eclipse and Mozilla. This practice is not described in the Zeller's lifecycle [30], and we revise it by adding the *TOSSED* cycle in Figure 1.

Most bug tracking systems including Mozilla use an email address as developer identification. However, it is possible that a developer use more than one email address for bug tracking systems. For example, a bug report is tossed to *Mike_Wilson@oti.com*, then *Mike_Wilson@ca.ibm.com* works on the bug as shown in Table 1. In fact, *Mike_Wilson@oti.com* and *Mike_Wilson@ca.ibm.com* are email addresses of the same developer. We use only the name part of email to avoid considering the same developer as two or more different developers. For example, we use *Mike_Wilson* as developer identification. This technique is commonly used for bug report and email mining [6, 7].

2.2 Bug Assignment

We noticed 426 Eclipse bugs are reported in one day. Similarly, Mozilla received 390 bug reports on Nov 22, 1999. Assuming that all bug processes are manual, how long would it take managers to perform an action on a new bug or to assign the bug to developers?

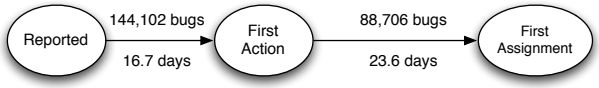
As shown in Figure 2(a) and Figure 3(a), the bug report process takes a long time. We observed the time spent between bug creation and the first action which is any action taken by a manager such as changing the status from *UNCONFIRMED* to *NEW*. Then we measured the time spent between bug creation and assignment. The first action on an Eclipse bug took 16.7 days on average. For Mozilla, it took 26.1 days on average. The bug assignment also takes a long time. After the first action, it takes 23.6 more days for Eclipse and 161.1 more days for Mozilla.

Next, we examine the time spent for the first action and assignment of *only verified bugs* as showin in Figure 1. Some bug reports do not have enough information to be fixed, and they remain unassigned [4] for a long time. Figure 2(b) and Figure 3(b) shows the time spent on verified bugs. It takes 5.2 days in Eclipse and 7.1 days in Mozilla for the first action on verified bugs. The bug assignment task then takes 19.3 days for Eclipse and 38.1 days for Mozilla. Overall, the time is reduced, but still it takes a long time to assign verified bugs.

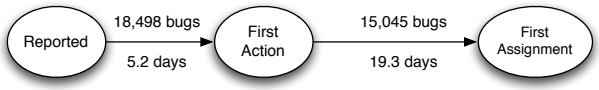
One of the main reasons for these slow actions and assignments is that bug reports are manually processed. Managers have to check bug reports one by one, understand the causes of the bugs, and match them to the proper developer. It would be desirable to get some assistance from automated bug assignment algorithms.

2.3 Bug Tossing Paths

In this section, we present statistics related to bug tossing. We first formally define the bug tossing concepts. We describe bug

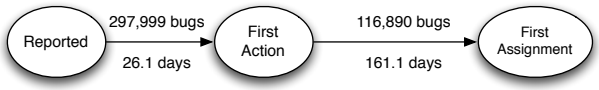


(a) All Bugs

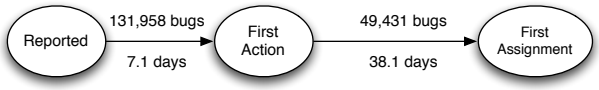


(b) Verified Bugs

Figure 2: Among the 145,000 bug reports from Eclipse, 144,102 bugs have at least one action and 88,706 bugs are assigned. On average, a bug takes 16.7 days to have the first action and 23.6 days to be assigned after the first action.



(a) All Bugs



(b) Verified Bugs

Figure 3: Among the 300,000 bug reports from Mozilla, 297,999 bugs have at least one action and 116,890 bugs are assigned. On average, a bug takes 26.1 days to have the first action and 161.1 more days to be assigned.

tossing as follows. A tossing process starts with one developer, say d_1, and moves from one developer to another until it reaches the *fixer*, d_f, the developer who fixed the bug. Each move is called a *tossing step* and a set of finite tossing steps $T = \{d_1, d_2, ..., d_f\}$ of a bug is called a *tossing path*.

A *tossing interval* denotes the time spent between two consecutive elements in a tossing path. A tossing path must have one fixer. The number of elements in a tossing path, $|T| - 1$ is called *tossing length*. If a tossing path, $T = \{ d_1 \mid d_1 = d_f\}$ has only one element, it has no or *zero tossing*. The bug was assigned to a developer, and she fixed the bug, i.e., no tossing event occurred.

We observed the tossing length of each bug. Figure 4 shows the Eclipse bug distribution based on their tossing length. For example, about 8,400 bugs (56%) have only one assigned developer, and no tossing event. However, 4,200 bugs (28%) have a single tossing event. Overall, about 44% of bugs have at least one tossing event.

Similarly, Figure 5 presents the Mozilla bug distribution, which is similar to that of Eclipse. About 37% of bugs have at least one tossing event. These results indicate that a large number of bug reports have tossing events and thus tossing is a common practice in the bug fixing process.

We observed the tossing intervals of bug reports. Figure 6 shows the average tossing interval at each tossing length. For example, in Eclipse it takes about 40 days on average to assign a bug to the first developer, and then it takes more than 100 days on average to reassign the bug to the second developer. Mozilla is worse. Surprisingly, it takes almost 180 days, a half year, on average for the first assignment and takes more than 250 days for the first tossing. Bug reports which do not have enough information take a long time to be assigned [4]. To eliminate these bugs from the time spent measurement, we observed tossing intervals for only verified bugs as shown in Figure 7. The average tossing intervals are reduced, but still it takes a long time, 10 to 60 days, to assign a bug.

