Predicting Defect Numbers Based on Defect State Transition Models

Jue Wang and Hongyu Zhang
School of Software, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology (TNList)
Beijing 100084, China
cecilia.juewang@gmail.com, hongyu@tsinghua.edu.cn

ABSTRACT
During software maintenance, a large number of defects could be discovered and reported. A defect can enter many states during its lifecycle, such as NEW, ASSIGNED, and RESOLVED. The ability to predict the number of defects at each state can help project teams better evaluate and plan maintenance activities. In this paper, we present BugStates, a method for predicting defect numbers at each state based on defect state transition models. In our method, we first construct defect state transition models using historical data. We then derive a stability metric from the transition models to measure a project's defect-fixing performance. For projects with stable defect-fixing performance, we show that we can apply Markovian method to predict the number of defects at each state in future based on the state transition model. We evaluate the effectiveness of BugStates using six open source projects and the results are promising. For example, when predicting defect numbers at each state in December 2010 using data from July 2009 to June 2010, the absolute errors for all projects are less than 28. In general, BugStates also outperforms other related methods.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *performance measures, process metrics, product metrics.* D.2.9 [**Software Engineering**]: Management – *software quality assurance (SQA)*

General Terms
Measurement, Reliability, Management

Keywords
defect state transitions, defect prediction, defect numbers, defect-fixing performance, Markov models

1. INTRODUCTION
For a large and evolving software system, the project team could receive defect reports over a long period of time. After a defect report is received and confirmed, a defect begins its lifecycle. A typical life cycle of a defect includes many states such as New, Resolved, and Closed. A defect can transfer from one state to an allowable state once it is handled by developers. Currently, the defect state transition process is often recorded and tracked by bug tracking systems such as Bugzilla and Jira.

We believe it is important to understand the defect state transition process as it records the defect-fixing activities of a project team. The number of defects transferring from one state to the other actually reflects the team's defect-fixing performance. By analyzing defect state transitions we can estimate the stability of a team's defect-fixing performance over time.

For projects that have stable defect-fixing performance, it is desirable to predict the distribution/number of defects at each state in future based on the project teams' past defect-fixing performance. For example, we can predict the number of defects that will remain unresolved and the number of defects that will be resolved in the next six months. Such information can help a project team better estimate software maintenance effort and allocate limited resources.

In this paper, we present BugStates, a method for predicting defect numbers at each state. In BugStates, we represent defect state transition processes as Markov-like models, specifically, the Discrete-Time Markov Chain (DTMC) model. Based on historical state transition data, we can calculate the likelihood of the occurrence of each transition and construct state transition models. We derive a metric from defect state transitions to measure the stability of a project team's defect-fixing performance. For teams that have stable defect-fixing performance, we apply Markovian method to predict the number of defects at each state based on the assumption that past defect-fixing performance can be used to predict future performance.

To evaluate the proposed BugStates approach, we perform experiments on six open source projects, including four major Eclipse projects (PDE.Build, JDT.Text JDT.UI and Platform.Debug), as well as the Lucene and Spring.NET projects. We collect their state transition data by mining the defect activity logs maintained by Bugzilla or Jira, and then construct Markov-like models. For the projects that exhibit stable defect-fixing performance, we predict the number of defects at each state in the next six months based on the models constructed using the past twelve months' data. For example, when predicting defect numbers at each state in December 2010 using data from July 2009 to June 2010, the absolute errors for all projects are between 0 and 28, which are relatively small comparing to the actual number of defects (0 to 547). Furthermore, our evaluation results show that the BugStates outperforms three other methods (arithmetic mean, G-O model and quadratic polynomial methods).

In recent years, many researchers have mined bug databases for defect prediction [13, 24, 26], bug triage [9], effort prediction [12, 19], etc. To our best knowledge, modeling and predicting defect

ESEM'12, September 19–20, 2012, Lund, Sweden.

state transition have not received enough attention. Our work dedicates to this topic and makes the following contributions:

- We show that we can measure the stability of the project team's defect-fixing performance by analyzing the data derived from a defect state transition model.
- We show that we can predict defect numbers at each state based on a state transition model constructed using the historical data.

We believe that BugStates can help project teams improve maintenance process and project management. The rest of the paper is organized as follows. Section 2 briefly introduces the background about defect state transitions and the Markovian method. Section 3 presents the proposed BugStates approach for modeling and predicting defect numbers at each state. Section 4 describes our experiments and results. Section 5 analyzes the threats to validity. Section 6 briefly describes related work and Section 7 concludes this paper.

2. BACKGROUND

2.1 Defect State Transitions

Currently, defects are often recorded and managed by bug tracking systems such as Bugzilla [3] and Jira [10]. Figure 1 shows the defect state transition process supported by Bugzilla [3]. The defect state transition process describes a defect's life cycle. All the major transitions are as follows. Once a defect is reported, if nobody has confirmed that the problem is real, the defect's status is UNCONFIRMED. If the defect is confirmed, a defect record is created and its status is set to NEW. A defect's state can be changed from NEW to ASSIGNED when the defect is assigned to a developer, and then to RESOLVED after the developer has resolved the defect. Once a defect fix is verified by a QA personnel, the defect enters the state VERIFIED and will be subsequently CLOSED. If the defect is not fixed successfully, the defect will be REOPENED and may enter other states again. The defect state transitions described above are probabilistic as a defect can enter any allowable state and independent from the earlier states that the defect entered before.

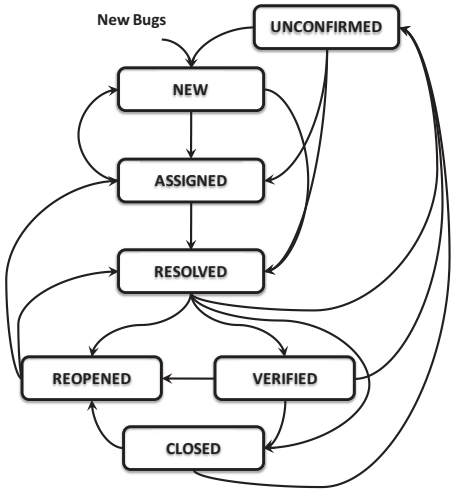

Figure 1. The defect state transition process of Bugzilla

Figure 2 shows the default state transition process supported by Jira [10], which is another widely-used bug tracking system. Unlike Bugzilla, there is no VERIFIED state in Jira. The NEW and ASSIGNED states are replaced by the OPEN and IN PROGRESS states, respectively. The other states remain the same with Bugzilla.

