# A Probabilistic Approach to Building Defect Prediction Model for Platform-based Product Lines

Changkyun Jeon, Neunghoe Kim, and Hoh In
Department of Computer Science and Engineering, Korea University, Korea

**Abstract**: *Determining when software testing should be begun and the resources that may be required to find and fix defects is complicated. Being able to predict the number of defects for an upcoming software product given the current development team enables the project managers to make better decisions. A majority of reported defects are managed and tracked using a repository system, which tracks a defect throughout its lifetime. The Defect Life Cycle (DLC) begins when a defect is found and ends when the resolution is verified and the defect is closed. Defects transition through different states according to the evolution of the project, which involves testing, debugging, and verification. All of these defect transitions should be logged using the Defect Tracking Systems (DTS). We construct a Markov chain theory-based defect prediction model for consecutive software products using defect transition history. During model construction, the state of each defect is modelled using the DLC states. The proposed model can predict the defect trends such as total number of defects and defect distribution states in the consecutive products. The model is evaluated using an actual industrial mobile product software project and found to be well suited for the selected domain.*

**Keywords:** *Defect prediction, defect life cycle, markov chain, product line engineering, software engineering.*

*Received June 12, 2014; accepted September 21, 2015*

## 1. Introduction

The mobile industry increasingly requires high performance, stability, and multi-functional features. Strong demand and the need to launch various specifications and features on time have been placing substantial pressure on determining how to guarantee expected quality level with given the limited budgets and resources.

To overcome this pressure with regard to requirements and features in consecutive products, many mobile products have been developed using Software Product Line (SPL) concepts. An SPL is a proactive and systematic approach to the development of software that allows for the creation of a variety of products [8]. Most SPLs are designed using a platform that serves as the basis for a family of products, and they rely on an a priori architecture and artifacts from other platform products. Members of a product line can have substantial commonality in, for example, requirements and characteristics, while also exhibiting variability in requirements, design decisions, and implementation details.

During consecutive product development, many features are added, removed, and changed for various reasons. Inevitably, many new defects are generated in consecutive products as the software evolves. The number of defects in such a software project has a significant impact on project performance and hence is an input to project planning [2, 22]. As the quality level of the final product is set at the beginning of the project, a large number of defects can result in project delays and cost overruns [13]. Planning precision and predictability is crucial for the any project in operation [14].

All defects must be recorded, tracked, and managed until the end of a project. It may be necessary to keep histories for subsequent projects or for the evolution of the software, for the efficiency of the testing process, and to aid the common understanding of multi-regional, distributed project members. The Bugzilla [5], Git [10], and Jira [16] Defect Tracking Systems (DTSs) are widely used for these purposes. Thus, it is possible to analyze the historical trends in DTSs, to use the resulting information to predict the number of defects in upcoming software products, and to obtain a concrete view of the life cycle of potentially unreported defects.

In this paper, we propose a Defect Prediction Model (DPM) for predicting the expected quality level in an SPL. We believe that predicting the defect trend in an SPL will provide project managers with better understanding for making proactive decisions about the arrangement of the development team and the implementation of the early test phases considering crucial schedules and limited resources. In addition, it could provide metrics for evaluating the performance of development teams and individual developers, such as the number of unresolved defects that will remain and the expected quality level at the end of a project. Our model for predicting the life cycle of defects with occurrence rates and severities within specific domains could be used as a reference model for evaluating development effectiveness for a platform-based SPL.

This paper makes the following contributions:

- We present a DPM using defect state transition histories after the analysis of repository data from a DTS.
- We can predict the defect state distribution at a given time and the defect closing rate for estimating the total number of defects to help project managers anticipate the quality level in a consecutive SPL.

The rest of the paper is organized as follows. Section 2 summarizes related work. Section 3 briefly introduces background information on DTSs, the Markov Chain (MC) and Software Reliability Growth (SRG) model. Section 4 describes our proposed model in detail to show how to calculate a transition matrix, obtain the initial probability of predicting a product, and estimate the total number of defects and defect distribution states. For evaluating our proposed model, a case study and results are described in sections 5 and 6 discusses limitations of our study. We conclude in section 7 with the future direction of our research.

## 2. Related Works

Software defect prediction has been a very active area in software engineering research [12, 18, 26]. Many effective new metrics and algorithms have been proposed for predicting defect-proneness. Source code metrics such as complexity and cohesion metrics are widely used for defect prediction, since there is a common understanding that complicated software may yield more defects. For example, simple complexity metrics such as Line Of Code (LOC) can be used to predict defect-proneness of components. Basili *et al.* [3] used Chidamber and Kemerer metrics, and Ohlsson [21] used McCabe’s cyclomatic complexity for defect prediction. In case of cohesion, Kuljit investigated the design level class cohesion metrics [17]. However, these published metrics related with defect prediction are complex and disparate, and no up-to-date comprehensive picture of the current state of defect prediction exists [6, 9].