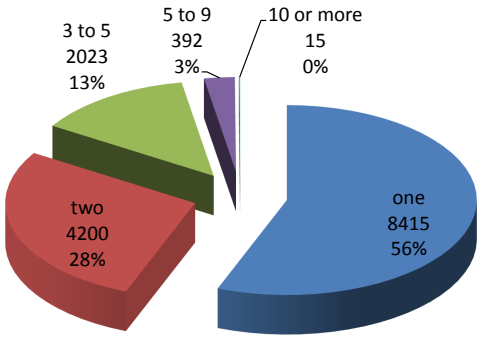


Figure 4: Eclipse bug report distributions based on the number of assigned developers. Only assigned and verified bugs are considered. About 56% of bugs are assigned to a single developer; 44% of bugs are assigned to more than one developer and have tossing events.

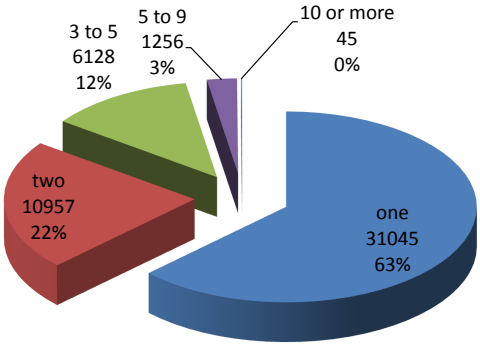


Figure 5: Mozilla bug report distributions based on the number of assigned developers. Only assigned and verified bugs are considered. About 37% of bugs have tossing events.

The results in this section indicate that these long tossing intervals could delay bug fixing processes. We believe that tool support can to remove unnecessary tossing steps and/or reduce the tossing intervals. Our proposed tossing graph model can reduce tossing steps and tossing intervals.

3. TOSSING GRAPH MODEL

In this section, we describe our tossing graph model.

3.1 Tossing Model

Recall that we defined a tossing path as a set of developers, $T = \{d_1, d_2, ..., d_f\}$ in Section 2.3. There are various ways to get tossing properties from tossing paths.

Suppose we have a tossing path, $A \rightarrow B \rightarrow C \rightarrow D$. The bug is fixed by D, the fixer. A simple way to obtain tossing properties is to consider every single step in the path, $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow D$. This model is called the *actual path model*. Formally, for a given tossing path T, we define the actual path model as a set of steps, $P = \{ d_i \rightarrow d_{i+1} \mid d_i \in T \ \& \ d_i \neq d_f \}$.

However, to quickly find or predict the bug fixer D from a given developer, we can use a model that decomposes the given paths to goal oriented steps, $A \rightarrow D$, $B \rightarrow D$, and $C \rightarrow D$. For a given tossing path T, we define the *goal oriented model* as a set of single

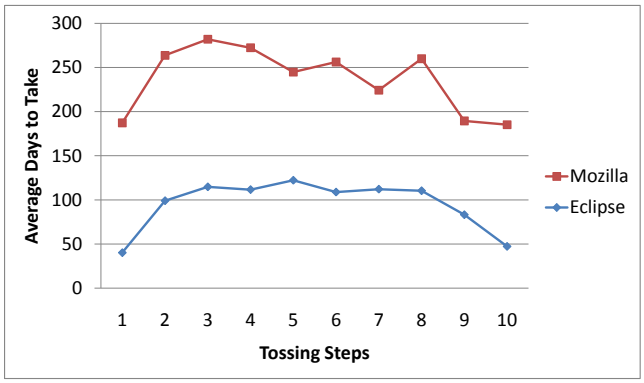


Figure 6: Average time spent per tossing step for all bugs.

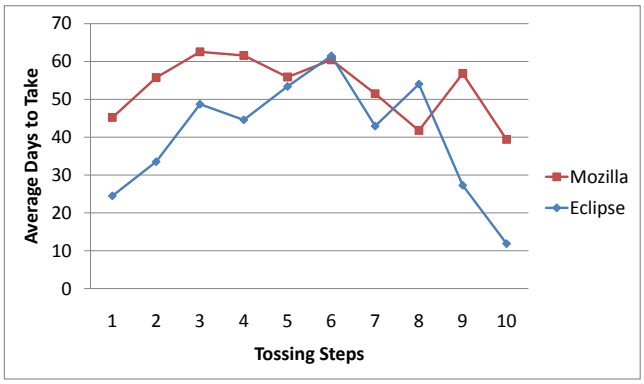


Figure 7: Average time spent per tossing step for verified bugs.

Table 2: Simple tossing paths.

$A \rightarrow B \rightarrow C \rightarrow D$
$A \rightarrow C \rightarrow D \rightarrow E$
$C \rightarrow E \rightarrow A \rightarrow F \rightarrow D$

steps, $G = \{ d_i \rightarrow d_f \mid d_i \in T \ \& \ d_i \neq d_f \}$. This model encodes the relationship between intermediate developers and the fixer.

Table 2 lists sample tossing paths, and Table 3 shows the decomposed steps from the sample paths. Two different models yield different sets of individual steps. All steps in the goal oriented model end with D or E, since they are the fixers. The actual path model encodes all intermediate steps. The numbers in Table 3 represent the occurrences of the steps. For example, in the goal oriented model, there are two steps $A \rightarrow D$ and one step $F \rightarrow D$. We assume that more frequent steps are the more significant ones.

Which model is better? It depends on the purpose of using tossing graphs. In our experimental study, the goal oriented model outperforms for reducing paths, presented in Section 4.2, and for bug assignment prediction, shown in Section 4.3.