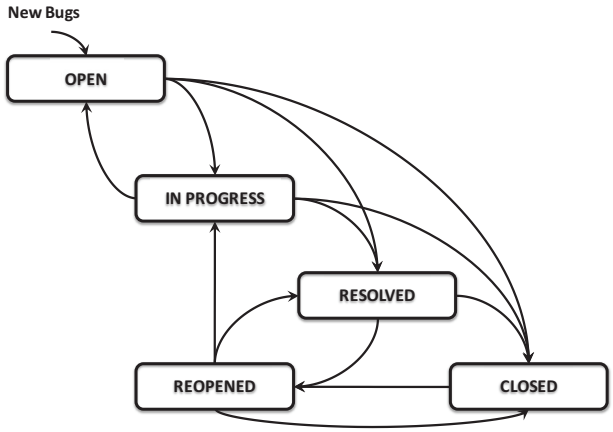

Figure 2. The default defect state transition process of Jira

2.2 Markovian Method

A Markov model is for modeling stochastic process with the property that the next state depends only on the current state [2, 11]. In this work, we consider discrete-time stochastic process and adopt the Discrete Time Markov Chain (DTMC) model. Formally, considering a stochastic process $X_1, X_2, X_3, \ldots, X_n$, where X_t is a random variable denoting the state of the process at discrete time t (t = *0, 1, 2* …), the process is a Markov chain if the following property holds:

$$Pr(X_{n+1} = x | X_1 = x_1, X_2 = x_2 \ldots, X_n = x_n)$$

$$= Pr(X_{n+1} = x | X_n = x_n). \quad (1)$$

For a Markov model, its transition probability matrix P is defined as follows:

$$P = \begin{pmatrix} p_{11}p_{12} & \cdots & p_{1,x-1}p_{1,x} \\ p_{21}p_{22} & \cdots & p_{2,x-1}p_{2,x} \\ & \vdots & \\ p_{x,1}p_{x,2} & \cdots & p_{x,x-1}p_{x,x} \end{pmatrix} \quad (2)$$

where p_{ij} represents the probability from x_i to x_j. Then we have:

$$\alpha_{t+1} = P \cdot \alpha_t \ , \quad (3)$$

where α_t represents the state probability vector at time t, $\alpha_t(i) = \Pr(X_t = i)$, α_{t+1} represents the state probability vector at time t+1.

In order to apply the Markovian method, a process must satisfy the property that the next state depends only on the current state. In software defect state transitions, a defect's state is changed by a developer when a certain defect-fixing action is completed. The defect's next state depends on its current state, not its previous states. Furthermore, a defect's next state cannot be known deterministically in advance. Therefore, the defect state transition process can be represented using Markov-like models (we use the term "Markov-like" as we did not rigorously prove that all defect state transition processes always hold the Markov property).

3. THE PROPOSED METHOD: BUGSTATES

3.1 Overall Structure

During software evolution new features could be added to the system and new defects could be reported to the project team. In this section we introduce the proposed BugStates method for measuring defect state transitions and predicting future defect numbers. Figure 3 shows the overall structure of BugStates. We first mine the historical defect records to obtain the state transition data. We then use the obtained data to construct defect state transition models. We measure the stability of defect-fixing performance using a distance metric derived from the transition model. If the defect-fixing performance is stable, we apply the Markovian method to predict distribution of defects among states at a future time. Based on the historical defect data we can also construct defect growth models and predict the total number of defects at a future time. Having estimated the defect distributions and the total numbers, we can then predict the number of defects at each state in future.

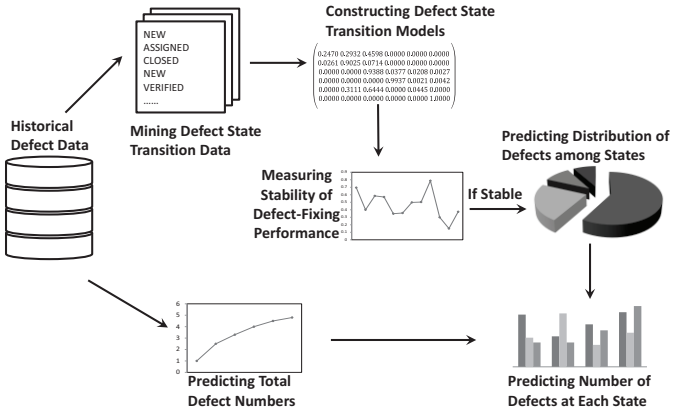

Figure 3. The overall structure of BugStates

3.2 Mining Defect State Transition Data

Typical defect tracking systems, such as BugZilla and Jira maintain an activity log for each defect. An activity log includes information such as a defect's previous states, current state, state change time, and developers assigned to this defect. Defect state transition data can be collected by mining the activity logs.

Table 1 gives an example of the activity logs for JDT.UI defect #280333, which is maintained by Eclipse Bugzilla. The bold lines mark the transitions of the defect's state. From the table we know that the defect is created at 07:40 on June 16, 2009, resolved as "Won't Fix" at 9:27 on the same day, re-opened on June 22, 2009, fixed on August 3, 2009, verified on August 6, 2009 and finally closed on August 10, 2009.

Table 2 gives an example of the activity logs for Lucene-Java defect #2060, which is maintained by Jira. The bold lines mark the transitions of the defect's state. The defect is resolved as "Fixed" on November 14, 2009 and closed on November 25, 2009. However, it was re-opened on May 30, 2010 and eventually closed on June 18, 2010.

To construct a defect state transition model, we collect data by analyzing activity logs (such as the ones shown in Table 1 and Table 2). In the bug tracking systems, these activity logs are typically presented as HTML pages. By crawling and parsing these HTML pages we can extract the state transition information for each defect. We then count the number of defects at each state at each month, and calculate the probability of a defect transiting from one state to the other state, thus obtaining the defect state transition data.

Table 1. The activity log of defect #280333[1] for Eclipse JDT.UI

Who	When	What	Removed	Added
pwebs-ter	2009-06-16 07:40:50	CC		pwebs-ter
markus_keller	2009-06-16 09:27:01	CC		markus_keller
		Status	**NEW**	**RESOLVED**
		Resolution		WONTFIX
paules	2009-06-22 12:09:30	**Status**	**RESOLVED**	**REOPENED**
		Resolution	WONTFIX	
daniel_megert	2009-06-23 02:30:39	**Status**	**REOPENED**	**NEW**
		…	…	…
markus_keller	2009-08-03 07:59:13	**Status**	**NEW**	**RESOLVED**
		Resolution		FIXED
daniel_megert	2009-08-06 03:26:16	**Status**	**RESOLVED**	**VERIFIED**
paules	2009-08-10 08:24:07	**Status**	**VERIFIED**	**CLOSED**