Change history based metrics also have been proposed and widely used for defect prediction. Nagappan *et al.* [20] proposed the code churn metric, which is the amount of changed code, and showed that code churn is very effective for defect prediction. Moser *et al.* [19] used the number of revisions, authors, past fixes, and age of a file as defect predictors. Hassan [12] introduced entropy of changes, a measure of code change complexity. Entropy of changes was compared to number of changes and previous bugs and found often to give better results. Kim *et al.* [18] proposed the change classification technique, which involved learning buggy change patterns from history and then predicting whether a new code change would lead to bugs. Zimmermann *et al.* [26] proposed a method for predicting the defect proneness of a file from defect information extracted from the Concurrent Versions System (CVS)/Concurrent Versions Subversion (SVN) repositories and for predicting the defect proneness of a file. The distributions of defects over modules of a large software project are also studied [1].

Recently, much of the industry software engineering research has been conducted using data Mining of Software Repositories (MSR), such as DTSs and version control systems. Therefore, some researchers studied defect life cycles and triage without using suggested metrics. 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.* [15] found that half of defect reports for Mozilla and Eclipse are re-assigned to other developers.

Most of these studies have focused on the classification model to analyze how the code or behavior of a developer affects software quality and to predict whether the resulting code will be buggy or clean, however, these models cannot be used to predict future defect trends for a consecutive product level in the concept of SPLs. A new MC-based prediction model for using SPL knowledge to predict future defect trends is proposed in this paper to overcome this limitation. We have also showed that our proposed model is valid and effective for applying in SPLs.

## 3. Background

### 3.1. Defect Life Cycle in Software Engineering

The Defect Life Cycle (DLC) is the cycle that a defect passes through during its lifetime. The cycle starts when a defect is found and ends when it has been retested, resolved, and closed. Bugzilla [5], Git [10], and Jira [16] are examples of DTSs in current use. There are some minor differences between these DTSs, but their major defect transition states, identified by Zeller [25], are shown in Figure 1.

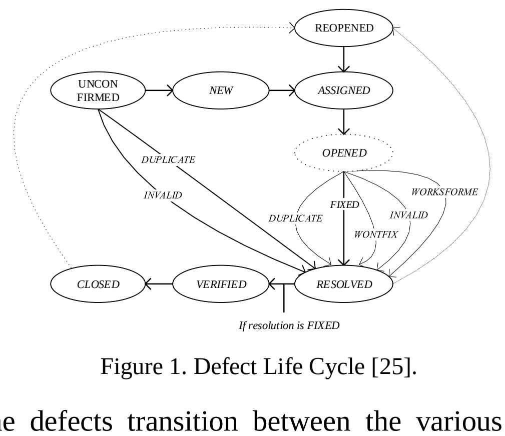

Figure 1. Defect Life Cycle [25].

The defects transition between the various states during their life cycles. When a bug is found for the first time, the tester needs to check to determine whether it is a valid defect. After confirmation, the defect is submitted to the DTS with the status, "New." Once the defect has been assigned to the correct developer, it transitions to the "Assigned" state. The defect is forwarded automatically or by a manager to the designated person in the software development team and he or she changes it to the "Opened" state to indicate that action is being taken to find a solution. When the developer has determined the root cause and found a solution, the state is changed to "Resolved," and the software is released to the tester for verification. If a defect is determined to be invalid according to the intended design or an explanation from the development team, its state is also changed to "Resolved." Once the solution has been "Verified" with the resolution that the defect has been fixed, the tester closes the defect by changing it to the "Closed" state. However, in cases where the same defect reoccurs, or closely related quality issues are discovered, the tester reopens the bug and changes it to the "Reopened" state.

The italic-character states in Figure 1 are the main focus of this paper. Usually, the "Reopened" state can be combined with the "Assigned" state, because it can normally be merged with that state.

## 3.2. MC Model

An MC [7] model is concerned with a sequence of random variables (i.e., $X_1$, $X_2$, $X_3$, ...) with the Markov property, namely, that the state in a given time epoch depends only on the state in the previous time epoch as follows Equation 1:

$$P_r(X_{n+1}=x \mid X_n=x_n,\cdots,X_1=x_1,X_0=x_0)=P_r(X_{n+1}=x \mid X_n=x_n) \tag{1}$$

The possible values of $X_i$ form a countable set S called the state space of the chain. MCs are often described using a directed graph, where the edges are labelled by the probabilities of passing from one state to another (e.g., finite state machine), as shown in Figure 2, in which $p_{ij}$ represents the probability of a transition from state $x_i$ to state $x_j$.

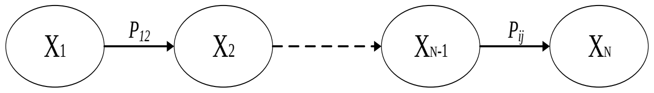

Figure 2. Overview of MC [8].

We can also define the next state at time $t$+1 for the state probability vector $v$ at time $t$ using the following equations:

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

$$v_{t+1}=P\cdot v_t \tag{3}$$

In order for the MC model to apply, state transitions must satisfy the property that the next state depends only on the current state. In addition, all of the transitions in the DTS must be initiated by a developer based only on the current state, rather than a previous state. Therefore, the states of the defects can be expressed using the Markov property, as it is normally defined.

## 3.3. Software Reliability Growth Models

The Software Reliability Growth Models (SRGM) has been used as the most important and successful predictor of software quality. It attempts to correlated defect detection data with estimated residual defects and time. These models are grouped into concave and S-shaped models on the basis of assumption about failure occurrence pattern as shown in the Figure 3 [24].