3.2 Markov Model

A Markov chain is a set of states, transactions, and transaction probabilities [15, 16]. It has a set of states, $S = \{s_1, s_2, ..., s_n\}$. A Markov process moves from one state to another with a certain probability, called a transaction probability. The Markov chain is valid if the next transaction probabilities of all states in the chains only depend on the current state (Markov property). It is also known as the memory-less transaction probability property, since

Table 3: Decomposed single steps from the sample tossing paths in Table 2 using two models. The numbers in parenthesis indicate the occurrences of each path.

actual paths	goal oriented paths
$A \rightarrow B$ (1), $B \rightarrow C$ (1), $C \rightarrow D$ (2), $A \rightarrow C$ (1), $D \rightarrow E$ (1), $C \rightarrow E$ (1), $E \rightarrow A$ (1), $A \rightarrow F$ (1), $F \rightarrow D$ (1)	$A \rightarrow D$ (2), $B \rightarrow D$ (1), $C \rightarrow D$ (2), $A \rightarrow E$ (1), $C \rightarrow E$ (1), $D \rightarrow E$ (1), $E \rightarrow D$ (1), $F \rightarrow D$ (1)

previous states do not influence the transaction probabilities of the current state.

Suppose the current state is s_n. In a Markov model, the probability to get to the next state s from the current state s_n is not affected by prior states. Formally, $Pr\{S_{n+1} = s_j \mid S_n = s_n, S_{n-1} = s_{n-1}, ..., S_1 = s_1\} = Pr\{S_{n+1} = s_j \mid S_n = s_n\}$ [15, 16].

It is not guaranteed that the Markov property always holds for bug tossing. For example, the tossing probability from X to Y in two paths, $A \rightarrow X \rightarrow Y$ and $B \rightarrow X \rightarrow Y$ might be different, since the prior states A and B of the paths are not the same. However, we believe that in the majority of cases, previous states do not influence the tossing probability from X to Y, i.e., it does not matter for tossing decisions whether someone received a bug from Alice or Bob. In this paper, to simplify our model, we use the Markov model to generate tossing graphs. In future work we will look at hidden Markov models, which also can account for previous states.

Assuming the Markov property holds, we decompose a given tossing path to several single steps described in Table 3. Then we determine transaction probabilities of each step. For example, consider single steps, $C \rightarrow D$, $C \rightarrow D$, and $C \rightarrow E$ which start with C in Table 3 (for the goal oriented model). From these steps, we determine the tossing transaction probabilities as $C \rightarrow D$ to 67%, and $C \rightarrow E$ to 33%.

Figure 8 and Figure 9 show two tossing graphs generated from steps in Table 3. These graphs provide immediate picture of tossing relationship. For example, the tossing graph in Figure 9 indicates D is a good candidate to toss a bug from A, since in history they have stronger tossing relationship than those of others. As a developer reassigns bugs, this graph provides hints to identify appropriate developers.

3.3 Model Options

In Section 3.1, we proposed two options, actual and goal oriented path models to decompose tossing paths. The actual path model captures individual tossing relationships while the goal oriented model records the fixer from every developer in a tossing path.

Another option to shape tossing graphs to limit the minimum probability of transactions. Based on the frequency of previous tossing paths, some transactions have a high probability, and some have a low probability. We assume the high probability transactions are likely to happen in the future. It is possible to set a threshold on the transaction probability and exclude all transactions whose probability is lower than the threshold.

The value of minimum support is another option. Suppose we found one tossing step, $X \rightarrow Y$ in the entire tossing history. Since the tossing happened only once, the transaction probability from X to Y is 100%. However, we do not have enough steps to support this case. On the other hand, if the tossing from X to Y happened more than 30 times, we may have enough evidence to believe that X will toss a bug to Y in the future. We use a minimum support

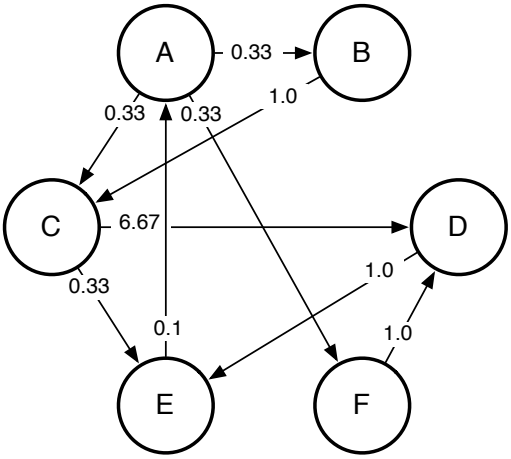


Figure 8: A tossing graph using the actual path model and decomposed steps in Table 3.

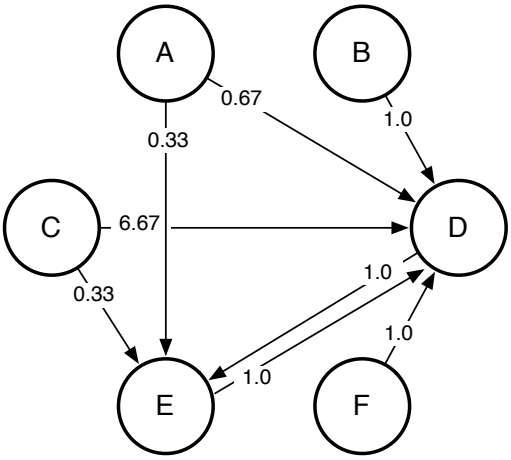


Figure 9: A tossing graph using the goal oriented path model and decomposed steps in Table 3.

value as an option to eliminate tossing paths which are not frequent enough and might be incidental.

To illustrate the options, we show a tossing graph of Eclipse with the goal oriented model, 0 minimum support, and 0 transaction probability options in Figure 10. This graph includes all tossing transactions in the history. Since some of them happened only a few times and have low transaction probability, this graph can be less useful to predict future tossing steps or discover developer groups. Predictions based on this graph may include noise as well.