Table 2. The activity log of defect #2060[2] for Lucene-Java

Michael McCandless made changes – 14/Nov/09 11:22		
Field	Original Value	New Value
Status	**Open**	**Resolved** (Resolution Fixed)
Uwe Schindler made changes – 25/Nov/09 16:47		
Status	**Resolved**	**Closed**
Michael McCandless made changes – 30/May/10 12:40		
Status	**Closed**	**Reopened**
…		
Uwe Schindler made changes – 18/Jun/10 08:03		
Status	**Resolved**	**Closed**

3.3 Constructing Defect State Transition Models

In defect state transition graphs such as the ones shown in Figure 1 and Figure 2, a defect's next state is only dependent on its current state. Therefore, the state transition process can be treated as a Markov-like model (more specifically, the DTMC model). Figure 4 describes the mapping from a Bugzilla defect state transition graph to a Markov-like model. In the mapping, each defect state is translated into a state labeled with a sequential number. Each arrow linking two states has a transitional probability from the start state to the destination state. Moreover, for each state in the Markov-like

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=280333
[2]https://issues.apache.org/jira/browse/LUCENE-2060

model we add an arrow pointing to itself as a defect could remain in the same state at the next time interval. Note that in our method we only consider confirmed bugs (for our experimental subjects like Eclipse, all known bugs are confirmed bugs), therefore, for conciseness, we do not consider the UNCONFIRMED state in Figure 4.

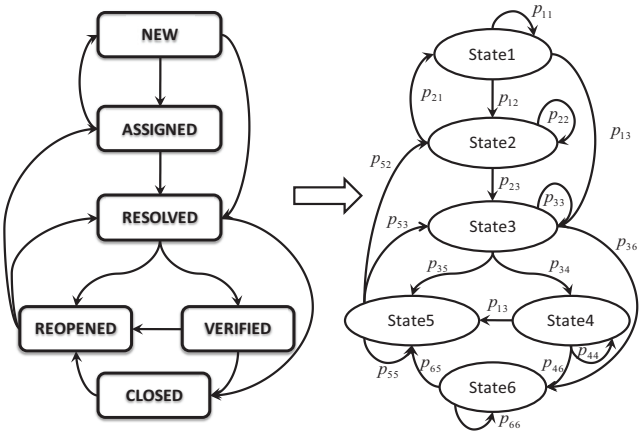

Figure 4. Mapping from Bugzilla defect state transition graph to Markov-like model

For the model shown in Figure 4, we can obtain the state transition probability matrix as follows:

$$P = (p_{ij}), \quad 1 \le i \le 6, 1 \le j \le 6, \tag{4}$$

where p_{ij} represents the probability from defect state *i* to state *j*, e.g. p_{13} represents the probability of a defect transferring from NEW to RESOLVED. Similarly, we can construct Markov-like models for Jira's defect state transition process.

3.4 Measuring the Stability of Defect-Fixing Performance

We derive a metric from the defect state transition matrix to measure the stability of a team's defect-fixing performance. Projects having stable defect-fixing performance should exhibit similar defect state transition matrix over time, while unstable defect-fixing performance would lead to large variances in defect state transitions.

To measure the stability of a project's defect-fixing performance over a time period, we first compute the defect state transition matrix that represents the project's average performance during the period. Then, we calculate the Euclidean distance between the current defect state transition matrix and the average defect state transition matrix. Formally, let P_m denote transition matrix in month *m* in a period *T,* and P_a denotes the average transition matrix during *T*, we calculate distance *D* between P_m and P_a by computing the average Euclidean distance for all the states as follows:

$$D = \frac{\sum_{i=1}^{n} \sqrt{\sum_{j=1}^{n} (p_{mij} - p_{aij})^2}}{n}, \tag{5}$$

where *n* is the number of states. We also use $D_{average}$ to represent the average *D* value during a time period *T*.

$$D_{average} = (\sum_{m=1}^{T} D_m)/T \tag{6}$$

The $D_{average}$ distance value is between 0 and 1, with a lower value indicating a more stable process. If $D_{average}$ exceeds a predefined distance threshold (such as 0.25), we consider the defect-fixing performance instable. The actual threshold value can be specified by each individual team.

3.5 Predicting Defects Numbers at Each State

For a project that has stable defect-fixing performance, we can predict its future defect state transitions based on its current performance. Such ability can help the project team better estimate software maintenance efforts and plan the quality assurance resources.

To achieve the prediction, we utilize the characteristic of Markov model as given in Equation 3. Basically, we can predict the future defect state transitions based on the initial states and the state transition matrix. Our method consists of the following steps:

1) Predicting the distribution of defect among states
2) Predicting the number of defects at each state

3.5.1 Predicting the Distribution of Defects among States

We first gather statistics of the number of the defects at each state at an initial time *t*, we then compute the distribution of defects among the states (i.e., the state probability vector). For example, for Bugzilla, we define the number of defects in NEW, ASSIGNED, RESOLVED, VERIFIED, REOPEN, and CLOSED as $n_1, n_2, n_3, n_4, n_5,$ and $n_6,$ respectively. Then the sum of the defects is $S = \sum_{n=1}^{6} n_i$, and the state probability vector at time *t* is:

$$\alpha_t = (n_1/S, n_2/S, n_3/S, n_4/S, n_5/S, n_6/S). \tag{7}$$

According to the Equation 3, we can infer the distribution of defects across the states at time *t+N* as follows:

$$\alpha_{t+N} = P^N \cdot \alpha_t. \tag{8}$$

Equation 8 allows us to obtain the probability of defects at each state at a future time *t+N*.

3.5.2 Predicting the Number of Defects at Each State

Equation 8 only gives the probability of defects staying at a state. To estimate the number of defects at each state at a future time, we need to predict the total number of defects. Following typical software reliability engineering methods [15, 17], we can depict the growth of cumulative defect numbers based on historical data, apply non-linear regression analysis to fit the defect growth curve, and then use the fitted regression model to predict the number of newly reported defects at time *t+1*. More specifically, we treat the defect data as a time series:

$$f(1), f(2), \ldots, f(t), \ldots, \tag{9}$$

where $f(t)$ represents the number of defects at month *t*. The cumulative number of defects can be also represented as a series, which describes the growth of defects over time:

$$f(1), f(1) + f(2), \ldots, \sum_{i=1}^{t} f(i) \tag{10}$$

In this research, we adopt the Goel-Okumoto (G-O) model [8, 17] to describe the growth of defect numbers over time. The G-O model is one of the commonly-used software reliability growth models, which is defined as follows:

$$m_t = a(1 - e^{-bt}), \tag{11}$$

where t represents the time (in month), m is the number of cumulative defects at time t, a and b are coefficients. We should note that our method does not depend on the G-O model. Other reliability growth models can be also integrated into our approach.