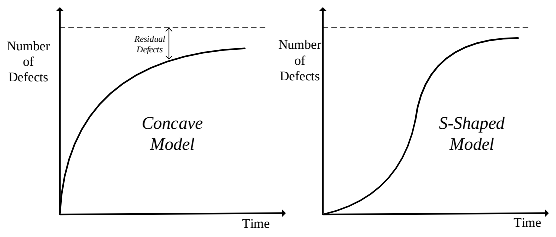

Figure 3. Concave and S-shaped SRGM [11].

In the concave shaped models, the increase in failure intensity reaches a peak before a decrease in failure pattern. Therefore, the concave models indicate that the failure intensity is expected decrease exponentially after a peak is reached. On the other hand, the S-shaped model assumes that the occurrence pattern of cumulative number of failures is S-shaped: initially the testers are not familiar with the product, then they become more familiar and hence there is a slow increase in fault removing. As the tester's skill improves the rate of uncovering defects increases quickly and then levels off as the residual defects become more difficult to remove.

In this paper, we applied Goel-Okumoto (G-O) and Confidence Interval (CI) model for applying it to predict defect growth prediction because our case studied follows the characteristic of concave model [11]. The G-O model is one the commonly used SRGM, which is defined as follow Equation 4:

$$\mu(t)=a(1-e^{-bt}), a>0, b>0 \tag{4}$$

Where, $\mu(t)$ represent the cumulative number of defect through time $t$, $a$ is expected total of number of defects and $b$ is shape factor for representing the rates at which failure rate decreases.

## 4. Proposed Defect Prediction Model

We propose a Defect Prediction Model (DPM) that is able to predict the number of defects that will occur in a series of products and the manner in which these defects will transit through the various states. We hope that this model can also be used, with additional research, to evaluate the performance of the development team or individual developers.

Figure 4 shows the overall structure of the DPM. First, we must select a product that has a platform-based model and a series of products. Second, we mine and refine information from a history log of the DTS. The log contains many types of information, such as current and previous status, importance, currently assigned developer, history of change times, and the like. These refined logs (training datasets) can be attached to the DLC states in accordance with their transitions. Third, the transition probability matrix can be modelled and tested with the test dataset for correctness and validity.

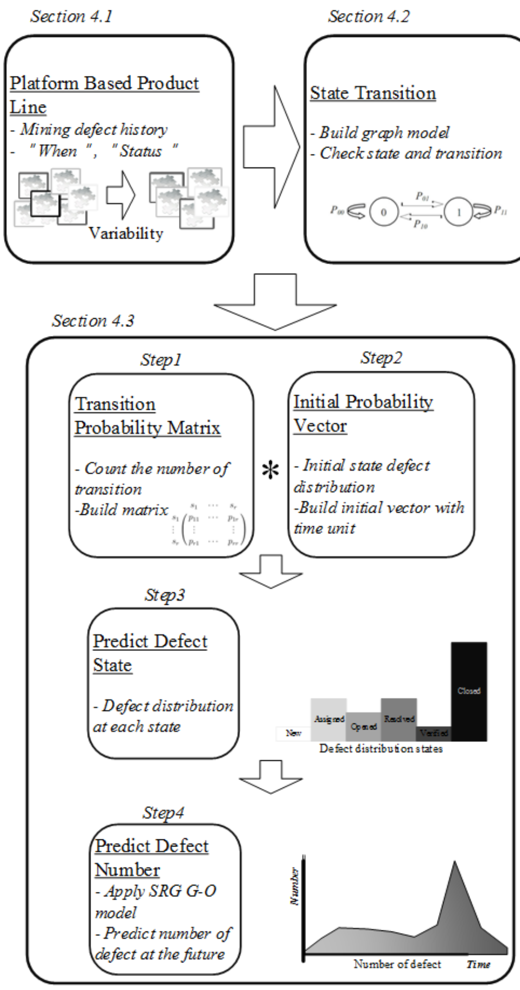
Section 4.1
Platform Based Product Line
- Mining defect history
- "When", "Status"
Variability
Section 4.2
State Transition
- Build graph model
- Check state and transition
P00
0
P01
P10
1
P11
Section 4.3
Step1
Transition Probability Matrix
- Count the number of transition
-Build matrix
*
Step2
Initial Probability Vector
- Initial state defect distribution
-Build initial vector with time unit
Step3
Predict Defect State
- Defect distribution at each state
New
Assigned
Opened
Resolved
Verified
Closed
Defect distribution states
Step4
Predict Defect Number
- Apply SRG G-O model
- Predict number of defect at the future
Number
Number of defect
Time

Figure 4. Overall structure of DPM.

The procedures for building the proposed model are as follows:

1. Select a software project that has a platform-based model and a series of products.
2. Mine a recorded dataset from repositories for a period of time for defect states (when, status) information.
3. Build the transition probability matrix with initial probability vector of test data.
4. Validate the matrix with test data.
5. Predict the total number of defects, the number of instances of each state at specific times, and the total number of closed defects.

Each rounded rectangle in Figure 4 is illustrated in detail in the next section.