Now, we changed the options to 25 minimum support and 15% transaction probability. The resulting tossing graph is shown in Figure 11. It only includes transactions which happened more than 25 times and where probabilities are greater than 15%. This graph is more useful to discover developer clusters which are clear in the graph. All transactions depicted in the graph are more likely to happen in the future than the transactions in Figure 10.

We tried various options to provide the best tossing graphs for the purpose of our experiments. The utilized options are listed with experiment results.

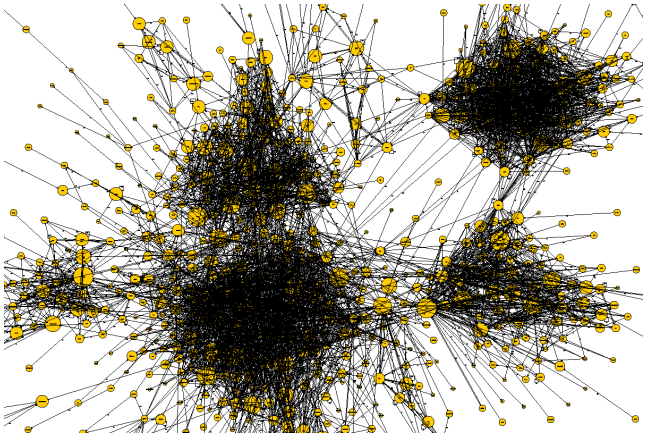


Figure 10: A tossing graph of Eclipse using bug report id 100,000 to 150,000. Goal oriented with 0 minimum support and 0% transaction probability options are used.

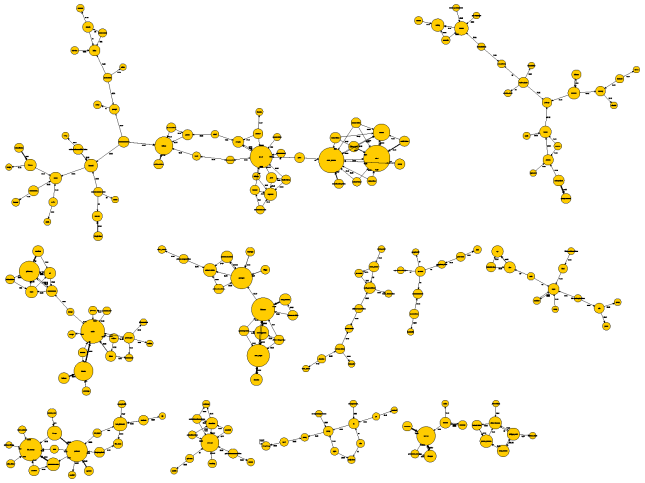


Figure 11: A tossing graph of Eclipse using bug report id 100,000 to 150,000. Goal oriented with 25 minimum support and 15% transaction probability options are used.

4. LEVERAGING TOSSING GRAPHS

This section demonstrates the usefulness of the tossing graphs.

4.1 Identifying Developer Structure

Recently, research focused on extracting developer structure within a project has received much attention. Goals for this research include understanding the developer structure, improving the development process, and locating faults [6–8, 22]. Previous approaches discover developer structures by mining co-change logs, email threads, or comment threads in bug reports.

Our bug tossing graphs reveal yet another kind of developer structure based on bug tossing. This structure is useful, since it represents the direct working relationship among developers. For example, if a manager wants to assign a new bug to a developer, she can consult the tossing graphs to identify the most likely developers. Similarly, if a developer finds that an assigned bug is not for her, and wants to reassign the bug to another developer, she can consult the graphs to find developers.

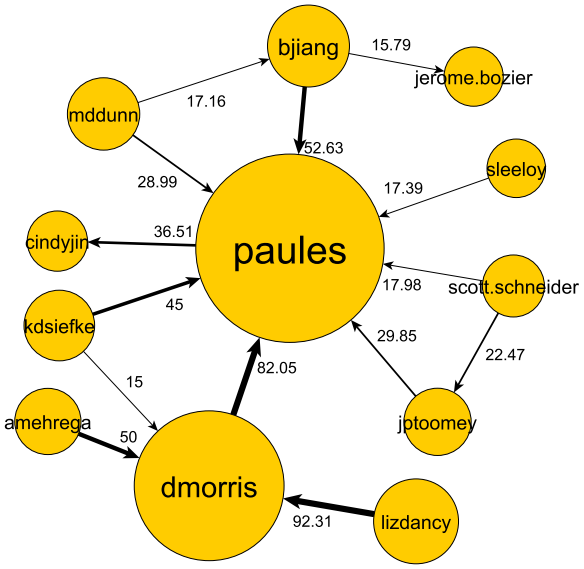


Figure 12: A partial tossing graph of Eclipse. Goal oriented with 25 minimum support and 15 transaction probability options are used. Nodes indicate developers and connecting lines (edges) represent tossing relationship. The numbers on edges show the tossing probability. The thick edges mean the corresponding edge has high tossing probability and the big nodes indicate they receive many bugs from others. For example, *paules* receives many bugs from other developers.

Figure 12 shows a partial tossing graph. Suppose you want to assign a bug to *lizdancy*. Then perhaps you want to consider assigning the bug directly to *dmorris* or *palues*, since many of *lizdancy*'s bugs are fixed by *dmorris* or *palues*. Or suppose you want to reassign a bug to *paules*, but he is on vacation and not available to fix the bug. Then you might want to consider reassigning the bug to *cindyjin*, since she fixed lots of bugs from *paules*. Tossing graphs and the revealed developer structure are very useful in such situations.