Having estimated the defect state probability vector α_{t+N} and the total number of defects m_{t+N} at time $t+N$, we can then predict the number of defects at each state as follows:

$$M_{t+N} = m_{t+N} \cdot \alpha_{t+N}, \quad (12)$$

M_{t+N} is the vector representing the number of defects at each state at time $t+N$.

4. EXPERIMENTS

4.1 Subjects

To evaluate the proposed BugStates method, we perform experiments on six open source projects (Table 3), including four major Eclipse projects (PDE.Build, JDT.Text JDT.UI and Platform.Debug), as well as the Lucene-Java.Index and Spring.NET projects. The four Eclipse projects use Bugzilla as their bug tracking system. The other two projects use Jira. All these projects have evolved over a long period. In this experiment, we only consider defects reported from January 1, 2009 to December 31, 2010.

Table 3. The projects studied

Project	Description	Defects Reported in 2009	Defects Reported in 2010
PDE.Build[3]	build support for Java	227	139
JDT.Text[4]	Java editing support	248	235
JDT.UI[5]	user interface for the Eclipse Java IDE	663	472
Platform. Debug[6]	debug support for Eclipse platform	348	231
Lucene-Java.Index[7]	the index component of the Apache Java search engine	47	65
Spring.NET[8]	a port and extension of the Java based Spring Framework for .NET	66	39

4.2 Constructing Defect State Transition Models

For each defect in each project, we collect its state transition data by mining its activity log stored in the Eclipse Bugzilla[9] and Jira bug databases[10].To automatically extract the defect state transition information from an activity log, we developed a tool that can automatically grab the web page of the activity log, parse it and extract transition information between every two defect states. If a defect's state is not changed within a month, it has a transition to itself (staying at the same state). The tool counts the number of state transitions for each defect in each month in 2009 and then computes the total transitions in a year.

Having collected the state transition data, we then calculate the transition probability matrix P and construct a state transition model for the project. As an example, the transition matrix for the JDT.UI project ($P_{JDT.UI}$) for year 2009 is given below. The probabilities of some major transitions are as follows:

- From New to New: 24.70%,
- From New to Assigned: 29.32%
- From Assigned to Resolved: 7.14%
- From Resolved to Reopen: 2.08%.
- From Resolved to Verified: 3.77%

$$P_{JDT.UI} = \begin{array}{c} \begin{matrix} NEW & ASSI. & RESO. & VERI. & REOP. & CLOSED \end{matrix} \\ \begin{pmatrix} 0.2470 & 0.2932 & 0.4598 & 0.0000 & 0.0000 & 0.0000 \\ 0.0261 & 0.9025 & 0.0714 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.9388 & 0.0377 & 0.0208 & 0.0027 \\ 0.0000 & 0.0000 & 0.0000 & 0.9937 & 0.0021 & 0.0042 \\ 0.0000 & 0.3111 & 0.6444 & 0.0000 & 0.0445 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{pmatrix} \end{array}$$

4.3 Measuring Stability of Defect-Fixing Performance

For the projects studied, we apply the method described in Section 3.4 to measure the stability of a project team's defect-fixing performance.

We measure the Euclidean distances between each month in 2009 (P_m) and the average value of 2009 (P_a). Figure 5 shows the distance values D for all the projects from January 2009 to December 2009. For the Lucene-Java.Index project, its distance values vary between 0.150 and 0.787, while for other projects, the distance values vary between 0.006 and 0.393. Clearly, the performance of Lucene-Java.Index project is less stable than the other projects.

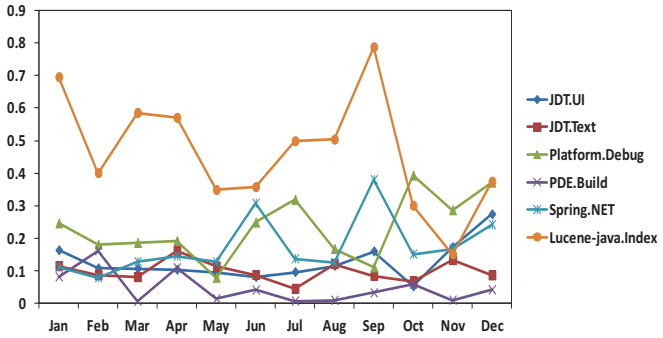

Figure 5. The Euclidean distances *D* for all projects in 2009

For JDT.UI, JDT.Text, PDE.Build, Platform.Debug, Spring.NET and Lucene-Java.Index, the computed average distance value $D_{average}$ in 2009 (12 months) is 0.127, 0.098, 0.048, 0.231, 0.175 and 0.464, respectively. We set the distance threshold as 0.25 in our experiments.

For all projects except Lucene-Java.Index, the $D_{average}$ value is below the distance threshold. Therefore, we consider the defect-fixing performance of these projects stable. The $D_{average}$ for the

[3]http://www.eclipse.org/pde/pde-build/
[4]http://www.eclipse.org/eclipse/platform-text/index.php
[5]http://www.eclipse.org/jdt/ui/index.php
[6]http://www.eclipse.org/eclipse/debug/index.php
[7]http://lucene.apache.org/java/docs/index.html
[8]http://www.springframework.net/
[9]https://bugs.eclipse.org/bugs/
[10]http://issues.apache.org/jira, http://jira.springsource.org

Lucene-Java.Index project is 0.464, which is above the distance threshold and its defect-fixing performance is considered unstable.

4.4 Predicting Defect State Transitions

In this section, we evaluate the effectiveness of BugStates for predicting the number of defects at each state, on the projects that exhibit stable defect-fixing performance. We design the following experiments:

- *Experiment A:* Construct a state transition model using data from January 2009 to December 2009, and then use the model to predict the number of defects at each state from January 2010 to June 2010.
- *Experiment B:* Construct a state transition model using data from July 2009 to June 2010, and then use the model to predict the number of defects at each state from July 2010 to December 2010.

Basically, the above experiments use past twelve months' data to construct a Markov-like model, and then use this model to predict the number of defect at each state in the next six months.

To evaluate the accuracy of the predictions, we use the metrics AE (absolute error) and MAE (mean absolute error). AE is defined as the absolute prediction error $|y–y'|$, where y and y' are the actual value and its estimate, respectively. MAE is the average absolute prediction errors over the dataset:

$$AE = |y–y'|, \quad MAE = \frac{\sum_{i=1}^{n} AE}{n} \qquad (13)$$

where n is the number of data points (in our experiments, n is the number of months being estimated). The smaller the MAE value the better the estimation.

4.4.1 Predicting the Distribution of Defects among States

Following the method described in Section 3.5, for projects having stable defect-fixing performance (all projects except Lucene-Java.Index), we predict the future distribution of defects among the states (a_{t+N}) based on the transition matrix (P) and the initial defect distribution (a_t).