## 4.1. History Log of Software Repository

Rounded rectangle section 4.1 in Figure 4 shows the platform-based product, the feature that changes in consecutive products and the defect repositories related to each product. Table 1 gives an example of a history log from Bugzilla stored at the repositories. The "Status" field keeps track of the defect state over the time "When" the bold italic characters show the transitions of the defect's state. The example defect was moved to the resolved state on November 19, 2012, reopened on December 1, 2012, reassigned on December 3, 2012, and finally closed again on December 4, 2012. In addition, the log contains questions, such as why this defect was reopened and why it took a day for the defect to be reassigned to the correct developer.

Table 1. History log of Bugzilla defect (#394495).

| Who | When | What | Removed | Added |
|---|---|---|---|---|
| daniel_megert@ch.ibm.com | 2012-11-19 06:57:25 | **Status** | **New** | **Resolved** |
| | | CC | | daniel_megert@ch.ibm.com |
| | | Resolution | --- | WorksForme |
| | | Summary | can't open editor from "find references" search result | [search] can't open editor from "find references" search result |
| eclipse.rc@gmail.com | 2012-12-01 19:47:13 | Status | Resolved | Reopened |
| | | Resolution | WorksForme | --- |
| daniel_megert@ch.ibm.com | 2012-12-03 02:59:46 | Keywords | | needinfo |
| | | Status | Reopened | Assigned |
| daniel_megert@ch.ibm.com | 2012-12-04 03:30:48 | Status | Assigned | Closed |
| | | Resolution | --- | Duplicate |

For the model proposed in this paper, we need to collect data related to states and to the timings of transitions from one state to another ("When", "Status").

## 4.2. Graph Model for DLC

The transitions in each log can be mapped into the graph model based on the predefined time period. The time base can be adjusted after considering the number of history logs and total development periods of the product. Figure 5 describes the defect state transition graph that maps the states in the DLC to the Markov graph model. We used the same naming conventions for the states as for the names in Figure 1. However, some states in Figure 1 were reorganized and combined during the mapping process, because they are not intended to be predicted, or because some states, such as "Reopened," can be considered to be the same as other states. Each of the defect states has a state label with a series of sequential numbers. The arrows between pairs of states represent transition probabilities from previous states to next states.

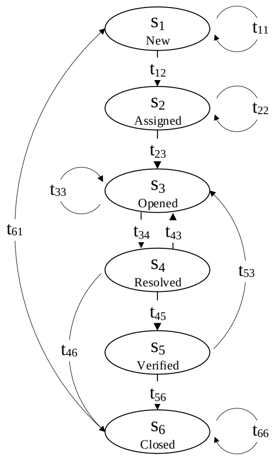

Figure 5. Defect states expressed with graph model.

For example, the transition $t_{34}$ signifies that the developer found a solution and changed the state from "Opened" to "Resolved." If another defect is found related to the current solution, the state returns to "Opened" as $t_{43}$. In addition, the arrow returning to itself signifies that the new state is the same state as the previous state over the predefined time period, after checking "When" information in Table 1. The transition $t_{22}$ signifies that the defect was not assigned to the correct developer and has been reassigned to another developer or to more than one developer. Some arrows can also point back to previous states or skip the next state. $t_{53}$ represents situations where defects have not been resolved completely, such as situations where the defect still exists, reoccurs, or leads to other closely related defects.

The states and transitions are as follows:

- $s_1$: the new state is the state for newly reported defects.
- $s_2$: the assigned state is the state in which a developer has been assigned to check a defect.
- $s_3$: the opened state is the state in which a developer has confirmed a defect.
- $s_4$: the resolved state is the state in which a developer has found a solution.
- $s_5$: the verified state is the state in which a solution is undergoing verification.
- $s_6$: the closed state is the final state in the DLC.
- $t_{11}$, $t_{33}$, $t_{66}$: the defect stays in the same state for longer than the predefined time period.
- $t_{22}$: the defect is assigned to a developer and then is reassigned to another developer.
- $t_{46}$: the defect transitions directly from resolved to closed. This can occur when the defect is not a real defect. This can be due to misunderstanding of the intended behaviour or concept on the part of the tester.
- $t_{53}$: the defect is still alive. This can occur if the tester finds that the issue has not been fixed, or if the tester finds another way to recreate the issue.
- $t_{61}$: the original defect is closed, but another defect is found while testing the solution for the original defect.

Based on this graph model, we can say that if $S$ is a set of defect states and $T$ is occurrence data of defect transitions, then $S$ and $T$ can be defined as follows:

$$S=\{s_1, s_2, \cdots, s_n\},\ T=\{t_{11}, t_{12}, \cdots, t_{61}, t_{62}, \cdots, t_{nn}\} \tag{5}$$

## 4.3. Building the Proposed DPM

The state transition probability matrix for rounded rectangle section 4.3 in Figure 4 is a square matrix describing the probabilities of passing from one defect state to another. To obtain the transition matrix, the following steps are performed:

- *Step 1.* Define State and Build Transition Matrix: the defect states are listed by mapping the transition occurrence data of each defect state to another state. The matrix is constructed by counting the number of steps from one defect state to another. The state transition probability matrix (rounded rectangle 3 in Figure 4 can be expressed as:

$$P_{platform}=\begin{pmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ t_{21} & t_{22} & \cdots & t_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ t_{n1} & t_{n1} & \cdots & t_{nn} \end{pmatrix} \tag{6}$$

$$\sum_{j=1}^{n} t_{1j}=1,\ \sum_{j=1}^{n} t_{2j}=1, \cdots,\ \sum_{j=1}^{n} t_{nj}=1 \tag{7}$$

In each row, the probabilities of moving from the state represented by that row to the other states are shown. Thus, the rows of a transition matrix each add up to one.