Perhaps an expert, who has lots of management and development experience, does not need help from the tossing graphs. She would know all developers and managers, and will be able to identify good developers for a new bug. However, usually this developer structure knowledge remains in the minds of experts and is rarely documented. Tossing graphs automatically capture this knowledge from history. Therefore, tossing graphs help new managers and developers, who do not yet have deep knowledge of the developer structure in a project.

To evaluate their usefulness, tossing graphs were generated for the Eclipse and Mozilla projects. These graphs were sent to the corresponding developers in these projects, with a request for feedback on the usefulness of the graphs. While the number of responses was not large enough to draw any substantive conclusions, the generally positive tone was a good initial reality check on the extracted tossing graphs.

Developer 1: "Very neat stuff! The clustering was correct for the ... team."

Developer 2: "Probably it can be used to tell who the major assignees of development are."

Developer 3: "Another interesting use for these types of graphs might be to gauge triage effectiveness."

Developer 4: "This would be useful for both integrators and managers wanting to understand the life cycle of bugs."

The information in the tossing graphs is useful, but it is challenging to utilize the static graphs. Perhaps providing tool support to search, zoom-in/out, and find suitable next developers would increase the usability of the graphs. This remains as our future work.

4.2 Reducing Tossing Paths

In Section 2, we showed that some bugs have a long tossing length, for example because some bug tossing steps were unnecessary due to mistakes or confusion during bug assignment process. It is desirable to avoid unnecessary tossing for timely bug fixing.

Could we reduce these long tossing paths by predicting proper developers? This problem, called *ticket routing* is a well-known problem in the machine learning literature [1, 24–26]. Various statistical models are used to improve ticket routing processes.

Inspired by one of the proposed algorithms in the machine learning literature [24], we applied their idea to Eclipse and Mozilla tossing events using our tossing graphs and a modified graph search algorithm. Similar to Shao et al. [24], we conducted experiments to evaluate whether our tossing graphs help reduce the tossing length.

First, we separated bug history into two folds, training and testing. A tossing graph is built using bugs in the training fold. Then we try to reduce tossing steps in the testing fold.

Suppose an original tossing path is A → B → C → D → E → F → G and G is the fixer. Our goal is predicting a path to get to G with fewer steps. For example, if we can find a path A → D → G using the tossing graph, this path is reduced to three steps from seven steps. The reduced steps allow bugs to be assigned to the optimal developers sooner, thus reducing the bug fixing time.

Beginning with the first assigner, we predict the best next developer by consulting the bug tossing graph. The search continues until we reach the fixer, or there are no more developers to recommend. If there are no developers to recommend, our search fails. In this case, usually the predicted path is long, since we traversed the graph to the end. This long predicted path would be undesirable. Our goal is to avoid dead ends and reduce the original paths.

Using a given tossing graph, we can imagine various algorithms to traverse and predict the next developer. We used tossing graphs built by the *goal oriented path model*. To utilize the graphs and maximize the path reduction, we use an algorithm based on weighted breath first search (WBFS).

The WBFS algorithm is introduced by Wang et al. [28] based on the breadth-first search algorithm (BFS) [12]. WBFS is similar to BFS except for the weight consideration. WBFS begins at a given node and explores heavy (or light) neighboring nodes first. In our case, we visit the node with the highest transaction probability first. We found WBFS works well with the goal oriented model. For a given node, since all neighboring nodes are fixers, WBFS reaches to most probable fixers first.

Using WBFS and tossing graphs we predict next developers. Suppose a bug is assigned to *lizdancy* in Figure 12. Our prediction for the next developer will be *dmorries*. The next after that will be *paules* based on the graph. In the same way, we predict future ones.

Then we compare the tossing length of the original paths and predicted paths. Figure 13 shows the predicted tossing length and the original tossing length of Eclipse and Mozilla. Overall our prediction reduced tossing length significantly, especially for the bugs with long tossing length. For example, 12 steps of Eclipse tossing length are reduced to less than 4 steps on average. Similarly, 9 steps of Mozilla tossing length are reduced to 2.5 steps, with a 72% reduction on average.

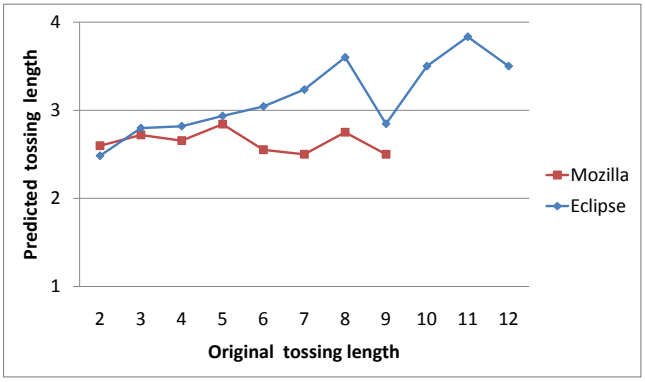


Figure 13: The reduced tossing length by predicting next proper developers using the tossing graph model.

For bugs with short tossing length, the improvement of our reduction algorithm is marginal or the predicted length is only slightly higher than that of the original. To avoid this situation, it is possible to apply our reduction algorithm only for bugs with long tossing length. For example, if a new bug is reported, managers or developers assign bugs manually. If the tossing length of the bug grows, then we apply this technique and consider predicted developers as the next assigner. The other approach is to use this reduction algorithm with a classifier. It is possible to build a binary classifier to predict whether a given bug would have long tossing length. Then we apply our reduction algorithm only for the bugs which are likely to have long tossing length. Building a binary classifier for our reduction algorithm is future work.