In Experiment A, we first calculate the percentage of defects at each state at the end of December 31, 2009. These percentage values form the initial defect distribution vector. As an example, for the JDT.UI project, 7.99% of the defects at the end of 2009 are in the NEW state, 21.57% of the defects are in state ASSIGNED, and 51.89% of the defects are RESOLVED. The initial defect distribution vectors for all projects are given below.

	NEW	ASSI.	RESO.	VERI.	REOP.	CLOS.
PDE.Build	(0.3744	0.0044	0.5991	0.0088	0.0000	0.0132)
JDT.UI	(0.0799	0.2157	0.5189	0.1237	0.0015	0.0603)
JDT.Text	(0.0363	0.2472	0.4960	0.1250	0.0040	0.0645)
Platform. Debug	(0.3420	0.0344	0.2414	0.3420	0.0115	0.0287)
Spring.NET	(0.4394	0.0303	0.5152	N/A	0.0000	0.0152)

Having obtained the transition matrix P and the initial defect distribution vector a_t, we can then predict the future defect distributions by following the Equation 8. As an example, the resulting distributions of defects at the end of June 2010 are as follows:

	NEW	ASSI.	RESO.	VERI.	REOP.	CLOS.
PDE.Build	(0.2792	0.0071	0.6643	0.0106	0.0000	0.0389)
JDT.UI	(0.0152	0.1967	0.5034	0.1751	0.0108	0.0988)
JDT.Text	(0.0093	0.2437	0.4506	0.1742	0.0132	0.1090)
Platform. Debug	(0.2340	0.0289	0.2525	0.4040	0.0176	0.0630)
Spring.NET	(0.3238	0.0697	0.5220	N/A	0.0121	0.0723)

The above results show the predicted distributions of defects at each state on June 30, 2010. For example, for the JDT.UI project, 1.52% defects are at the NEW state, 19.67% defects are assigned, 50.34% are resolved and 9.88% are closed. Actually, on June 30, 2010, there are 3.10% defects at the NEW state, 20.50% defects are assigned, 50.11% defects are resolved and 10.04% defects are closed. The absolute prediction errors (AE) are between 0.2% and 1.6%, showing that the estimations are accurate.

Similarly, we perform the Experiment B to predict the distribution of defects from July 2010 to December 2010 based on historical data from the past 12 months. Figure 6 shows the comparisons between the predicted and actual values as of December 31, 2010. Clearly, the estimated values are all close to the actual data, the absolute prediction errors (AE) are within 17%.

Table 4 shows the MAE values of the predicted distributions for all twelve months of 2010. The average absolute errors between prediction and actual values range from 0.1% to 6.8%, showing that the predictions are accurate and consistent across all months.

Table 4. The MAE values for predicting defect distributions (all twelve months of 2010)

	NEW/ OPEN	ASSI./ INPRO.	RESO-LVED	VERI-FIED	REOP-ENED	CLOS-ED
PDE. Build	0.053	0.001	0.053	0.002	0.002	0.004
JDT. UI	0.023	0.009	0.009	0.016	0.010	0.003
JDT. Text	0.023	0.006	0.013	0.015	0.013	0.010
Plat. Debug	0.068	0.014	0.012	0.041	0.005	0.005
Spring. NET	0.052	0.025	0.034	N/A	0.002	0.008

4.4.2 Predicting the Number of Defects at Each State

Having predicted the distribution of defects among states, we then evaluate the effectiveness of the BugStates in predicting the number of defects at each state.

Following the method described in Section 3.5, we first predict the total number of defects by modeling the growth of defect numbers. For the projects studied, we calculate the cumulative monthly defect numbers from January 2009 to December 2009 (for Experiment A) and from July 2009 to June 2010 (for Experiment B), and then use Matlab to construct G-O models (Equation 11). Our results show that the G-O model can well represent the growth of defects over time, with R^2 values ranging from 0.968 to 0.997. Table 5 shows the coefficients of G-O models constructed in Experiment A, as well as the predicted and actual number of defects in June 2010.

Having predicted the defect distributions among states and the total number of defects, we can then predict the number of defects at each state using Equation 12. Figure 7 shows the predicted and

actual numbers of defects at all states in December 2010. The prediction results are promising for all projects. For example, for the JDT.Text project, as of December 31, 2010, 188 defects are at the RESOLVED state and the predicted number is 185, the absolute prediction error (AE) is only 3. There are 123 defects at the ASSIGNED state and the predicted number is 113, the absolute prediction error is only 10. For all projects (except Lucene-Java.Index), the absolute prediction errors for all states are small, ranging from 0 to 28 (the actual numbers of defects range from 0 to 547).

Table 5. The G-O models constructed in Experiment A

	a	*b*	R^2	#Actu.	#Pred.	MRE
PDE. Build	354.2	0.0890	0.990	311	283	9.00%
JDT.UI	1572	0.0450	0.997	905	875	3.31%
JDT.Text	383	0.0860	0.997	350	302	13.71%
Platform. Debug	978	0.0350	0.988	509	460	9.63%
Spring. NET	9455	0.0004	0.968	88	83	5.68%

Table 6. The MAE values for predicting defect numbers (all tweleve months of 2010)

	NEW/ OPEN	ASSI./ INPRO.	RESO-LVED	VERI-FIED	REOP-ENED	CLOS-ED
PDE.Build	17.33	0.083	14.17	0.75	0.67	1.58
JDT.UI	20.58	9.50	6.75	13.17	9.33	3.50
JDT.Text	8.83	7.58	7.58	3.92	4.25	5.42
Platform. Debug	30.75	7.08	14.42	25.08	2.58	4.58
Spring. NET	4.25	2.00	3.42	N/A	0.17	0.58

Table 6 also shows the MAE values for all twelve months of 2010. The average absolute prediction errors range from 0.083 to 30.75, which are relatively small comparing to the total number of defects. The MAE results show that the predictions are accurate and consistent across all months.

In summary, our experiments confirm that BugStates is effective in predicting the number of defects at each state in future, and that past defect-fixing performance can be used to predict future performance.

4.5 Comparisons

To further evaluate the effectiveness of BugStates, we compare it with the following three prediction methods:

- **Arithmetic mean**: this is a simple prediction method that uses the mean average number of changes in the defect state transitions ($\overline{\Delta x}$) in the past to predict the number defects at each state in the next *n* months:

$$y = y_0 + \overline{\Delta x} \cdot n\,, \tag{14}$$

  where y_0 is the initial defect number.
- **G-O model**: it is a classic software reliability model [8, 15]. Unlike Equation 11, here we use it to estimate the cumulative number of defects for each state:

$$y = a(1 - \mathrm{e}^{-bn})\,, \tag{15}$$

  where a and b are parameters and n is the number of months.
- **Quadratic function**: this function assumes that the growth of defects follows a quadratic function. It is defined as follows:

$$y = an^2 + bn + c\,, \tag{16}$$

  where a, b and c are parameters, and n is the number of months. It has been found that for many Eclipse projects, the growth of the cumulative number of defects follows the quadratic function [25].