- *Step 2.* Obtain Initial Probability of Predicting Product: to obtain the initial probability vector representing the occurrence probability of each defect state being in the initial state, the recent defect state occurrence data in the predicting product are used, which can be divided by the unit of time such as week, month or year depend on the applied domain's characteristics. For example, the domain that the numbers of defect are quite lots, and the defect transitions are changed actively, can be divided by weekly basis. The initial probability vector is calculated using Equation 8, satisfying condition 9.

$$P_{initial}\ (s_1, s_2, \cdots, s_n)=P_{initial}\left(\frac{\alpha}{F}, \frac{\beta}{F}, \cdots, \frac{\gamma}{F}\right) \tag{8}$$

$$F=\sum_{i=1}^{n} f_i=\alpha+\beta+\cdots+\gamma \tag{9}$$

Where $\alpha$, $\beta$, and $\gamma$ represent the number of transitions for each state $s_1$, $s_2$, through $s_n$, can be denoted as $f_i$, $i$

represent each states. Therefore, *F* can be the sum of all transition number during the time unit.

The initial probability $P_{initial}(s_i)$ for each state $s_i$ satisfies the Equation 10 because the sum of initial probabilities must be one.

$$\sum_{i=1}^{n} P_{initial}\ (s_i) = 1 \tag{10}$$

- *Step 3*. Prediction of Defect State Distribution: the probability of defect transition is estimated, predicting the defect in the consecutive product line, using the transition matrix created in step 1 and initial probability vector created using the consecutive SPL in step 2.

$$P(s_1, s_2, \cdots, s_n) = P_{initial}(s_1, s_2, \cdots, s_n) \times P_{platform} \tag{11}$$

where *n* is the number of states for representing DLC states, $P_{intiial}(s_n)$ is the initial probability for the consecutive product line, $P_{platform}$ is the state transition probability matrix calculated using Equation 6 and $P(s_n)$ is the next probability of defect transition. We can also calculate a specific defect transition state using Equation 12.

$$P(s_k) = \sum_{i=1}^{n} P(s_i) P_{ik} \tag{12}$$

Where *k* is a specific defect state, such as “Closed” $P_{ik}$ is element of the transition matrix for specific defect state.

- *Step 4*. Prediction of Total Number at Defect States: to predict the total number defect at each state such as closed states at a future time, we need to estimate the number of defects at each time over the entire development period. With defects newly reported at every time, $d(t)$, representing the number of defects at time *t*, can be represented as follows

$$d(1), d(2), \cdots, d(t), \cdots, \tag{13}$$

The total number of defect at each time, $d(t)$, is represented as a series, which cumulative number of all of previous number.

$$d(1), d(1) + d(2), d(1) + d(2) + \cdots + d(t) \tag{14}$$

Thus, we can represent the total number of defects in the end of time as a series that describes the growth of defects over time.

$$TotalNumberofDefect = \sum_{i=1}^{t} d(t) \tag{15}$$

To predict growth of defect over time, we adopt the Goel-Okumoto (G-O) model. We can predict the number of defects at each state using Equations 4 and 11 as follows:

$$M_{t+1} = P \cdot \mu_{t+1} \tag{16}$$

Where *P* represents defect state probability vector and the $\mu$ is the total number of defects at time $t$+1. *M* is vector representing the number of defects at each state at time $t$+1.

## 5. A Case Study

### 5.1. Background

As a case study, we evaluated the proposed DPM using three consecutive mobile products that were based on one platform and included two diversified products. Each of these products had been developed over the course of a year. The platform product consists of full-featured mobile devices. The first product project was established with value-engineering concepts to reduce the cost value. Some software-related components were changed or removed. Accordingly, quite a few common parts became variable parts with the software product-line method. In contrast, the second product had exactly the same common parts as the platform product, but with some features added and User Experience Design (UXD) concepts changed. All of the products had been developed in that manner by the same development team.

Figure 6 shows the actual defect distribution ratio for the series of products on a month-by-month basis. The defect distribution is slightly different for each consecutive product. For example, the test team takes a while to report defects of the platform product, and further defects remain undiscovered for some time owing to software instability. At some point, the defects peak and then decrease over two months, approaching zero. In contrast, many of the first product’s defects remain, as compared with those of the platform product, owing to changes in commonality and removal of features. The second product’s defects were normally distributed, similarly to the platform product’s, over the entire development period.

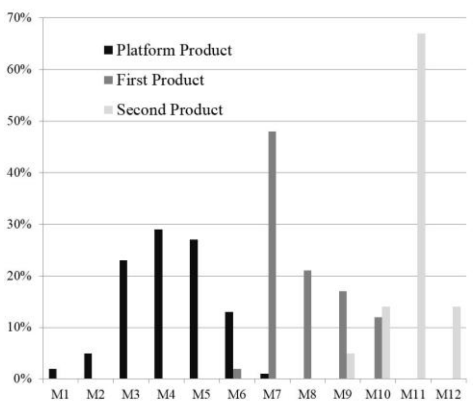

Figure 6. Monthly actual defect distribution for case study.