The tossing length reduction results depend on tossing graph options. Through experimentation, we found that using the goal oriented model with 10 minimum supports and 10% of transaction probability yields the best result. Search failure rates also depend on the graph options. Under the same option, about 10% of search failures are due to the missing start node. Since our search starts with a given developer, if the developer is new, or he does not have enough tossing history, the developer will not be on the graph. About 30% to 40% searches fail because the fixer is not in the neighboring nodes of the given node. We were not able to compare our failure rates with Shao et al. [24], since they do not report any search failure rates.

There is a tradeoff between the reduction rates and search failure rates, similar to the tradeoff between recall and precision [2]. If we put all developers into the tossing graphs by lowering the minimum support and transaction probability, the search failure rates will be lower. However, the path reduction rates will not be good, since, from a given node, there are many neighboring nodes to traverse. On the other hand, we can use tossing graphs with high minimum support and transaction probability. These graphs increase the search failure rates, since many developers are not in the graph. However, they only predict the next developer when there are enough supporting cases with high probability. We recommend using the tossing graphs with high values of minimum support and transaction probability so that graphs will predict next developers with high confidence. For uncertain cases, the graphs will quickly yield predictions rather than predicting long and wrong tossing steps.

4.3 Improving Bug Triage

Bug triage is a widely known problem in the software engineering literature, and many automatic bug assignment algorithms are proposed [3, 13]. They extract features from bug reports such as keywords and metadata. These features are used to train a machine learning model, and the trained learner predicts proper developers for new bugs.

If the prediction accuracy is high enough, these techniques are very useful in practice, since the resources to read bug reports and find appropriate developers are limited. The prediction accuracy of previous techniques is promising, but not yet satisfactory. Usually prediction accuracies are around 30% [5]. To increase the accuracy, previous algorithms select the first 3 or the first 5 candidates. For the first 5, prediction accuracy is around 60 to 70% [5].

Is the information in tossing graphs useful to improve the bug assignment accuracy? Consider this situation: developers d_1 and d_2 have a close tossing relationship. Developer d_1 usually tosses bugs to d_2 and d_2 fixes the bug. Similarly, d_1 fixes d_2's bugs. Suppose in the training set, a bug was fixed by d_1. Previous approaches would extract features and learn from the bug and its assignment. They assumed that similar bugs would be fixed by d_1 in the future. Now we have a new bug which is very similar to the previous bug. However, this new bug was fixed by d_2. This is a possible situation in real software development. Unfortunately, previous approaches that learned from previous bug reports would predict d_1 as developer to be assigned for the new bug, which is a wrong prediction. Since previous approaches do not utilize tossing information, they will miss this type of bug assignment.

Our approach is integrating bug tossing information into previous approaches. We obtain developer prediction lists from a previous machine learning technique. Suppose, for a given Eclipse bug, a previous approach predicts the first 5 developers, *{robert.elves, mik.kersten, wuamy, susan_franklin, nitind}*. Based on this prediction set, we try to identify more appropriate developers using the tossing relationship. Suppose that *robert.elves* and ***steffen.pingel*** have a strong tossing relationship. And *mik.kersten* and ***relves*** also have a strong relationship. Then we bring them into our new prediction list, and our first 5 prediction list would be *{robert.elves, **steffen.pingel**, mik.kersten, **relves**, wuamy}*.

In fact, this Eclipse bug was fixed by ***steffen.pingel***. Our prediction is correct while the previous approach fails to predict the fixer. The tossing graph information is useful to identify proper developers in this case.

Formally, we define our prediction algorithm as follows. For a given prediction set $P = \{ p_1, p_2, ..., p_n \}$ from existing approaches, our approach creates a new prediction set, $RP = \{ p_1, t_1, p_2, t_2, ..., p_n, t_n \}$ with t_i is the developer who has the strongest tossing relationship with p_i. From the new prediction set, we select first n developers. For example, for the first 5, our list would be a set, $\{ p_1, t_1, p_2, t_2, p_3 \}$.

To evaluate our approach, we use the bug triage benchmark (fold 10 of the master set) used by Bettenburg et al. [5] and two machine learning algorithms, Naïve Bayes and Bayesian Networks [2]. Bettenburg et al. used the first 165,385 Eclipse bugs as a training set [5]. They extracted string vectors in the bug report title and long description. The string vectors are used to train machine learners. The machine learners predict developers for bugs in a testing set, Eclipse bugs from number 165,397 to 211,822. Similarly, the first 345,343 Mozilla bugs are used as a training set, and bugs numbers 345,345 to 429,903 are used as a testing set. (The interested reader is encouraged to pursue Bettenburg et al. [5] for the detailed experiment description.)

We generated tossing graphs using bug reports in the training set. Then we obtained predictions from the machine learning approach [5]. From these prediction lists, we generate new prediction lists utilizing the tossing graphs. We select the best first 2, first 3, first 4, and first 5 from each prediction set and determine the

Table 4: Bug assignment prediction accuracy in percentages using Naïve Bayes and Bayesian Network with/without tossing graph information. Since the accuracy of first 1 is the same with and without tossing graphs, it is omitted.

Program	ML algorithm	Selection	Accuracy (%)		Improvement
			ML only	ML + tossing graph	
Eclipse	Naïve Bayes	first 2	43.70	44.71	1.01
		first 3	49.87	53.15	3.27
		first 4	56.42	59.95	3.53
		first 5	60.71	63.48	2.77
	Bayesian Network	first 2	57.91	58.29	0.38
		first 3	66.71	68.47	1.76
		first 4	69.47	71.48	2.01
		first 5	75.88	77.14	1.26
Mozilla	Naïve Bayes	first 2	33.41	56.39	22.98
		first 3	45.39	63.82	18.43
		first 4	52.94	69.51	16.57
		first 5	59.35	72.92	13.58
	Bayesian Network	first 2	40.02	55.85	15.84
		first 3	50.25	63.05	12.81
		first 4	55.40	67.45	12.05
		first 5	59.53	70.82	11.29

accuracy for each. (The accuracy of first 1 is omitted, since the prediction sets for first 1 are the same thus their accuracy is the same.)

