Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach

Ali R. Sharafat and Ladan Tahvildari
Dept. of Electrical and Computer Engineering
University of Waterloo, Ontario, Canada, N2L 3G1
{arsharaf, ltahvild}@uwaterloo.ca

***Abstract*— An estimation of change-proneness of parts of a software system is an active topic in the area of software engineering. Such estimates can be used to predict changes to different classes of a system from one release to the next. They can also be used to estimate and possibly reduce the effort required during the development and maintenance phase by balancing the amount of developers' time assigned to each part of a software system. This research work proposes a novel approach to predict changes in an object-oriented software system. The rationale behind this approach is that in a well-designed software system, feature enhancement or corrective maintenance should affect a limited amount of existing code. Our goal is to quantify this aspect of quality by assessing the probability that each class will change in a future generation. Our proposed probabilistic approach uses the dependencies obtained from the UML diagrams, as well as other code metrics extracted from source code of several releases of a software system using reverse engineering techniques. These measures, combined with the change log of the software system and the expected time of next release, are used in an automated manner to predict whether a class will change in the next release of the software system. The proposed systematic approach has been evaluated on a multi-version medium sized open source project namely JFlex, the Fast Scanner Generator for Java. The obtained results indicate the simplicity and accuracy of our approach in the comparison with existing methods referred in the literature.**

***Index Terms*— measurement applied to SQA and V&V, reverse engineering, software maintenance, probability and statistics, software change prediction**

I. Introduction

Software engineering deals with "the construction of multi-version software" which will undergo a number of revisions either to enhance functionality or to fix bugs [1]. The modularity of object-oriented programs aims to reduce the impact of addition of new functionality or bug fixes in such systems. If the modification of a class method imposes code changes to a number of existing classes, the object-oriented design will be of limited value [2]. In a nutshell, predicting source code changes has become a crucial factor, since a number of studies conclude that the largest percentage of software development effort is spent on rework and maintenance.

Based on "A Probabilistic Approach to Predict Changes in Object-Oriented Software Systems," by Ali R. Sharafat and Ladan Tahvildari, which appeared in the Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR), Amsterdam, Netherlands, March 2007.

This work was funded by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

This research work aims to address the problem of correctly predicting changes in a software system. Correct prediction of changes can help managers to allocate resources more appropriately; this results in a reduction of costs associated with software development and maintenance, as well as more evenly distributed workload among the developers and testers. Correct prediction of changes also brings some insight on the design of the software. For example, if changes in one module have a considerable effect on other parts of the system, then the coupling between the modules may need to be reduced. We aim to determine the probabilities of change of classes within a system, which can also be used to assist maintenance and to observe the evolution of stability through successive generations.

The goal of this research is to predict the probability that each class will change in a future generation. Our proposed probabilistic approach can be applied when a few successive versions of an application are available. In order to calculate these probabilities, axis of time, through which a change in one class can affect another class of the design, is identified. We apply our technique on an object-oriented open source project, JFlex [3]. Obtained results validate that the proposed analysis offers improved prediction accuracy compared to a model that simply considers information from changes in past generations. It should also be noted that our proposed model provides the prediction in an automated manner. This is a considerable improvement over the related work, as discussed later in the paper.

This paper is organized as follows. Section II discusses previous approaches which directly or indirectly address the same problem in the literature. The analysis process for predicting source code changes along with some background information about the probability theory and the notation used in this paper are presented in Section III. A case study and its statistical analysis results are discussed in Section IV. Finally, Section V provides conclusions and Section VI gives some insight into future work.

II. Related Work

Several researchers have proposed the use of historical data related to a software system to assist developers gain a better understanding of their software system and its evolution. Zimmermann *et al.* [4] and Ying *et al.* [5] use

© 2008 ACADEMY PUBLISHER

data gathered from logs of version control systems in order to predict changes in several open source software systems. Their goal is very similar to ours, but they exploit a much larger set of input data than we do. Due to this fundamental difference, we do not compare our methods with [4] or [5]

Arnold and Bohner give an overview of several formal models of change propagation [6]. The models propose several tools and techniques that are based on code dependencies and algorithms such as slicing and transitive closure to assist in code propagation.

In [7], a change impact model has been proposed for changeability assessment with primary goal to investigate the relationship between existing design metrics (e.g., Weighted Methods per Class) and the impact of change. Although it is useful to know which classes would be impacted by a given change, one has to know the actual changes that have occurred in a system to assess the probability of change for a certain class. The relationship between metrics and maintainability has also been studied in [8]. In [9], a set of algorithms that determine what classes are affected by a given change is proposed. The methodology represents a system as a set of data dependency graphs, which is an effective approach for object-oriented designs. However, as in any change impact model, reports about the potential impact of a given change can be generated only after the user explicitly specifies the changes.

Briand *et al.* [10] empirically investigate whether coupling measures are related to ripple effects, using a commercial OO system. The aim is to rank classes according to their probability of containing ripple effects, while the approach proposed in this paper aims at identifying classes that are highly probable to change in a future generation, regardless of whether the change is internal or due to a ripple effect. An advantage of using coupling measures is that they are inherently related to ripple effects since common changes are usually due to relationships between classes. However, ripple effect-prone classes cannot be used for predicting whether they will change in a future release since changes originating in the class itself are neglected.

Hassan and Holt [11] tackle this problem in a different manner. They devise a technique in which a stand-alone system learns from changes to a software product, by associating the files that change in one commit [12] to a version control system. As developers modify the code in parts of that product, their proposed learning system suggests other files which may need to be modified due to the propagation of changes. Kim *et al.* [13] use a similar method by *caching* classes that have recently caused changes and faults. These classes along with those *nearby* them are considered prime candidates for causing faults soon.

Graves *et al.* [14] have a slightly different goal of predicting *faults*, which are a subset of changes, in aged software systems. They find the change-history of the system to be a better predictor than code metrics. In that model, Graves *et al.* assign weights to the perceived changes, with the most recent receiving the most. These weighted values provide a trend that is used to predict the number of faults in an upcoming period.

Mockus and Weiss [15] attempt to predict faults in a software system based on information extracted from changes to the system (e.g., lines of code modified, the changed components, etc.). This approach differs from many of other the related work in the respect that it uses changes to the state of the system, rather than the state itself. The authors apply their method on several updates to a software system to predict likely faults in future updates.

Kagdi and Maletic [16] propose combining results from impact analysis with those from mining source repositories [4] to achieve a better accuracy in prediction of future changes. A case study has not been presented in that work, but the foundations of the framework are described there. We use a similar method and combine the metric-based and history-based probabilities to predict changes.

The work of Basili *et al.* [17], which focuses on validation of a software metrics suite [18], gives correlations between values of a software metric suite and the number of bugs and fixes that appear in a software system. While this study is not directly related to prediction of changes, it does provide useful information as to which metrics are good indicators of a change due to a software bug.

A recent work of Girba *et al.* [19] proposes an approach to summarize the changes in the history of a system that can offer a solid basis for starting a reverse engineering effort. The methodology consists of identifying the classes that were changed the most in the recent history and at the same time checking whether the same classes are among the most changed ones in the successive versions. However, only the addition or removal of methods is considered as changes. Arisholm *et al.* [20] investigate the use of dynamic coupling measures as indicators of change proneness. Their approach is based on correlating the number of changes to each class (a continuous variable which represents change proneness) with dynamic coupling measures and other class-level size and static coupling measures. Consequently, it cannot be considered a prediction model since no attempt is made to correlate the proposed measures with changes/no changes in the next generation. In addition, requirement changes have been factored out since they are not driven by design characteristics.