A dataset was gathered using an in-house DTS. We used this dataset to build the proposed model and validate it by comparing the actual and predicted results.

## 5.2. Predicting Defect State Transitions

The graph model of the platform product is as shown in Figure 5. To gather information about the defect states for each product, we collected and refined the defect transition state data from DTS.

- *Step 1*. After counting the number of transitions between states for each month, we calculated the total number of transitions for the entire development period. Using this number, we determined the transition probability matrix for the platform product. The state transition probability matrix, $P_{platform}$, for our case study is calculated as follows.

$$P_{platform} = \begin{pmatrix} .07 & .93 & 0 & 0 & 0 & 0 \\ 0 & .48 & .52 & 0 & 0 & 0 \\ 0 & 0 & .21 & .79 & 0 & 0 \\ 0 & 0 & .05 & 0 & .87 & .08 \\ 0 & 0 & .26 & 0 & 0 & .74 \\ .16 & 0 & 0 & 0 & 0 & .84 \end{pmatrix} \tag{17}$$

The entries in the transition matrix must satisfy condition 6, that each must sum to one.

- *Step 2*. The most recent transition data, those for one week unit in this study, are used to calculate the initial probabilities for the first product. The most recent transition count is shown during a first week.

$$f_1 = 7, f_2 = 9, f_3 = 28, f_4 = 17, f_5 = 0.5, f_6 = 0.5,\ F = 62$$

The initial probabilities of products are calculated as follows.

$$P_{initial}^{first} = \left(\frac{7}{62}\ \frac{9}{62}\ \frac{28}{62}\ \frac{17}{62}\ \frac{0.5}{62}\ \frac{0.5}{62}\right) = \left(0.11 \quad 0.15 \quad 0.45 \quad 0.27 \quad 0.01 \quad 0.01\right) \tag{18}$$

- *Step 3*. Based on results 17 and 18, the next month's defect state distribution can be estimated using the probability transition matrix and the initial probability, as follows.

$$P^{first} = \left(0.0093 \quad 0.1743 \quad 0.1886 \quad 0.3555 \quad 0.2349 \quad 0.0374\right) \tag{19}$$

- *Step 4*. From these results, the first product's probabilities of closed and resolved states in the next week are 0.0374 and 0.3555, respectively. To estimate the number of closed defects in the next week, the closed defect number of the platform product was counted as 2,784 for *C(S$_k$)*.

$$d_2 = P(s_6)C_{platform}(s_6)(t) = 0.0374 \times 2784 \cong 104 \tag{20}$$

The closed defect number in the next week can be predicted as approximately 104.

With predicted the distribution of defects at each states, we predict the total number of defects by applying the growth of defect numbers. By building G-O models with Equations 4 and 16 using Matlab over weekly basis, we can estimate the total number of closed defects for the product line during the development period.

## 5.3. Results

Figure 7 shows the predicted and actual defect distributions for all states in the second month for the first product. For example, the rates of new and closed defects were predicted to be 1% and 3%, respectively, but the actual rates were 5% and 9%, respectively. Similarly, Figure 8 shows the predicted and actual defect distributions for all states in the second month for the second product. The actual and predicted new defect results are similar, and the gap in the closed rates is also reasonable.

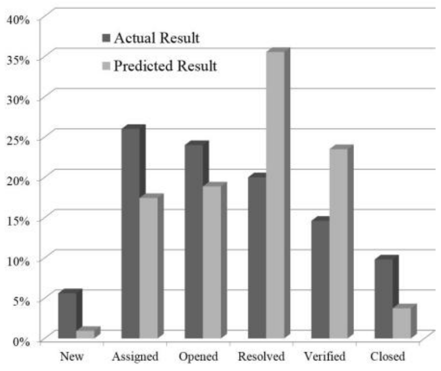

Figure 7. Defect distribution states for first product in next month.

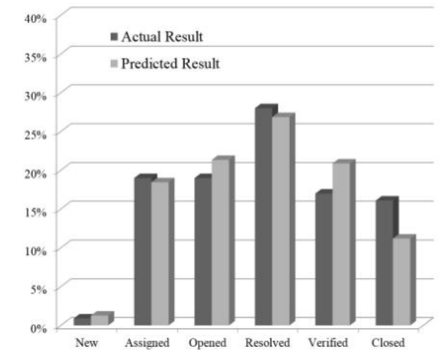

Figure 8. Defect distribution states for second product in next month.

Next, we evaluate the effectiveness of using the probability matrix with the platform product defect history to predict the future number of closed defects. Figures 9 and 10 show the distributions in the number of actual and predicted closed defects for the first and second products, respectively. Clearly, the predicted results for the first product are not as good as those for the second product, as compared with the actual results. In case study, we predicted only the closed states, because the numbers of closed states are expected to be the same as the numbers of new states, which should be closed at the end from the software quality point of view.

The results in this case study show that the proposed model is suitable for the second product, but not necessarily for the first. The proposed prediction model can be applied to SPLs that possess commonalities. The gaps between the actual and predicted results are likely to be relatively high for the first product. As we mentioned, quite a few common aspects of the first product are changed to variability, as compared with the second product, resulting in gaps in the defect prediction results for the first product. One of the big changes with regard to variability comparing with the platform product is resolution changing of display. It drives some commonality to variability such as system software, usability experience evaluation and design of user interface. Hence, it made lot of duplicated, invalid defects related with hardware components in actual early defect reports.