Table 4 summarizes the results. As we expected, the predication accuracy using first 3 is around 50 to 60%. The accuracy increased up to 76% with first 5. The prediction results are slightly different from the original results in Bettenburg et al. [5]. The reason is that we use only the name part of emails to identify developers as described in Section 2.1, while Bettenburg et al. use the entire email to identify developers.

Overall our approach increases the original prediction accuracy. It shines with Naïve Bayes by increasing the prediction accuracy up to 23 percentage points. With Bayesian Network, it increases the accuracy by 0.4 to 16 percentage points. The improvement rates vary, since our prediction partially depends on the prediction sets provided by machine learning approaches. If the prediction sets from previous approaches do not give us room to find good tossing relationships (e.g., they are good enough), it is difficult for our approach to increase the accuracy significantly.

The accuracy of our approach depends on the tossing graph options. From our experiments, we found the best tossing graph options are to use the goal oriented model with between 0 and 10 minimum support and 25% transaction probability.

Our approach, leveraging tossing graphs for bug assignments is orthogonal to previous approaches. Since we utilize tossing relationship based on existing prediction results, our approach potentially improves accuracies of existing approaches.

5. DISCUSSION

5.1 Is tossing bad?

We found some bugs have long tossing length and each tossing step usually takes 30 to 60 days on average.

Questions arose: “Why does tossing happen?" “Is tossing bad?" Some bugs are related to many modules, and it is hard to fix these bugs with a single developer. In this case, managers or developers make plans to fix bugs. According to their plans, they toss bugs to relevant developers. Such bug tossing is inevitable. Tossing graphs can help this situation when they make plans. If fixing a bug involves many developers, managers and developers can find the effective developer order using the developer relationship information in the tossing graphs.

As shown in Figure 1, when a bug is reopened, it is necessary to reassign the bug to a developer. If the previous fixer is available, usually it would be reassigned to the fixer. However, if the previous fixer is not available, we have to find another developer to fix the bug. As a result, a tossing event occurs. In this case, a manager can get help from tossing graphs to find candidates who have strong tossing relationship with the previous fixer.

Bug tossing also happens when developers retire from a project. For example, if someone retires from a project, all his bugs will be reassigned to others. This tossing process is also inevitable. Our tossing graphs capture these events, since there are many tossing events from the retired developer to others. For a company, it is easy to remove retired developers from their database and disable assigned bugs from the retired developers. However, for open source projects, retirement information is often not documented, and it is hard to delete accounts of inactive developers. In this case, the automatically captured tossing information would be useful to identify retired developers.

Some bug tossing processes are due to confusion or mistakes during assignment processes. As discussed in Section 1, a developer claims that the assigned bug is not his. Another developer is not sure who should be assigned a given bug. These mistakes and confusion contribute to unnecessary tossing steps. In this case, tossing graphs would be very useful. By predicting the next match for the best developers, it will put the tossing path on the right track and prevent unnecessary tossing steps.

5.2 Threats to Validity

We identify the following threats to validity.

Systems examined might not be representative. Bug reports of two systems were examined in this paper. Since we intentionally chose systems for which we could extract high quality bug reports, we might have a project selection bias.

Systems are all open source projects. All systems examined in this paper are developed as open source projects. Hence they might not be representative of closed-source development.

Usually commercial software developers have quality assurance (QA) team support. The QA team classifies bug reports, identifies problems, locates the faults, and assigns the bugs to corresponding module owners. They might assign bugs quickly. However, the developer structure discovery and bug triage systems based on bug tossing are useful for both, open source and commercial software development.

Developer retirement information is hidden. It is possible that some of the developers are retired and do not fix bugs any more. However, our tossing graphs may include activities of retired developers. This could affect bug assignment prediction accuracy.

6. RELATED WORK

Triaging bug reports typically involves two activities in open source projects. First developers check whether the bug has been already filed, i.e., is it a duplicate of another bug report? If it is not, then it is assigned to a developer who is then responsible to correct the bug. Many approaches have been proposed to automate these steps. To detect duplicates, several approaches use natural language processing techniques on the bug description [18, 23], sometimes combined with runtime traces [27]. To assign developers to bug reports, again several approaches used natural language processing [3, 5, 10, 13], however only with moderate success, about 50-60% accuracy. In this paper, we contribute to this body of knowledge by showing that bug tossing graphs can improve the accuracy substantially, as much as 70%.

Bug tossing is the same as ticket routing ("transferring [a] problem ticket among various expert groups in search of the right resolver to the ticket." [24]), which is a well-known problem in the machine learning literature. Most approaches use various statistical models to mine workflow from activity logs [1, 24–26].

Shao et al. proposed a ticket routing algorithm that works without accessing the ticket content by using Markov models [24]. In this paper, we transferred that idea to Eclipse and Mozilla tossing events using a modified graph search algorithm. Like their work, our approach also reduced the length of tossing paths. Furthermore, we combined their content-less approach with a content-based approach to locate an initial developer using a traditional bug assignment algorithm. Another important difference to Shao et al. is that we used our graphs reveal the structure of development teams, while they did not.

Several studies investigated which factors predict the lifetime of bug reports, that is from submission to being closed [4, 19, 21]. For example, bug reports that are easy to read or bug reports with several attachments, stack traces, or code samples are fixed sooner. Only few studies have focused on the actual effort that it takes to fix a bug report, mainly because of limited data [29].

Only a few studies investigated the reassignment of bug reports. D'Ambros et al. [14] visualized the life cycle of bugs, including the assignment of developers. Halverson et al. [17] defined (anti)-patterns in bug reports, one of them was the reassignment of developers. In this paper, we quantify the assignment and reassignment of developers empirically. In addition, we contribute a model of bug tossing, which helps to reduce the number of reassignment events in bug reports.