Our research was inspired by the work of Tsantalis *et al.* [21], in which they propose a technique for prediction of changes in an object-oriented system. Their underlying principles are very similar to that of our work. Tsantalis *et al.* divide changes to a class into two categories: internal and external. Internal changes to a class are those that are initiated within that class, and external changes are those that occur due to changes in neighboring classes. Thus, the probability of change of a class is the probability of the union of internal and external changes.

The values of *probability of internal change of a class* in [21] are defined as the percentage of past releases

© 2008 ACADEMY PUBLISHER

in which there was an internal change in that class. Classification of changes into internal and external is done by manual inspection of the source code from previous releases of the software product which limits extensibility of that model. In Section III-A, we discuss a technique that estimates the probabilities of internal change, based on OO metrics that can readily be extracted from the source code. Our proposed approach automates the process of calculating the probability values.

External changes to a class A depend on two events: i) a neighboring class of A changing, and ii) that change propagating to class A. Thus, to calculate the probability of an external change in class A due to a change in its neighbor B, the probability of a change propagating from B to A has to be known. Tsantalis *et al.* assign a uniform probability value for propagation of changes between all pairs of classes that are dependent on each other (two classes are defined to be dependent, if there exists a direct dependency between them in the UML diagram of the software system). That probability value is defined as the percentage of changes in the past releases of the system that have propagated, and is obtained by manual inspection of changes. In Section III-B, we propose methods to estimate the propagation probabilities between pairs of classes, based on the number and types of dependencies between them. Our methods not only automate the process of obtaining these probabilities, but also yield more accurate predictions about changes in the next release of the system.

The construction presented in [21] leads to solving a nonlinear system of equations to get the probabilities of change. Solving that system of equations becomes difficult, if there exists a set of classes which form a cycle in the dependency graph of the software system. Tsantalis *et al.* present an approximation method to bypass that difficulty. We propose additional techniques in Section III-C to deal with cyclic dependencies. We also bring into the picture the axis of time, where we consider the time between consecutive releases as a parameter that will affect the probabilities of change (see Section III-D). We observe through empirical evaluation, that the inclusion of time as a parameter results in better predictions about changes in future releases.

Prior to [21], Xia and Srikanth [22] introduced the idea that a change initiate in a class and then *ripples* to that class's neighbors with a probability r. Thus, as we get further from the changed class, the probability of a propagated change reduces by factors of r. The value of r mirrors the conditional probabilities defined in [21].

III. The Proposed Probabilistic Approach

In order to determine which classes will change in the next release of a software system, we propose a probabilistic approach which uses the change history as well as the source code of the system. For notational consistency, we will use $\mathbb{P}(E)$ to denote the probability of event E throughout this paper. As shown in Fig. 1, the first stage of the process estimates the likelihood that a class will be modified due to changes originating from the class itself. Such an estimation can be obtained based on a metrics suite that can measure the relevant features of the source code. We call this value the *probability of internal change of the ith class*, $i = 1, 2, \dots, N$, denoted by $\mathbb{P}_s(\mathrm{IC})$, where the subscript s indicates that the probability value is based on the source code. At the second stage of the process, we extract the dependencies between classes using UML diagrams representing the design of the system. Based on the extracted data, we approximate the probabilities that a change would propagate from one class (j) to another class (i). This value is refereed as the *propagation probability*, α_{ij}.

Ideally, we would like to map each kind of dependency to a unique propagation probability value, but when we have multiple dependencies between classes, we need to use alternative methods to obtain a single propagation probability between pairs of classes. As shown in Fig. 1 at the third stage of the process, the values of all $\mathbb{P}_s(\mathrm{IC})$ and α_{ij} are used to find the total probability of change of the class obtained from source code namely $\mathbb{P}_s(\mathrm{TC}_i)$. This probability represents the likelihood of a class being modified due to changes originating from itself, and those which propagate from the neighboring classes. It is assumed that internal and propagated changes are two independent events. Furthermore, we assume that propagation of changes from different classes are independent as well. These assumptions simplify the calculation of intersection and union of those events. Using Bayes' theorem, $\mathbb{P}_s(\mathrm{TC}_i)$ can be calculated as follows:

$$\mathbb{P}_s(\mathrm{TC}_i) = \mathbb{P}\left(\mathrm{IC}_i \cup \left(\bigcup_{j \neq i} \mathrm{C}_{i|j} \cap \mathrm{TC}_j\right)\right), \qquad (1)$$

where IC_i represents an internal change in class i, with $\mathbb{P}_s(\mathrm{IC}_i)$ representing the probability of that event; $\mathrm{C}_{i|j}$ represents the propagation of a change from class j to i, with α_{ij} representing its probability; and TC_i represents the total change (internal or propagated) in class i, with $\mathbb{P}_s(\mathrm{TC}_i)$ as its probability. Again, the subscript s indicates that the probability values are based on the source code.

As an example, consider a simple system consisting of three classes, A, B, and C indexed by $i = 1, 2, 3$, which form a chain of acyclic dependencies between them namely *C inherits B, which inherits A*, as shown in Fig. 2. We denote classes by circles and dependencies with arrows which point in the direction of change propagation. Assume that all classes have a probability of internal change of 0.5 and that the the propagation probabilities

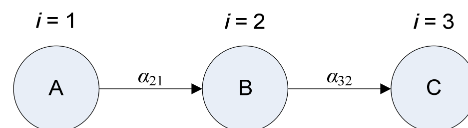

Figure 2. An example of a simple system with three classes forming a chain of dependencies.

© 2008 ACADEMY PUBLISHER

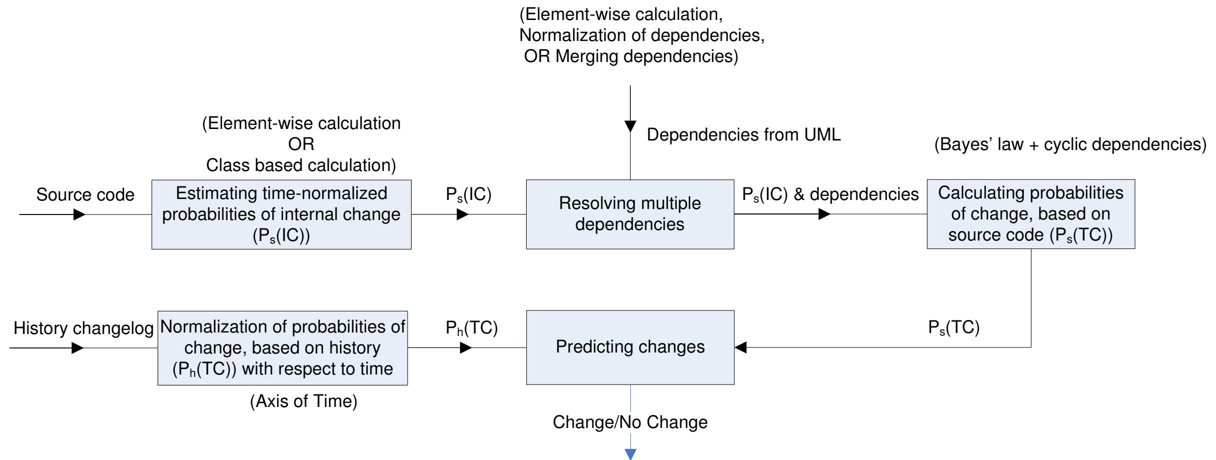

Figure 1. The Block Diagram of the Proposed Probabilistic Process to Predict Changes

are given by $\alpha_{21} = \alpha_{32} = 0.25$. Thus, assuming that the events $\mathrm{C}_{i|j}$ and TC_j are independent for all i and j, the probabilities of change are calculated as follows:

$$\begin{aligned}
\mathbb{P}_s(\mathrm{TC}_1) &= \mathbb{P}_s(\mathrm{IC}_1) = 0.5 \\
\mathbb{P}_s(\mathrm{TC}_2) &= \mathbb{P}\left(\mathrm{IC}_2 \cup \left(\mathrm{C}_{2|1} \cap \mathrm{TC}_1\right)\right) \\
&= \mathbb{P}_s(\mathrm{IC}_2) + \alpha_{21} \cdot \mathbb{P}_s(\mathrm{TC}_1) - \\
&\quad \mathbb{P}_s(\mathrm{IC}_2) \cdot \alpha_{21} \cdot \mathbb{P}_s(\mathrm{TC}_1) \\
&= 0.5625 \\
\mathbb{P}_s(\mathrm{TC}_3) &= \mathbb{P}\left(\mathrm{IC}_3 \cup \left(\mathrm{C}_{3|2} \cap \mathrm{TC}_2\right)\right) \\
&= \mathbb{P}_s(\mathrm{IC}_3) + \alpha_{32} \cdot \mathbb{P}_s(\mathrm{TC}_2) - \\
&\quad \mathbb{P}_s(\mathrm{IC}_3) \cdot \alpha_{32} \cdot \mathbb{P}_s(\mathrm{TC}_2) \\
&= 0.5703
\end{aligned}$$

Note that the calculations were done in a top-down fashion (we calculated probabilities of change in a serial manner, starting with A and finishing with C). This can be done as long as the dependency graph is acyclic. We present methods in Section III-C to calculate the probabilitis when the graph contains cycles.

As shown in Fig. 1, the history change-log of the software system can be used to get another measure of the probability of change. In order to make use of this information, we convert the raw probabilities of change into time-normalized values. The raw probability of change of a class is defined as the percentage of past releases in which that class was changed. The time-normalized probability provides the probability of change of a class in a future release by using the raw probability value and taking into account the length of the Time Between Releases (TBR) of the past revisions and the expected Time To Release of the next version of the system (TTR). In other words, we first attempt to estimate the probability of change of a class in *unit time* and then use the TTR value to estimate the probability of change of that class in the next release; larger values of TTR imply that a change is more probable. We refer to this history-based time-normalized probability as $\mathbb{P}_h(\mathrm{TC}_i)$, where the subscript h indicates that the probability value is based on the history of the system.

Note that neither $\mathbb{P}_s(\mathrm{TC}_i)$ nor $\mathbb{P}_h(\mathrm{TC}_i)$ can provide reliable information alone. This is because the former only considers the structure of the source code, and the latter is only based on the nature of the source code by using the history change log. Thus, after calculating the values of $\mathbb{P}_s(\mathrm{TC}_i)$ and $\mathbb{P}_h(\mathrm{TC}_i)$ for all classes, we average those values and predict as to whether or not each class will change in the future release.

Our approach also uses several other measurements to provide a comparison with other solutions. While the list is as follows, it should be noted that *Overall Accuracy* is considered the most important measure:

- **False Positive Ratio (FPR)**: The percentage of cases where a class was predicted to change, but in fact it did not.
- **False Negative Ratio (FNR)**: The percentage of cases where a class was predicted not to change, but in fact it did.
- **Sensitivity**: The percentage of changes correctly predicted. It is equal to $1 - FNR$.
- **Overall Accuracy**: The weighted percentage of changes and no-changes correctly predicted. It is given by $1 - \frac{FNR+FPR}{2}$.

The details of the proposed methods are elaborated further in the following sections.

A. Estimation of $\mathbb{P}_s(IC)$

In [21] the probabilities of internal change are obtained by manual inspection of the source code from one revision to the next. While this procedure is applicable to small software systems, it becomes very time consuming as the size of the system increases. In this section, we propose methods that will automate this procudure and will use the code metrics and the history-log of the system to provide estimates of the abovementioned probability values. Later, we will show that these methods give prediction that are very close to those obtained by manual inspection.

© 2008 ACADEMY PUBLISHER

The nature of the source code provides useful information regarding the stability of the software system. In order to find an estimate for $\mathbb{P}_s(\text{IC})$, we use a suite of metrics along with a method similar to that of *Hazard Rate Functions* described in [23].

Assume that we have a real valued metric function $X : \mathcal{S} \mapsto [0, +\infty)$, where $\mathcal{S}$ is the set of all the *elements* in the source code (methods, variable, classes, etc.). Assume that we can extract probabilities of internal changes of an element, from its corresponding metric value X.

Let us partition the axis corresponding to the values of X into small (non-overlapping) segments of length dx and compute the probability that we have a change when the value of X falls in $(x, x + dx]$ while there has been no change for $X = x$. To do this, let $F(x)$ denote the probability of change if the value of the metric X is equal to x. Expressing $F(x)$ as the sum of the probabilities with values $f(x)dx$ (union of disjoint events), we have

$$F(x) = \int_0^x f(x)dx$$

We also have,

$$\begin{aligned} &\mathbb{P}(\text{change for } X \in (x, x+dx] | \text{no change for } X = x) \\ = \quad & \frac{\mathbb{P}(\text{change for } X \in (x, x+dx] \cap \text{no change for } X = x)}{\mathbb{P}(\text{no change for } X = x)} \\ \approx \quad & \frac{f(x)}{1 - F(x)} dx \end{aligned}$$

Now, Let

$$\begin{aligned} \lambda(x) &= \frac{f(x)}{1 - F(x)} \\ &= \frac{\frac{d}{dx} F(x)}{1 - F(x)}. \end{aligned}$$

Integrating both sides gives

$$F(x) = 1 - \exp\left(- \int_0^x \lambda(x) \; dx \right). \quad (2)$$

It is assumed that $\lambda(x)$ is a constant-value function. This would simplify the integration in (2) to

$$F(x) = 1 - e^{-\lambda x}. \quad (3)$$

The probability function, F, does not need to be time-normalized as its input only depends on the structure of the code. Therefore, we can assume that F gives the probability of change of a class in unit time if a suitable value for λ is chosen.

Assuming that there are t units of time until the next release (i.e., TTR = t), we have

$$\text{IC}_i = \bigcup_{\tau=1}^{t} \text{IC}_{i,\tau}, \quad (4)$$

where IC_i is an internal change to class i over a period of length t unit times, and $\text{IC}_{i,\tau}$ is an internal change in the jth unity period. Assuming that internal changes in different periods are independent events, we combine (3) and (4) to get

$$\mathbb{P}_s(\text{IC}_i) = 1 - e^{-\lambda x_i t}, \quad (5)$$

where x_i is the metric value of class i. We can define x_i as a combinatin of OO metrics that are good indicators of change-proneness of a class. A simple way of doing so, is by letting x_i to be a linear combination of code metrics corresponding to class i, with each metric weighted according to its correlation with past changes. We can find a suitable value for λ by using some sample values and then predicting changes in the past releases for each value. We choose the value gives predictions that correleate best with the occured changes.

In our approach, we need to select a set of object-oriented metrics that will be used to assess the changes. In order to make a selection, we first need to establish a set of criteria that should guide the selection process. Establishing these criteria requires us to consider and identify which of the metrics can be successfully used in order to assess the changes and to collect proper information from the source code features at the method or class level (depending on our choice of the level of granularity). In this respect, we focus on two criteria : i) the theoretical evaluation of the definition of the metric, and ii) the aspects of changes that we plan to predict.

Table I illustrates our selected metrics at method level which will be used in the proposed approach. Note that as the value of the metrics in Table I increases, so does the probability of change of the methods [5], [21]. Metrics in Table I can be extracted using Borland Together [26]. We have chosen to use metrics at method and data-member level, as they provide more detail about the structure of the code. We can also easily define the corresponding class-level metrics by aggregating the lower level metrics.