In case of second product, whole of hardware components are exactly same as platform product. Although, there are some difference ratios for some states comparing with actual result, the prediction results are promising in that the proposed model can predict defect states for the second product, which has commonalities with the platform product. If the predicted product lacks sufficient commonality, or the development team differs from that of the platform product, the proposed model fails to predict potential defects.

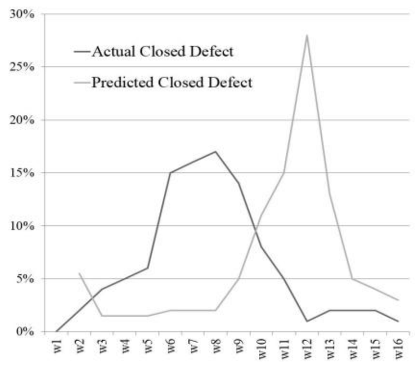

Figure 9. Actual and predicted closed defect number for first product.

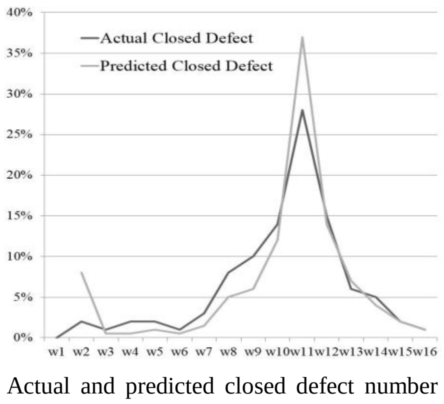

Figure 10. Actual and predicted closed defect number for second product.

## 5.4. Discussions

To measure the precision sensitivity of proposed model with respect to the variability in SPL, we derive some metric to evaluate the variability degree of case study.

It is desirable to measure the degree of variability, important factor for reusability in SPL, to figure out the relevance between variability change and precision of the proposed prediction model. Recently, the metrics for evaluating SPL architecture are discussed in several publications. One of them, Berger [4] investigated the assessment of product variants to extract a product line and propose a set of metrics that enable the software architects and project managers to estimate the variability.

To measure the variability of an experimented case study, we select metrics such as Product-related Reusability (PrR), Impact of Product-related Reusability (IPrR), Individualization Ratio (IR), Reusability Benefit (RB) and Relationship Ratio (RR). Table 2 shows the results of metrics for the case studied products.

Table 2. Results of metrics for the case study.

| | Platform Product | First Product | Second Product | Platform /First | Platform /Second |
|---|---|---|---|---|---|
| **PrR** | 0.94 | 0.54 | 0.87 | | |
| **IPrR** | 0.12 | 0.61 | 0.32 | | |
| **IR** | 0 | 0.76 | 0.21 | | |
| **RB** | | | | 0.87 | 0.43 |
| **RR** | | | | 0.65 | 0.91 |

The Impact of Commonality (IoC) metric has a value of 0.76 which means that the studied products have many shared common components. The platform product has highest PrR, it has a baseline of consecutive products and makes contribution for building probability transition matrix. The IPrR is highest at the first product, it mean the impact of all commonly share components by first product is not greater. The ratio IR second product has smaller than first product; it means the second product is similar with platform product. The RB of platform and second product is the smaller than first product. In case of RR, they have higher value which means they share the common components between platform based and second product. So, we can guess the precision of proposed model depend on the variability degree level comparing with platform product used for building transition probability matrix.

## 6. Limitations

We have identified the following limitations:

### 6.1. Non-Platform based or Small-Scale Software Projects

We intentionally chose as a case study a platform-based project with common parts. In addition, the number of defects must be high, and the defect transitions must be changed actively, to ensure the accuracy of the prediction model. The validity of the transition probability matrix is difficult to guarantee for small non-platform software projects. The proposed prediction model can work well, when sufficient data are available in an organization's software repository to support it.

### 6.2. Not Applicable in Case of Team Change

The model we proposed works only if one

development team is involved. If team members are changed, transitions from opened to resolved states are especially prone to differ from those of the team that initially built the prediction model. We need to conduct a sensitivity analysis against development team change to evaluate the sensitivity of our prediction model in the future.

### 6.3. Interference in the Repository Data

In a DTS, there are some transitions that do not follow the DLC. While preparing the input for the proposed model, we found that transitions from new to closed were direct. These took place as a result of removing a feature or changing UXD concepts. If those defects are not filtered, the accuracy of the prediction model might be affected.

## 7. Conclusions

In this paper, we proposed a DPM that predicts how many defects will occur during the development of consecutive software products. The MC and SRG model that were used in our study has been used widely and has proven to be effective for various kinds of information processing. The experimental results demonstrate the effectiveness of the proposed model.

Defect prediction can be used to improve the management of software development efforts. One of the most visible advantages of using defect prediction is the ability to predict trends and directions of defect cycles in software projects. Therefore, we believe that the proposed model can enable managers and project leaders to detect defect trends and to anticipate problems under uncertainty, thereby controlling resources more effectively, gaining insight into the likely quality to be achieved from development efforts, and ensuring the success of development objectives, such as time to market and software quality.