For the above three methods, we use the data from July 2009 to June 2010 to train the models, and predict the number of defects at each state in December 2010. For each method, there are in total 29 predictions for all states.

Table 7 shows the comparison results. A '+' sign indicates that the BugStates performed better than the related method in terms of the absolute prediction error (AE) in prediction; a '0' sign means that they achieved the same performance; and a '-' signs means BugStates lost. The numbers besides the '+' and '-'signs show the differences between the prediction results of BugStates and the related methods. For example, '-25' on the first row of the table means that BugStates predicts worse than the Arithmetic Mean method by 25 defects. '+29' means that BugStates predicts better than the G-O model by 29 defects.

In summary, among total 29 predictions, BugStates won the Arithmetic Mean method 17 times and only lost 9 times. Similarly, BugStates also won the G-O model (18/29) and the Quadratic function (18/29). The results confirm that, in general, BugStates can achieve better prediction results than the related methods.

Table 7. The comparisions between BugStates and other methods

Project	State	Arithmetic Mean	G-O Model	Quadratic Function
JDT.UI	NEW	-25	+29	+101
	ASSIGNED	+41	-4	+32
	RESOLVED	+172	+80	+82
	VERIFIED	-19	+7	+1
	REOPENED	-10	-11	-11
	CLOSED	+37	+13	+8
JDT.Text	NEW	-12	-15	+24
	ASSIGNED	+3	0	-8
	RESOLVED	+55	+10	0
	VERIFIED	+15	+18	0
	REOPENED	-3	-4	-3
	CLOSED	+26	+8	+20
Platform. Debug	NEW	+22	+52	+42
	ASSIGNED	0	+2	-2
	RESOLVED	-4	-6	+3
	VERIFIED	-12	-35	-17
	REOPENED	0	-4	-3
	CLOSED	+9	-3	+3
PDE.Build	NEW	+41	+21	+39
	ASSIGNED	0	+1	+1
	RESOLVED	+24	+5	-15
	VERIFIED	-1	+1	+1
	REOPENED	-1	-1	-1
	CLOSED	+17	+1	+8
Spring.NET	OPEN	+43	+38	+50
	IN PROGRESS	+1	-1	-1
	RESOLVED	+15	+32	+35
	REOPENED	+2	+2	+2
	CLOSED	+5	+6	+6
Summary	*Won (+)*	*17*	*18*	*18*
	Tie (0)	*3*	*1*	*2*
	Lost (-)	*9*	*10*	*9*