B. Resolving Multiple Dependencies

The propagation probabilities are dependent on the type of the relationship between two classes. For the sake of simplicity, we wish to assign a single propagation probability to each kind of dependency. Furthermore, we want to combine multiple dependencies between classes to extract a single propagation probability between them and exploit (1) to find the probabilities of change. Relationships between classes and their respective elements are extracted using an Eclipse plug-in, called Creole [27]. A list of extracted relationships is presented in Table II. These relationships are exported using the Rigi Standard Format (RSF) [28], which is a set of tuples that take the following format: (*FromElement, ToElement, Type of Relation*).

Assigning propagation probabilities becomes problematic when we have more than one dependency between two classes and their elements, as the above notation is only defined for single dependencies. In order to make the most use of the available data, we seek to incorporate all dependencies between any two classes when estimating the propagation probability between them. We propose three techniques to bypass this problem: two rely on

© 2008 ACADEMY PUBLISHER

TABLE I.
A METRIC SUITE USED AT METHOD LEVEL

Metric Name	Description	Definition
AID	Access of Import Data	Number of data members accessed in a method directly or via accessor-methods, from which the definition-class of the method is not derived.
ALD	Access of Local Data	Number of the data members accessed in the given method, which are local to the class where the method is defined.
CC [24]	Cyclomatic Complexity	Number of possible paths through an algorithm by counting the number of distinct regions on a flowgraph, meaning the number of `if`, `for`, and `while` statements in the method's body.
LOC	Lines of Code	Number of lines of code in a method, including comments and white-lines.
MNOB	Maximum Number of Branches	Maximum number of `if-else` and/or `case` branches in the method.
MPC [25]	Message Passing Coupling	Number of method call expressions made into body of the measured method.
NIC	Number of Import Classes	Number of external classes from which the given method uses data.
NOLV	Number of Local Variables	Number of local variables are declared within a method.
NOP	Number of Parameters	Number of parameters that build the signature of a method.

TABLE II.
EXTRACTED RELATIONS USING CREOLE

Relation	Type
accesses	method to attribute
calls	method to method
casts to type	method to class
contains	package to class
creates	method to class
extended by	class to class
has parameter type	method to class
implemented by	class to class
is of type	attribute to class
overridden by	method to method

keeping the probability calculations at the class level, while the other suggests performing calculations at the method and data-member level.

1) Normalization of Dependencies: In this method, a mapping function m is used to map the number of dependencies between classes to a conditional probability value; since dependencies are from an *element* (i.e., data-member and method) to an *element*, m takes the following form :

$$m : \{0, 1, 2, \ldots, n(\text{Elements of } A) \times n(\text{Elements of } B)\} \mapsto \{z \in \mathbb{R} |\ 0 \leq z \leq 1\}$$

where A and B are two classes with x dependencies between them. One simple way of finding a propagation probability is to normalize the number of dependencies by defining m as

$$m(x) = \frac{x}{n(\text{Elements of } A) \times n(\text{Elements of } B)}. \quad (6)$$

This approach provides a very simple technique for obtaining conditional probability values from the relationships. On the other hand, the number of dependencies between two classes rarely gets even close to $n(\text{Elements of } A) \times n(\text{Elements of } B)$.

2) Element-wise Calculation: In this method, the focus moves to the *elements* of a class which means instead of calculating the probability of change of a class, the probabilities of change of the elements of that class are calculated. Then, the probability of change of a class is the probability of the union of changes in the elements of that class. Note that there is at most one dependency (i.e., call, access, and overriding) from one element to another in a software system. Thus, moving the focus from a class to its elements removes the problem with multiple dependencies.

Using this method, we keep the simple framework of assigning uniform propagation probabilities to each type of dependency. In order to represent the structure of an object-oriented system, we consider dependencies between elements of one class to be stronger than that between elements of different classes. For example, changes to a method would probably affect the dependent elements within the same class more than those in other classes. Thus, we assign a larger propagation probability value to a dependency between two elements of the same class, than the same type of dependency between elements of different classes.

Element-wise calculation increases the number of unknowns in the non-linear system of equations that needs to be solved by 20 to 30 times. This increase, in turn, results in more complexity.

© 2008 ACADEMY PUBLISHER

3) Merging Dependencies: Multiple dependencies between classes can be treated as independent conditional probabilities and thus can be merged by simply finding the union of those events. Therefore, for n dependencies between classes A and B, with conditional probabilities $\alpha_1, \alpha_2, \ldots, \alpha_n$, the equivalent probability of propagation is

$$\beta = 1 - (1 - \alpha_1)(1 - \alpha_2)\ldots(1 - \alpha_n). \qquad (7)$$

This method allows us to collapse multiple dependencies between classes into one and to compute the probabilities of change at the class level as opposed to the more fine grained analysis in element-wise calculation. While the computational complexity is reduced due to the smaller size of the system of equations, we lose some accuracy when we merge dependencies compared to using element-wise calculation.

Note that all the above methods rely solely on the dependencies that are obtained from the UML diagram of the system. The proposed methods use these dependencies in a systematic way to provide an estimate of propagation probabilities; hence, these methods do not require any human intervention and can be fully automated. This is a considerable improvement with respect to [21] where a single propagation probability was used to describe all dependencies; that value was obtained by manual inspection of all the change logs of the system.

C. Calculation of $\mathbb{P}_s(TC_i)$ and Cyclic Dependencies

As explained in Section III, we use (1) to calculate the values of $\mathbb{P}_s(\mathrm{TC}_i)$ for all classes. Assuming that that changes in different classes are independent, we can (1) into a nonlinear equation. Note however, that we cannot necessarily compute these values in a top-down fashion due to cyclic dependencies in the UML diagram.

Tsantalis *et al.* [21] use an approximation technique to get around this problem. They identify the cycles in the graph using a spanning tree [29], and temporarily remove edges from the graph until there are no cycles left. Then, the probabilities of change can be easily computed. After this stage, the removed edges are restored, one by one, and the probabilities of change of the nodes adjacent to those edges are updated. This method provides a close approximation to the true probabilities of change, but the results seemed to be biased; during few test runs, the estimated probabilities were always smaller than the true values. In order to get a better approximation, our approach considers three techniques. While all of them rely on solving a system of equations to get the probabilities, two techniques use simplifying assumptions to reduce the complexity of the calculations by making the system of equations linear or by making the dependency graph acyclic. The details of the three techniques are elaborated further as follows.

1) Nonlinear System of Equations: This approach uses (1) with no major simplification to calculate the probabilities of change. The only assumption associated with this approach is that changes in different classes are independent events. Assuming that events of change in different classes are independent, (1) can be negated and written as :

$$1-\mathbb{P}_s(\mathrm{TC}_i) = (1 - \mathbb{P}_s(\mathrm{IC}_i)) \times \prod_{j \neq i}(1-\alpha_{ij}\mathbb{P}_s(\mathrm{TC}_j)). \quad (8)$$

Using (8), a system of nonlinear equations can be constructed whose solution is the set of probabilities of change of the classes. Note that due to the nonlinearity of the system, it is more difficult to solve than linear systems. We use an implementation of Newton-Raphson Method [30] to solve a nonlinear system of equations in C++ given in [31].

2) Linear System of Equations: Similar to the previous method, this approach is based on (1). In this approach, however, it is assumed that changes in a class due to different sources are *mutually exclusive*. Based on this assumption, (1) can be written as

$$\mathbb{P}_s(\mathrm{TC}_i) = \mathbb{P}_s(\mathrm{IC}_i) + \sum_{j \neq i} \alpha_{ij}\mathbb{P}_s(\mathrm{TC}_j). \qquad (9)$$

Note that (9) is linear and is fairly easy to solve. Nevertheless, the simplifying assumption makes the solution to the linear system of equations an approximation to the solution of (8).