In our next study, we intend to eliminate the statistical inferences in the repository data. The inferences exist in all of the data and they can lead to incorrect results when building the MC model. Also, we aim to investigate some of the states, such as "Reopened" and "Verified," to evaluate and estimate developers' and testers' performance and effectiveness. In addition, the sensitivity analysis between the precision of proposed model and the variability degree in SPL is also needed to clarify the validity of applicable SPL.

## Acknowledgements

This research was supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2012M3C4A7033345).

## References

[1] Andersson C. and Runeson P., "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems," *IEEE Transactions on Software Engineering*, vol. 33, no. 5, pp. 273-286, 2007.

[2] Bach J., "Good Enough Quality: Beyond the Buzzword," *IEEE Computer Society*, vol. 30, no. 8, pp. 96-98, 1997.

[3] Basili V., Briand L., and Melo W., "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, 1996.

[4] Berger C., Rendel H., and Rumpe B., "Measuring the Ability to Form a Product Line from Existing Products," *in Proceeding of the Fourth International Workshop on Variability Modelling of Software-intensive Systems*, Linz, pp. 151-154, 2010.

[5] Bugzilla, http://www.bugzilla.org/, Last Visited 2013.

[6] Catal C. and Diri B., "A Systematic Review of Software Fault Prediction Studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346-7354, 2009.

[7] Ching W. and Ng M., *Markov Chains: Models, Algorithms and Applications*, Springer Science, 2006.

[8] Clements P. and Northrop L., *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[9] Fenton N. and Neil M., "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675-689, 1999.

[10] Git, http://git-scm.com/, Last Visited 2013.

[11] Goel A. and Okumoto K., "Time-Dependent Error-Detection Model for Software Reliability and other Performance Measures," *IEEE Transactions on Reliability*, vol. R-28, no. 3, pp. 206-211, 1979.

[12] Hassan A., "Predicting Faults using the Complexity of Code Changes," *in Proceeding of ICSE*, Washington, pp. 78-88, 2009.

[13] Hribar L., Bogovac S., and Marincic Z., "Implementation of Fault Slip Through in Design Phase of the Project," *in Proceeding of the 31st International Convention on Information and Communication Technology, Electronics and Microelectronics*, Opatija, pp. 134-138, 2008.

[14] Hribar L., "Usage of Weibull and other Models for Software Faults Prediction in AXE," *in Proceeding of the 16th International Conference on Software, Telecommunications and Computer Networks*, Dubrovnik, pp. 157-162, 2008.

[15] Jeong G., Kim S., and Zimmermann T., "Improving Bug Triage with Bug Tossing Graphs," *in Proceeding of ESEC/FSE '09*, Amsterdam, pp. 111-120, 2009.

[16] Jira, http://jira.dspace.org, Last Visited 2013.

[17] Kaur K. and Singh H., "An Investigation of Desing Level Class Cohesion Metrics," *The International Arab Journal of Information Technology*, vol. 9, no. 1, pp. 66-73, 2012.

[18] Kim S., Jr J., and Zhang Y., "Classifying Software Changes: Clean or Buggy?," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181-196, 2008.

[19] Moser R., Pedrycz W., and Succi G., "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction," *in Proceeding of the 30th International Conference on Software Engineering*, Leipzig, pp. 181-190, 2008.

[20] Nagappan N., Ball T., and Zeller A., "Mining Metrics to Predict Component Failures," *in Proceeding of the 28th International Conference on Software Engineering ICSE '06*, Shanghai, pp. 452-461, 2006.

[21] Ohlsson N. and Alberg H., "Predicting Fault-Prone Software Modules in Telephone Switches," *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886-894, 1996.

[22] Staron M. and Meding W., "Predicting Short-Term Defect Inflow in Large Software Projects-an Initial Evaluation," *in Proceeding of the 11th International Conference on Evaluation and Assessment in Software Engineering EASE*, Swindon, pp. 33-42, 2007.

[23] Weib C., Premraj R., Zimmermann T., and Zeller A., "How Long will it Take to Fix this Bug?," *in Proceeding of the 4th International Workshop on Mining Software Repositories MSR*, Washington, pp. 308-318, 2007.

[24] Wood A., "Software Reliability Growth Models," Technical Report, 1996.

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

[26] Zimmermann T., Premraj R., and Zeller A., "Predicting Defects for Eclipse," *in Proceeding of the International Workshop on Predictor Models in Software Engineering*, San Francisco, pp. 91-97, 2007.

**Chang-Kyun Jeon** received his Ph.D. degree in the College of Information and Communications at Korea University and also is a principal engineer at Samsung Electronics. His interests include software product lines, defect prediction, and embedded software engineering. He received his M.S. degree in Control and Instrumentation Engineering from Kwangwoon University in Seoul, South Korea.

**Neung-Hoe Kim** is a Ph.D. candidate in the College of Information and Communications at Korea University. His interests include requirements engineering, value-based software engineering, software engineering economics, and embedded software engineering. He received his M.S. degree in Computer Science from Korea University in Seoul, South Korea.

**Hoh In** is a professor in the College of Information and Communications at Korea University. His primary research interests are embedded software engineering, social media platform and service, and software security management. He has published over 100 research papers and earned the most influential paper for 10 years award at ICRE 2006. Prof. In was an Assistant Professor at Texas A&M University. He received his Ph.D. in Computer Science from the University of Southern California (USC).