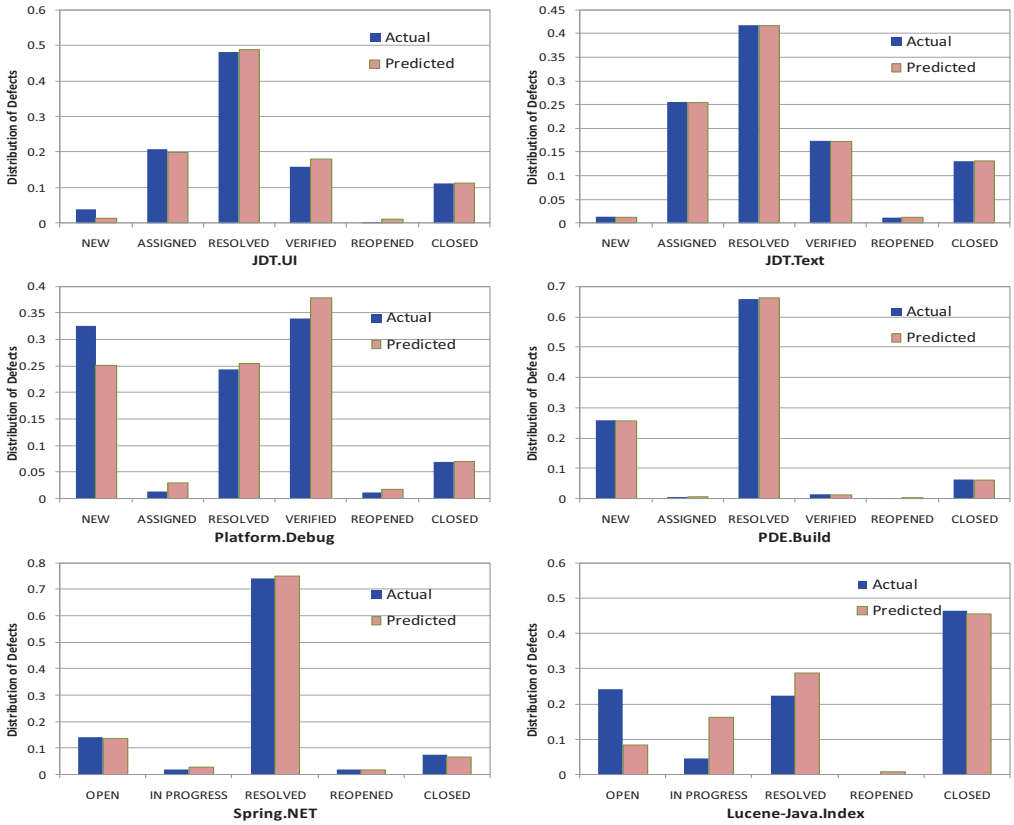

Figure 6. The actual and predicted distribution of defects in December 2010

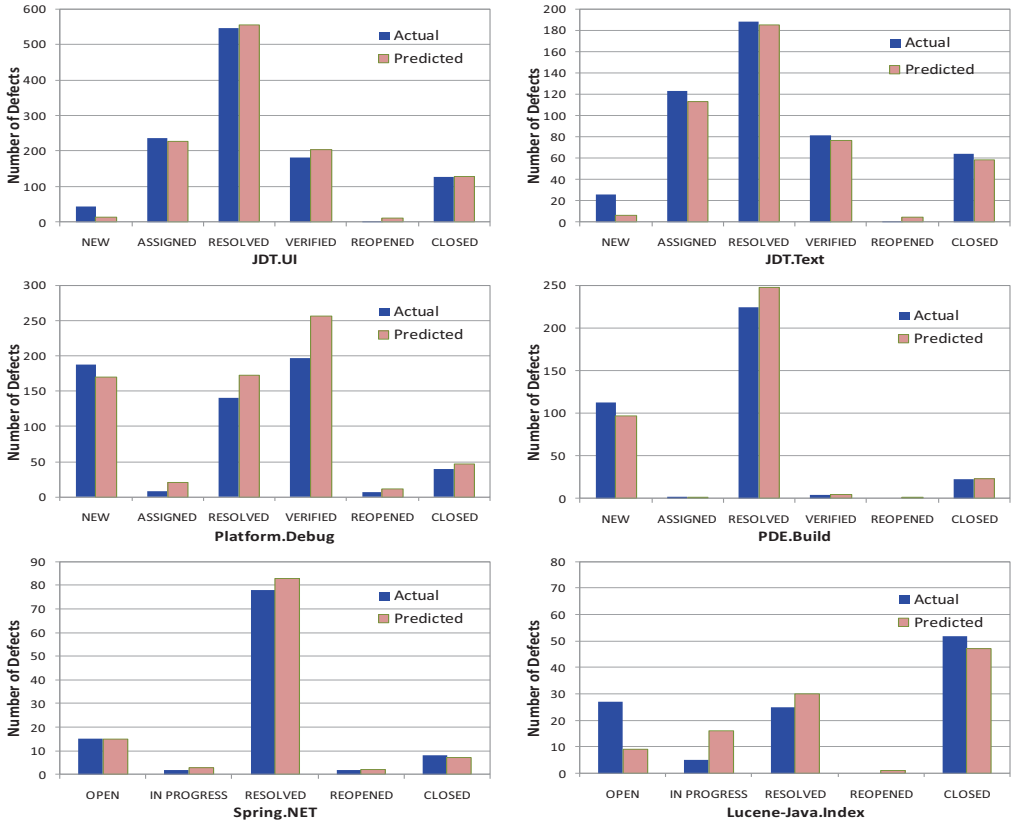

Figure 7. The actual and predicted number of defects in December 2010

4.6 Discussions

The proposed BugStates method can be applied to projects that are experiencing long-term evolution and have stable defect-fixing performance. For such projects, new features are constantly added and new defects are discovered, it is thus desirable to measure and predict the project teams' defect-fixing performance. Our experimental results show that it is reasonable to predict a team's future defect-fixing performance based on its past performance. The ability to have long term prediction of defect numbers can help a project team better plan limited maintenance resources and adjust their development schedule.

For projects that do not have stable defect-fixing performance, BugStates is less effective as the state transition probabilities are varying. Figures 6 and 7 show that for the Lucene-Java.Index project (which is the project exhibiting less stable performance), BugStates leads to larger relative prediction errors, especially for the OPEN and IN PROGRESS states. In practice, we suggest measuring the stability of the defect-fixing performance before applying the prediction method, as projects that have stable defect-fixing performance are expected to lead to consistent prediction results.

Table 8. The impact of distance thresholds on prediction results

	0.1	0.2	0.3	0.4	0.5
JDT.Text	√	√	√	√	√
PDE.Build	√	√	√	√	√
JDT.UI		√	√	√	√
Spring.NET		√	√	√	√
Platform.Debug			√	√	√
Lucene-Java.Index					√
min	0	0	0	0	0
max	24	34	59	59	59
Average MAE	6.014	6.278	7.733	7.733	7.349
Average MRE	0.106	0.086	0.133	0.133	0.170

To measure the stability of a project's defect-fixing performance, in our experiments, we set the distance threshold to 0.25. Table 8 shows the impact of different threshold values on the prediction results (for twelve months in 2010). If we set the threshold to 0.10, only the JDT.Text and PDE.Build projects meet the threshold. BugStates works well for these two projects. The maximum absolute prediction error is 24 and the average MAE value for all states is 6.014. If we set the threshold to 0.20, then the JDT.UI and Spring.NET projects also meet the threshold, with the maximum absolute error 34 and average MAE 6.278. When the threshold is increased to 0.5, all six projects are included. Although the average MAE value is slightly lower than the value achieved by threshold 0.4, the average MRE value is higher. In general, projects having stable defect-fixing performance (i.e., smaller $D_{average}$ values) can achieve better prediction results than projects exhibiting less stable performance (i.e., larger $D_{average}$ values).

5. THREATS TO VALIDITY

We have identified some threats to validities that should be taken into consideration when applying the proposed method:

- **Completeness**: In Eclipse Bugzilla database, there are some transitions that do not follow the defect life cycle (Figure 1). For example, some defects' states are transferred from NEW directly to CLOSED, but actually such transitions are not allowed. We suspect that some developers omitted certain intermediate results when filling the activity logs, thus leading to the "undefined" transitions. If a project has less rigorous QA procedures and has many "undefined" state transitions, the accuracy of our prediction method could be affected.
- **Large evolving system**: The method we proposed is suitable for large software systems that are experiencing a long period of evolution and their project team's defect-fixing performances are stable. For a small or short-living system, the number of defects and state transitions are often small, thus making the statistical analysis inappropriate.
- **Open source data**: All datasets used in our experiments are collected from the open source projects such as Eclipse. The defect state transition behavior of Eclipse projects may be different from those of commercial projects. We will evaluate if BugStates can be applied to commercial projects. This is our important future work.

6. RELATED WORK

In recent years, there have been extensive studies on software defect prediction and analysis. Many methods collect historical defect data by mining software repositories (such as bug database and version archives), identify program features (such as complexity and process metrics), and then build classification models to predict the defect-proneness (defective or non-defective) of a new module. For example, Kim et al. [13] proposed the Change Classification (CC) technique, which learns buggy change patterns from history and predicts if a new change introduces bugs. Zimmermann et al. [24] proposed method for extracting defect information from the CVS/SVN repositories, and to predict defect-proneness of a file. In [16, 26, 27], the authors found that simple complexity metrics such as LOC can be used to predict defect-proneness of components. There are also studies on the distribution of defects over modules of a large software project [1, 5, 28], and on the automated bug localization [29]. In this work, we predict the distribution/number of defects at each state. To our best knowledge, modeling and predicting defect state transition have not been received enough attention.

Some researchers also studied defect life cycles and triage. For example, Koponen [14] studied the defects reported for the Apache HTTP Server and Mozilla Firefox projects from 2003 to 2005. They noticed that the defect life cycles in the two projects were much more straight forward because the states of the defects moved to RESOLVED directly from the initial state. In our study of Eclipse, we also found a few instances whose states were changed directly from NEW to CLOSED, but most of the defects follow the standard defect state transition procedures. Weib et al. [23] studied the lifecycle of defects and presented a search-based approach that can predict the defect-fixing effort. Jeong et al. [9] found that 37%-44% of defect reports for Mozilla and Eclipse are re-assigned to other developers. They constructed a Markov model based on bug tossing history and used it for defect triage. In this work, we mine the state transition information from the defect activity logs, and then use this information to measure and predict defect state transitions.