3) Depth First Search Graphs: A problem that is applicable to both of the previous methods is that they calculate probabilities for the steady state. For example consider a simple program with two classes with cyclic dependencies, as shown in Fig. 3.

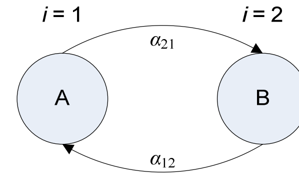

Figure 3. An example of circular dependencies.

Classes A and B indexed by $i = 1, 2$ have internal probabilities of change of $\mathbb{P}_s(\mathrm{IC}_1) = 0.1$ and $\mathbb{P}_s(\mathrm{IC}_2) = 0$, respectively. The probabilities of propagation are $\alpha_{12} = \alpha_{21} = 1.0$. Intuitively, we would expect to have $\mathbb{P}_s(\mathrm{TC}_1) = \mathbb{P}_s(\mathrm{TC}_2) = 0.1$, because the only cause of change in the system can come from an internal change in A, which has a probability of 0.1. However, the linear system of equations becomes degenerate and gives no answers, and the nonlinear system gives $\mathbb{P}_s(\mathrm{TC}_1) = 1.0$ and $\mathbb{P}_s(\mathrm{TC}_2) = 1.0$; these values are out of our range of 0.1. Therefore, both the linear and the nonlinear systems give incorrect probability values.

© 2008 ACADEMY PUBLISHER

```
GET-ALL-PROBABILITIES()
1  for each vertex v ∈ V(G)
2       do set visited[i] = WHITE   ∀i ∈ V(G)
3          prob[v] = GET-PROBABILITY(v)
4  return prob

GET-PROBABILITY(v)
 1  if visited[v] = BLACK
 2     then return p[v]
 3  visited[v] = RED
 4  p[v] = 1 − internal[v]
 5  for each vertex u ∈ V(G)
 6       do if visited[u] ≠ RED
 7             then p[v] = p[v] × (1 − dep[v, u]×
                          GET-PROBABILITY(u))
 8  visited[v] = BLACK
 9  p[v] = 1 − p[v]
10  return p[v]
```

Figure 4. The Depth First Search algorithm (DFS) is used for calculating probability values, where G is a directed dependency graph with $V(G)$ as its vertices (classes); $internal[v]$ denotes the probability of internal change of v; dep is a 2-dimensional array where $dep[v, u] = \alpha_{vu}$; $prob[v]$ is the probability of change of v, and $visited[v]$ denotes whether class v has or has not been visited (BLACK and WHITE respectively), or if it is being visited (RED).

This overestimation is due to the fact that we are implicitly taking into account the possibility that a change in A will affect it, through B, over and over. We counter this, by constructing a depth-first search subgraph when calculating the probabilities of change.

An informal description of the algorithm is as follows: consider a class A whose probability of change we are computing. We start constructing our tree by adding A as a node. Then, we add all classes which A depends on as children of A. We repeat this procedure until there are no additional classes to be added to the tree. Then, we calculate the probability of change of A using only the nodes and dependencies in the constructed subgraph, starting from the leaf nodes of the graph (those that do not depend on any other nodes). Calculation of probabilities should be straightforward as the constructed graph is acyclic by nature. A detailed description of the algorithm is given in Fig. 4; probabilities of change are obtained by calling the GET-ALL-PROBABILITIES procedure.

Using the DFS method, for each class i, we eliminate the cycles from the dependency graphs that are connected to i. This makes the portion of the graph that is relevant to finding the probability of change of i acyclic, and hence, the system of equations that gives the probability of change of i can be solved in a top-down fashion.

D. Axis of Time

As explained earlier, we need to normalize the raw probability values from the change history with respect to time. These raw probability values refer to the percentage of releases in which a class is changed. The values of Time Between Releases (TBR) and Time To Release (TTR) play critical roles in determining the probabilities of change based on history log. When time between consecutive releases is very short, an overestimation can be observed; the opposite is also true when this period is longer than the average. Note the proportionality between TBR and False Negative Ratio (FNR), and the inverse relationship between TBR and False Positive Ratio (FPR). In order to find $\mathbb{P}_h(\mathrm{TC}_i)$ in a unit time (e.g., one day), several methods were considered.

A key assumption associated with all of these methods is that internal changes made in two different time periods are independent of each other. The following sections elaborate further how our approach can deal with the axis of time.

1) Simple Conditional Estimation: This estimation method uses Bayes' law to find a mean probability of change

$$\overline{p} = \sum_{t=1}^{\infty} \mathcal{P}(\text{Change}|T = t)\mathbb{P}(T = t), \qquad (10)$$

where $\overline{p}$ denotes the average probability of change in unit time, and $\mathcal{P}(\text{Change}|T = t)$ indicates the probability of change in unit time, given that time between releases (TBR) is equal to t unit times. Using Bayes' theorem and (10) to calculate the probability of no-change gives

$$1 - \overline{p} = \sum_{t=1}^{\infty} \mathcal{P}(\text{No Change}|T = t)\mathbb{P}(T = t). \qquad (11)$$

If the probability of a class not changing in one unit time is $\mathcal{P}(\text{No Change}|T = t)$ and assuming that changes in different time periods are independent events, the probability of that class not changing in a period of length t is

$$\mathbb{P}(\text{No Change}|T = t) = \mathcal{P}(\text{No Change}|T = t)^t. \qquad (12)$$

Simplifying (11) using (12) gives

$$\overline{p} = 1 - \sum_{t=1}^{\infty} \sqrt[t]{\mathbb{P}(\text{No Change}|T = t)}\mathbb{P}(T = t). \qquad (13)$$

Using this method, the probabilities of change in unit time are fairly easy to compute. However, in order to get a good estimate we need at least two releases for each time interval (otherwise $\mathbb{P}(\text{No Change}|T = t)$ will be either 0 or 1); this means that quite a large number of samples are needed.

2) Logarithmic Estimation: This technique attempts to calculate the probability of no-change in unit time using the *observed* probability of no-change between consecutive releases. The observed probability is the probability of no-change in a given time interval and can be extracted from the change history of classes. The observed probability of no-change, q_o, for a class with a TBR of length t is

$$q_o(t) = q^t, \qquad (14)$$

where q is the probability of no-change in unit time and

© 2008 ACADEMY PUBLISHER

TABLE III.
PROPAGATION PROBABILITY RESULTS USING THE PIECEWISE PDF

	Simple Conditional Estimation		**Logarithmic Estimation**		**Root of Polynomial**	
p	p_{est}	std	p_{est}	std	p_{est}	std
0.1	0.1191	0.0656	0.1033	0.0464	0.1055	0.0485
0.2	0.2432	0.0972	0.2050	0.0612	0.2125	0.0655
0.4	0.5313	0.1628	0.3873	0.1078	0.4144	0.1174
0.8	0.9579	0.0702	0.8481	0.2019	0.8751	0.1700

is assumed to be constant for all periods. Assume that the TBRs have taken values of $t_1, t_2, \ldots, t_n$ over the past releases. Taking the logarithm of both sides of (14) and summing them over all periods gives

$$\sum_{i=1}^{n} \ln q_o(t_i) = \sum_{i=1}^{n} t_i \ln q, \tag{15}$$

where n is the number of periods. A simple averaging of (15) yields

$$\overline{\ln q_o} = \bar{t} \ln q. \tag{16}$$

Note that $\ln q_o$ cannot be efficiently computed when the number of past releases is small. In that case, $q_o(t_i)$ may be zero for some t_i. However, these values are non-zero when we have a large number of past releases, and if the system is stable enough, most of the values of $q_o(t_i)$ will be close to 1.