7. CONCLUSIONS

The assignment of bug reports is still primarily a manual process. Often bugs are assigned incorrectly to a developer or need to be discussed among several developers before the developer responsible for the fix is identified. These situations typically lead to bug tossing, i.e., a bug report being reassigned from one developer to another.

In this paper, we analyzed 445,000 bug reports and their detailed activities from the Eclipse and Mozilla projects. We found that it takes a long time to assign and toss bugs. In addition, some bugs have a long tossing length, which means they are passed among many developers before the bug is actually fixed. To improve the bug assignment process and reduce unnecessary tossing steps, we proposed a tossing graph model, which captures past tossing history. In our experiments, the model reduces tossing steps by up to 72% and improved the accuracy of automatic bug assignment by up to 23 percentage points.

The proposed tossing graph model can be easily integrated into existing bug tracking systems. For example, when bugs are assigned to a developer, the integration can recommend additional developers based on previous tossing history. This is helpful, especially when the first choice person to fix a bug is not available (e.g., because of vacation or other commitments). Another scenario is to assign bugs not to single developers but rather to a group of developers. Here an initial developer would be identified first and our integration would add the remaining related developers.

Our future work consists of the following.

- *Developing tossing models based on hidden Markov models.* One limitation of our current models is that they are based on regular Markov chains and thus only use the current state for prediction (see also discussion in Section 3.1). In contrast, hidden Markov models use all previous states [11]. We expect that they will further improve our results.

- *Integration with bug tracking tools and user studies.* We also plan to integrate our tossing models into existing bug tracking systems such as Bugzilla. For example, one can use the team structure to realize dynamic teams (similar to emergent teams [20]), who can be assigned to bugs just like regular developers. Once we have integrated our techniques, we will seek to further demonstrate their usefulness with user studies.

- *Modeling files and developers together.* Currently our tossing graph model only captures developer activities. We are planning to model fixed files and developers together. This model would provide information to predict developers from bug reports and files from bug reports or developers.

To our knowledge, this paper is the first to analyze tossing history and capture it in graphs to support bug triage tasks. We expect that future approaches will further leverage bug tossing history to improve bug reporting and tracking processes. Our approach is a first step in this direction.

8. ACKNOWLEDGMENTS

Our thanks to Prof. Kwangkeun Yi for his guidance and insightful comments on this project. We also thank ROPAS group members for their brainstorming and feedback. We thank Nicolas Bettenburg for providing bug assignment prediction data. We also thank the Eclipse and Mozilla Developers for their feedback on the tossing graphs. Thanks to Rebecca Aiken, Raymond Wong, Yungbum Jung, Hakjoo Oh, Soonho Kong and the anonymous reviewers of ESEC/FSE for valuable and helpful suggestions on earlier versions of this paper.

9. REFERENCES

[1] R. Agrawal, D. Gunopulos, and F. Leymann. Mining process models from workflow logs. In *EDBT '98: Proceedings of the 6th International Conference on Extending Database Technology*, pages 469–483, London, UK, 1998. Springer-Verlag.

[2] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[4] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th International Symposium on Foundations of Software Engineering*, November 2008.

[5] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, September 2008.

[6] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[7] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143, New York, NY, USA, 2006. ACM.

[8] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35, New York, NY, USA, 2008. ACM.

[9] Bugzilla bug tracking system. `http://www.bugzilla.org/`. Last accessed 2009-03-10.

[10] G. Canfora and L. Cerulo. Supporting change request assignment in open source development. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1767–1772, 2006.

[11] O. Cappé, E. Moulines, and T. Ryden. *Inference in Hidden Markov Models (Springer Series in Statistics)*. Springer, August 2005.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[13] D. Cubranic and G. C. Murphy. Automatic bug triage using text categorization. In *SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[14] M. D'Ambros, M. Lanza, and M. Pinzger. "A Bug's Life" Visualizing a Bug Database. In *Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VisSoft 2007)*, pages 113–120, Banff, Alberta, Canada, 2007. IEEE Computer Society.

[15] R. Durrett. *Probability : Theory and Examples*, chapter 5. Markov Chains. Wadsworth, Pacific Grove, California, 1991.

[16] C. M. Grinstead and J. L. Snell. *Introduction to Probability*, chapter 11. Markov Chains. American Mathematical Society, Pacific Grove, California, 1997.

[17] C. A. Halverson, J. B. Ellis, C. Danis, and W. A. Kellogg. Designing task visualizations to support the coordination of work in software development. In *CSCW '06: Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, pages 39–48, 2006.

[18] L. Hiew. Assisted detection of duplicate bug reports. Master's thesis, The University of British Columbia, Vancouver, Canada, May 2006.

[19] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 34–43, 2007.

[20] S. Minto and G. C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[21] L. D. Panjer. Predicting eclipse bug lifetimes. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 29, Washington, DC, USA, 2007. IEEE Computer Society.

[22] 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.

[23] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 499–510. IEEE Computer Society, 2007.

[24] Q. Shao, Y. Chen, S. Tao, X. Yan, and N. Anerousis. Efficient ticket routing by resolution sequence mining. In *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 605–613, New York, NY, USA, 2008. ACM.

[25] R. Silva, J. Zhang, and J. G. Shanahan. Probabilistic workflow mining. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 275–284, New York, NY, USA, 2005. ACM.

[26] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, 2004.

[27] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008.

[28] Y. Wang, L. Li, and D. Xu. Pervasive QoS routing in next generation networks. *Comput. Commun.*, 31(14):3485–3491, 2008.

[29] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society.

[30] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.