Our work is related to the field of software reliability engineering. Traditionally, in reliability prediction, the number of operational failures is predicted through a reliability growth model such as the Littlewood, S-Curve and Goel-Okumoto models [15, 17]. In our earlier work, we also proposed polynomial functions based method for predicting defect growth [25]. In recent years, people have also experimented with Markov-based software reliability prediction. For example, in [22] the authors applied Markov models to predict the reliability of a service-based software system. They proposed a

hierarchical reliability model. A Markov model was created for analyzing the reliability of the composite services based on the system control structure and the operational scenarios. In [6], the authors also applied DTMC to model service compositions. They proposed a method called ATOP, which converted an activity diagram to a DTMC model and utilized probabilistic model checking techniques for quality prediction. In [4, 21], the authors proposed a user-oriented reliability model and an architecture-based reliability model using Markovian methods.

There are also studies on the measurement and prediction of defect-fixing performance. For example, Mockus et al. proposed quality metrics (such as percentage of defective files) to understand software maintenance effort quantitatively [18, 7]. Mockus also proposed regression models to estimate maintenance effort and its distribution over time [19]. Kim et al. [12] studied the life span of bugs in ArgoUML and PostgreSQL projects, and found that bug-fixing times have a median of about 200 days. Panjer [20] proposed to use machine-learning models to predict the time to fix a defect. In our work, we apply Markov model to represent the defect state transition process, and use the model to measure defect-fixing performance.

7. CONCLUSIONS

During software evolution, a large number of defects could be reported. Each defect has a life cycle, which is typically maintained by a bug tracking system such as Bugzilla and Jira. In this paper, we have proposed BugStates, a method for predicting defect numbers at each state. In BugStates, we model defect state transitions as Markov-like models. We measure the stability of a project team's defect-fixing performance by using a distance metric derived from the models. Furthermore, we can predict future defect numbers at each state based on the state transition models, that is, to predict a project's future defect-fixing performance based on its past performance. We have evaluated the proposed approach using data mined from defect records of six open source projects. The experimental results confirm the effectiveness of our method. We believe that our method can help project teams better understand and manage software maintenance activities.

In future, we plan to carry out large-scale evaluations of the proposed method on a variety of projects, including commercial projects. We will also analyze defect-fixing performance of different teams and investigate generic performance models.

ACKNOWLEDGMENTS

This research is supported by the NSF China grant 61073006, the 8th Open Lab Project of State Key Lab of Software Engineering (Wuhan University), and the National HGJ Project 2010ZX01045-002-3. We thank Dr Jun Sun at SUTD and Songzheng Song at NUS for their comments and help on an early draft of this paper.

REFERENCES

[1] C. Andersson and P. Runeson. 2007. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Software Eng.* 33 (5), 273-286.

[2] W. G. Belch, S. Greiner, H. de Meer, and K. S. Trivedi. 1998. *Queuing Network and Markov Chains*. John Wiley, Chichester.

[3] Bugzilla: http://www.Bugzilla.org/

[4] R. C. Cheung. 1980. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*. 6(2), 118-125.

[5] N. Fenton and N. Ohlsson. 2000. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Eng.* 26 (8), 797-814.

[6] S. Gallotti, C. Ghezzi, R.Mirandola, and G. Tamburrelli. 2008. Quality prediction of service compositions through probabilistic model checking. In *Proceedings 4th International Conference on the Quality of Software-Architectures* (QoSA'08). 119-134.

[7] D. German, A. Mockus. 2003. Automating the measurement of open source projects. In *ICSE '03 Workshop on Open Source Software Engineering.* Portland, Oregon.

[8] A. Goel and K. Okumoto. 1979. A time dependent error detection model for software reliability and other performance measures. *IEEE Trans. Reliability*, vol R-28, 206-211.

[9] G. Jeong, S. Kim and T. Zimmermann. 2009. Improving defect triage with defect tossing graphs. In *Proceedings ESEC/FSE 2009*, 111-120.

[10] Jira: http://www.atlassian.com/software/jira/

[11] E. Kao. 1996. *An Introducation to Stochastic Processes*. Duxbury Press.

[12] S. Kim and J. E. Whitehead. 2006. How long did it take to fix bugs? In *MSR '06*, 173–174.

[13] S. Kim, J. E. Whitehead, and Y. Zhang. 2008. Classifying Software Changes: Clean or Buggy?. IEEE Trans. Softw. Eng. 34, 2 (March 2008), 181-196.

[14] T. Koponen. 2006. Life cycle of defects in open source software projects. In *IFIP International Federation for Information Processing*. 195-200

[15] M. R. Lyu. 1996. *Handbook of Software Reliability Engineering*. McGraw Hill.

[16] T. Menzies, J. Greenwald, and A. Frank. 2007. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw.Eng.*, 33(1): 2–13.

[17] J. D. Musa. 1987. *Software Reliability: Measurement, Prediction and Application*. McGraw-Hill, New York.

[18] A. Mockus, R. T. Fielding, and J. D. Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346.

[19] A. Mockus, D. M. Weiss, and P. Zhang. 2003. Understanding and predicting effort in software projects. In *ICSE 2003*. 274–284, Portland, Oregon.

[20] L. Panjer. 2007. Predicting eclipse bug lifetimes. In *MSR '07*.

[21] W. L. Wang, Y. Wu and M. H Chen. 1999. An architecture-based software reliability model. In *Proceedings Pacific Rim International Symposium on Dependable Computing*.

[22] L. J. Wang, X. Y. Bai, L. Z. Zhou and Y. N. Chen. 2009. A hierarchical reliability model of service-based software system. In *Proceedings COMPSAC'09*, 199-208.

[23] C. Weib, R. Premraj, T. Zimmermann, and A. Zeller. 2007. How long will it take to fix this bug? In *Proceedings MSR 2007*. Minneapolis, USA.

[24] T. Zimmermann, R. Premraj and A. Zeller. 2007. Predicting defects for eclipse. In *Proceedings PROMISE* 2007. Minneapolis, USA.

[25] H. Zhang. 2008. An initial study of the growth of eclipse defects. In *Proceedings MSR 2008*. Leipzig, Germany, 141-144.

[26] H. Zhang. 2009. An Investigation of the Relationships between Lines of Code and Defects. In *Proceedings ICSM'09*. Edmonton, Canada, 274-283.

[27] H. Zhang and R. Wu. 2010. Sampling Program Quality. In *Proceedings ICSM 2010*. Timisoara, Romania.

[28] H. Zhang. 2008. On the Distribution of Software Faults. *IEEE Trans. on Software Eng*. vol. 34(2), 2008.

[29] J. Zhou, H. Zhang, and D. Lo. 2012. Where Should the Bugs be Fixed? in *Proceedings ICSE'12*, Zurich, Switzerland, 14-24.