For large values of q_o, we have

$$\ln q_o \approx q_o - 1.$$

Thus, the following estimation can be used:

$$\overline{\ln q_o} \approx \ln \overline{q_o}. \tag{17}$$

Since $\overline{q_o} = 1 - \mathbb{P}(\text{Change})$, it can be easily computed from the given data. Thus, q is

$$q = \exp\left(\frac{\ln \overline{q_o}}{\bar{t}}\right). \tag{18}$$

This method involves very simple calculations and its complexity does not increase much as the periods get longer or the number of revisions increases.

3) Root of Polynomial: The average probability of no-change is

$$\mathbb{P}(\text{NC}) = \sum_{t=1}^{\infty} \mathbb{P}(\text{No Change}|T = t)\mathbb{P}(T = t). \tag{19}$$

Assuming that the probability of change in unit time is constant across all releases, the conditional probability on the right-hand side of (19) can be decomposed and written using the daily probability of change. This gives

$$\left(\sum_{t=1}^{\infty} \overline{q}^t \mathbb{P}(T = t)\right) - \mathbb{P}(\text{NC}) = 0, \tag{20}$$

where $\overline{q} = 1 - \overline{p}$. The values of $\mathbb{P}(\text{NC})$ and $\mathbb{P}(T = t)$ for all ts can be easily extracted from the given data, making (20) a polynomial whose degree is equal to the longest TBR; thus $\overline{p}$ can be obtained by finding the roots of (20). A possible problem with (20) is that it may seem not to have any real roots, or to have more than one real root. However, by Theorem 3.1, it has exactly one root between zero and one.

Theorem 3.1: Consider the function $g(x)$ defined by $g(x) = \sum_{i=1}^{n} \mu_i x_i - C$, where $0 \leq C \leq 1$, $0 \leq \mu_i \leq 1$ for all i and $\sum_{i=1}^{n} \mu_i = 1$. Then, $g(x)$ has exactly one non-negative real root and it lies between 0 and 1.

Proof: Since the polynomial $g(x)$ has non-negative coefficients, it is non-decreasing when $x \geq 0$. Thus, g has at most one real non-negative root. We note that substituting $x = 0$ yields $-C$ which is less than or equal to zero, and substituting $x = 1$ yields $1 - C$, which is non-negative. Therefore, by the Intermediate Value Theorem (IVT) [30], the polynomial has at least one root between 0 and 1. Thus, it follows from the above that $g(x)$ has exactly one non-negative root and that it lies between 0 and 1. ■

One drawback of this method is that it becomes very complex when the polynomial is of a high degree. We can bypass this issue by reducing the time resolution used to define a unit time (e.g., changing the unit time from 2 days to 8 days, makes the values of TBRs 4 times smaller, which results in a lower degree polynomial).

4) A Comparison: We set up a controlled experiment using generated data to compare the abovementioned techniques. We generate values for TBRs using two pdfs namely, an exponential distribution with $\mu = 50$, and a piecewise uniform distribution shown in Table IV. Changes in each unit of time occur according to a Bernoulli trial scheme [23] with a probability of p. If the total probability of change over a period is greater than 0.5, we designate it as a “change”; otherwise it is a “no change”. This data is then used by each of the above methods to estimate the value of p. We compare these estimates with the true values of p to determine the most accurate method.

TABLE IV.
THE PIECEWISE PROBABILITY DENSITY FUNCTION

x	1	2	3	4
$f(x)$	0.1	0.2	0.4	0.3

Each run of the experiment consisted of generated

© 2008 ACADEMY PUBLISHER

TABLE V.
PROPAGATION PROBABILITY RESULTS USING THE EXPONENTIAL PDF

	Simple Conditional Estimation		Logarithmic Estimation		Root of Polynomial	
p	p_{est}	std	p_{est}	std	p_{est}	std
0.1	0.4928	0.1446	0.0778	0.0267	0.1131	0.0560
0.2	0.6733	0.1414	0.1336	0.0968	0.2161	0.1189
0.4	0.8833	0.0945	0.3674	0.3406	0.5241	0.2722
0.8	0.9874	0.0317	0.8857	0.2742	0.9342	0.1616

TABLE VI.
REPEAT OF THE EARLIER EXPERIMENT, BUT WITH 200 PERIODS IN EACH RUN

	Simple Conditional Estimation		Logarithmic Estimation		Root of Polynomial	
p	p_{est}	std	p_{est}	std	p_{est}	std
0.3	0.6433	0.0537	0.1553	0.0120	0.2972	0.0304
0.7	0.9425	0.0238	0.2945	0.0778	0.7112	0.0754

information about change of a class in 15 periods of random lengths (according to the above pdfs). Results of 100 runs of the experiment are shown in Tables III and V. It is evident that Simple Conditional Estimation does not provide reliable results, as estimates are mostly far from the real value of p. While Logarithmic Estimation and Root of Polynomial provide similar estimates and standard deviations, only the estimate provided by the latter converges to the real value of p as the number of periods gets large (see Table VI). Thus, the Root of Polynomial method provides a more reliable estimate of the daily probability of change.

IV. EMPIRICAL EVALUATION

We now apply the proposed probabilistic approach on a medium-size system. First, the case study will be described, and then we present and discuss the results.

A. Case Study: JFlex

JFlex [3] is a Lexical Analyzer Generator for Java, written in Java, which takes a specially formatted specification file containing the details of a lexical analyzer as input and creates a Java file whose source code simulates the lexical analyzer. The source code for JFlex is publicly available, while the latest version that we examined consists of 58 Java classes; more detailed statistics regarding JFlex are presented in Fig. 5. Twelve subsequent versions have been analyzed using the proposed probabilistic approach to predict changes. JFlex has been selected for analysis for several reasons.

- JFlex is small enough that we can easily visualize and understand the relationships between different parts of software. This helps testing our framework during the development cycle, as inefficiencies within the model can easily be spotted.
- JFlex is large enough to qualify as an ordinary software tool. This characteristic provides a good understanding of how our approach will perform on larger software tools.
- The number of merges and splits of classes in the history of JFlex were quite low, which streamlined the development of our simulations.
- Tsantalis *et al.* [21] use JFlex in their work as a case study as well. Thus, we can easily benchmark our model relative to theirs, using the results obtained from simulations.

We use our model to predict changes in versions 1.2.2 to 1.4. In order to benchmark our model, we use the provided data for each release of JFlex, and predict changes in the succeeding release. We then compare our predictions against the actual changes from the change history and compute the Overall Accuracy, Sensitivity, FPR, and FNR corresponding to our predictions.

B. Evaluation Environment

Our procedure of evaluating our proposed methods involved several software tools and some small programs that we wrote in Java, C++ and MATLAB. We describe our environment here to show how our results can be reproduced for future research on a different software system.

Code metrics were extracted from each release of JFlex, using Borland Together, and exported in plain text format, with each line containing an element and its corresponding metric.

Dependencies between classes were extracted using Creole and exported in RSF format. This output was then parsed into an $n \times n$ matrix, where n is the number of classes or that of elements, depending on our level of granularity, and each element of the matrix contains the dependency between two classes/elements (α_{ij}). This matrix was then exported in plain text format, where each row of the output represents elements in a row of the obtained matrix.

Information regarding the actual changes to classes can be extracted by `diff`ing classes from consecutive releases

© 2008 ACADEMY PUBLISHER

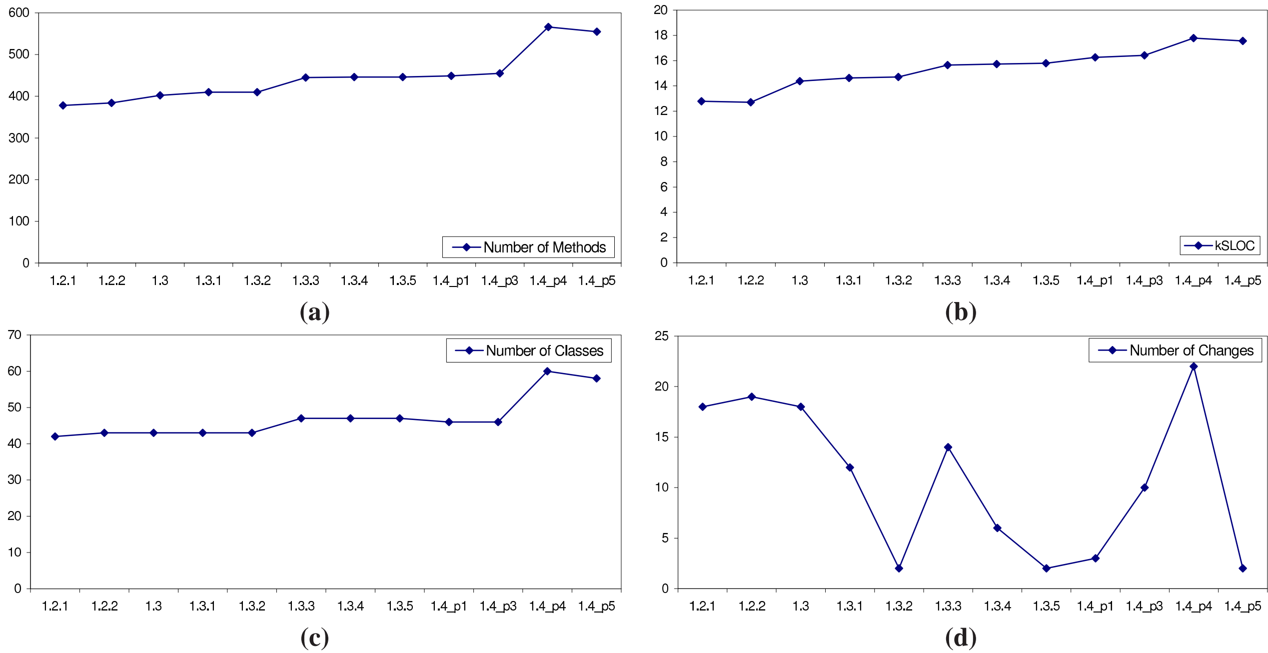

Figure 5. Some statistical information about JFlex: **(a)** Number of methods in consecutive releases of JFlex. **(b)** Growth of JFlex in terms of LOC. **(c)** Number of classes in 12 versions. **(d)** Number of changes to classes in different releases.

or by examining the logs of the code repository. Changes due to copyright or licensing updates should be ignored, as they do not have any effect on the functionality of the software tool. We did not use the above methods to extract this information about the changes, as this data was already available, courtesy of Tsantalis *et al.*.

The above information was passed to a MATLAB program, which estimated the probability of internal change, based on the given metrics, and then using the linear/nonlinear system of equations or the depth first search graph to determine the probability of change of classes. If the calculations were done at element level, the probability of change of each class is calculated as the union of those of its elements. The MATLAB program also calculates the time-normalized probability of change using the Root of Polynomial method. It then averages the metric-based and the history-based probabilities using predefined weightings. These probabilities were first rounded to 0 and 1, using a threshold of 0.5, and then compared against the actual changes in a release of JFlex. This process was repeated for all releases of JFlex.

We see that all the above steps are designed so that they can be performed without any human intervention. Thus, we can extend our methods to larger software system, given that the input data is supplied.

C. Calculation of $\mathbb{P}_s(TC_i)$ and Cyclic Dependencies

Nonlinear and linear systems equations, the depth first search graph (as presented in Section III-C), and the history change logs were used to compute the probabilities of change of classes in versions 1.2.2 to 1.4 of JFlex; we use information only from previous versions for this purpose. For example to compute probabilities for version 1.3, we use data only from releases 1.2.1 and 1.2.2. Note that in order to have an unbiased comparison, we have used the same probabilities of internal change and propagation probabilities that Tsantalis *et al.* applied. Results are presented in Table VII. Note that values related to *History* in Table VII are taken directly from [21], and therefore, no comments about the complexity of that method can be made. While *Linear System of Equations* (LSE) is the simplest technique, it provides the least overall accuracy for predicting changes. Some of the probability values calculated by this method are sometimes larger than unity. This is due to the assumption that propagated changes are mutually exclusive, which therefore, is not true.

The approximation method presented in [21], NLSE, and the depth first search method (DFS) seem to have the same level of prediction, but a closer look at the calculated probabilities reveals that there are indeed differences; these differences do not seem to have much effect on the overall accuracy, because probability values need to be rounded to 0 or 1 for predicting future changes. In terms of complexity, *Non-Linear System of Equations* (NLSE) and DFS seem complex in comparison with LSE, which is the simplest method. The complexity of NLSE also seems to grow faster than that of DFS with the size of the software system. Thus, DFS looks to be a better alternative when dealing with large software systems.

D. Estimation of $\mathbb{P}_s(IC)$ and Resolving Multiple Dependencies

As mentioned earlier, the probability values provided by normalization of dependencies using a simple mapping function were too small to be able to predict any change propagation. One solution considered to resolve this problem was magnifying the probability values. Although magnification improves the prediction of propagated changes,

© 2008 ACADEMY PUBLISHER

TABLE VII.
COMPARISON BETWEEN FIVE APPROACHES DEALING WITH CYCLIC DEPENDENCIES

Used Approach	Overall Accuracy	Sensitivity	FPR	FNR
Nonlinear System of Equations (NLSE)	0.6875	0.4786	0.1036	0.5214
Linear System of Equations (LSE)	0.6765	0.4701	0.1171	0.5299
Depth First Search Graphs (DFS)	0.6844	0.4701	0.1014	0.5299
Binary Dependencies	0.6844	0.4701	0.1014	0.5299
History [21]	0.6576	0.4188	0.1036	0.5812

it requires some manual searching to identify the best magnifying factor so that the magnified values are large enough to predict propagations, but not so large that they exceed their maximum limit. As this search needs to be done for each case study, it would reduce the automation and increase the complexity of the model. Therefore, while this approach provided some improvements over binary treatment of the dependencies, it was deemed not very suitable for calculation of conditional probabilities.

We examined many linear combinations of metrics for the value of x for estimating $\mathbb{P}_s(\text{IC})$. Our results indicate that $x = LOC$ yields the best results in terms of overall accuracy. This reflects one of the conclusions of [21], where LOC was found to be a very capable indicator of the probability of change of a class. We ran several simulations to show that dependencies correlate with the actual propagated changes. From simulation results, it was concluded that *call* and *access* relationships correlate the best with the actual changes. Therefore, only these relations were taken into account to estimate the conditional probabilities. This resulted in a modification to (6) since dependencies were from methods to elements only.

After several tests to find the best suiting values for *Element-wise Calculation*, conditional probability values of 0.9 and 0.4 were assigned to access and call dependencies between elements in the same class. The conditional probability values for inter-class dependencies were 60% of those for intra-class relationships. Note that a higher probability value was associated with access, as any change in an attribute will most probably propagate to its users (e.g., change of type or name of the attribute). The lower values associated with call is due to the fact that sometimes changes in a body of a method do not propagate to users of that method.

We followed a similar procedure to find best suiting values for conditional dependencies corresponding to call and access for class-level calculations using the depth first search graph algorithm and merged dependencies. We found that values of 0.1 and 0.05 corresponding to call and access provide fairly accurate results. Note, that the considerable difference between these values and those used for Element-wise Calculation are due to the difference in the probability of change of classes and their elements.

For a final comparison, we used *Element-wise Calculation* with the above parameters for computing the probabilities of internal change for each release of JFlex between 1.2.2 and 1.4 (12 releases in total). This method results in no need for resolving multiple dependencies, as there exists at most one relationship between any pair of elements. The non-linear system of equations was used to find the values of $\mathbb{P}_s(\text{TC}_i)$. The resulting values of $\mathbb{P}_s(\text{TC}_i)$ were averaged with the corresponding $\mathbb{P}_h(\text{TC}_i)$ values calculated by using the *Root of Polynomial Method*. These average values were rounded to 0 or 1 using a cut-off value of 0.5, with 1 predicting a change and 0 predicting no change. These predictions were then compared with the actual changes and the Overall Accuracy, Sensitivity, FPR, and FNR that were calculated by our approach. The results, shown in Table VIII, indicate a 3.5% improvement over [21] and 6.2% improvement over the use of history change logs.

We computed values of $\mathbb{P}_s(\text{TC}_i)$ again, but using the depth first search algorithm in conjunction with *Merging Dependencies* to resolve multiple dependencies. The history-based probabilities were calculated by the above-mentioned method and the same weighting coefficients were used to combine $\mathbb{P}_s(\text{TC}_i)$ and $\mathbb{P}_h(\text{TC}_i)$. Results are similar to those from *Element-wise Calculation* with 3% improvement over [21] and just shy of 5.8% improvement over use of history change logs.

E. Lessons Learned

Our experimental studies show that the solution to the set of nonlinear probability equations and our depth first graph based method yield more accurate values for the probability of change of the classes, compared with other methods. The use of nonlinear system of equations eliminates the approximation errors when calculating the steady state probabilities (this is a problem with the linear system of equations, as the probability values may exceed unity). The depth first search method does not calculate the steady state probabilities, and assumes that a class cannot cause a change to itself through the external axis. We deem DFS to be theoretically more valid (see Section III-C.3 for an example), but NLSE and DFS yield similar results, so we consider both to be practical approaches.

An elementary analysis of the frequency of change of classes in JFlex reveals that the time between releases is an important factor to determine the probability of changes in code. We found that the Root of Polynomial method best incorporates time into our calculations. This is due to the fact that the Root of Polynomial approach provides a better approximation of time-normalized probabilities compared with Simple Conditional Estimation and Logarithmic Estimation. The complexity of the Root of Polynomial method

© 2008 ACADEMY PUBLISHER

TABLE VIII.
COMPARISON BETWEEN FOUR APPROACHES FOR PREDICTING CHANGES

Used Approach	Overall Accuracy	Sensitivity	FPR	FNR
NLSE + Element-wise Calculation	0.7197	0.5385	0.0991	0.4615
DFS + Merged Dependencies	0.7154	0.5299	0.0991	0.4701
Binary Dependencies	0.6844	0.4701	0.1014	0.5299
History [21]	0.6576	0.4188	0.1036	0.5812

can also be easily reduced by increasing the length of the defined time unit.

Propagation probabilities are more accurately calculated based on the relationships between the elements of classes. This change incorporates the dependencies in the UML diagram more appropriately than considering relationships between classes as a binary effect. However, there is an associated increase in complexity due to the larger size of the system of equations, when we calculate probabilities at method and parameter level. We can alternatively calculate probabilities at the class level by merging multiple dependencies between classes. While this method has a lower complexity compared with element-wise calculations, it has a lower accuracy because of the difference in the level of granularity.

In order to increase the automation of the model, the method used in [21] to extract the internal-change probabilities needs to be changed. We found that LOC was the best indicator of internal changes, as it correlated best with internal changes in JFlex. A more thorough suite of metrics may be used for larger case studies. Since use of LOC only provides structural data about the software system, it should not be used as the only source of predicting changes (e.g., consider a small method that is modified often). Thus, the total probability of change of a class is calculated as a weighted average of probabilities extracted from the source code (i.e., based on code metrics and dependencies from the UML diagram) and those based on the time-normalized change history.

V. CONCLUSIONS

This paper proposes a probabilistic approach to predict changes in object-oriented systems. The proposed approach uses the axis of time to define and guide the prediction process.

We believe that this approach is noteworthy for two main reasons. First, it attempts to address a problem that has challenged the research community for several years, namely the maintenance of object-oriented mission critical systems. Second, it aims to devise a workbench in which the changes to the source code do not occur in a vacuum, but can be evaluated and fine-tuned in order to address specific quality requirements for the new target system such as enhancements in maintainability.

We plan to apply the developed model on various other software systems in a larger scale to ensure the extensibility of the proposed approach.

VI. FUTURE WORK

We made several simplifying assumptions regarding the independency of events while describing our approach (e.g., independency of the changes in different time intervals). These assumption did not seem to have any negative effect on the accuracy of our method, compared to other suggested methods, but relaxing those assumptions may improve our prediction. For example, it can be assumed that the change history of classes is not memoryless (i.e., changes are dependent). A Markov model can be used to take into account such effect.

We used several parameters in our calculations, whose values were determined empirically (e.g., parameter λ in (5) and the weights assigned to different dependencies). In future work, these parameters should be directly determined from the source code, the application domain, or any other related data.

Several possible solutions were discarded due to the lack of resources (e.g., CVS change logs for JFlex). These solutions can be reconsidered when new case studies are analyzed.

ACKNOWLEDGEMENT

The authors gratefully acknowledge the contributions from Professor Amir K. Khandani for useful discussions and for his comments on the first draft of this work.

REFERENCES

[1] D. Parnas, “Some software engineering principles,” *Structured Analysis and Design*, pp. 237–247, 1978.

[2] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[3] “JFlex – The Fast Scanner Generator for Java,” 2007, http://www.jflex.de/.

[4] T. Zimmermann, A. Zeller, P. Weigerber, and S. Diehl, “Mining Version Histories to Guide Software Changes,” *IEEE Trans. on Soft. Eng.*, vol. 31, no. 6, pp. 429–445, 2005.

[5] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, “Predicting Source Code Changes by Mining Change History,” *IEEE Trans. on Soft. Eng.*, vol. 30, no. 9, pp. 574–586, 2004.

[6] R. Arnold and S. Bohner, “Impact Analysis - Toward a Framework for Comparison,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 1993, pp. 292–301.

[7] M. Chaumun, H. Kabaili, R. Keller, and F. Lustman, “Change Impact Model for Changeability Assessment in Object-Oriented Software Systems,” *Science of Computer Programming*, vol. 45, no. 2–3, pp. 155–174, 2002.

© 2008 ACADEMY PUBLISHER

[8] L. Tahvildari and K. Kontogiannis, “Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach,” *Journal of Software Maintenance and Evolution: Research and Practice (JSME)*, vol. 16, no. 4–5, pp. 331–361, 2004.
[9] L. Li and A. Offutt, “Algorithmic Analysis of the Impact of Changes to Object-Oriented Software,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 1996, pp. 171–184.
[10] L. Briand, J. Wust, and L. Lounis, “Using Coupling Measurement for Impact Analysis in Object-Oriented Systems,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 1999, pp. 475–482.
[11] A. E. Hassan and R. C. Holt, “Predicting Change Propagation in Software Systems,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2004, pp. 284–293.
[12] “Version Management with CVS for CVS 1.11.21,” 2005, free Software Foundation Inc., http://ximbiot.com/cvs/manual/.
[13] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller, “Predicting faults from cached history,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007, pp. 489–498.
[14] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Trans. on Soft. Eng.*, vol. 26, no. 7, pp. 653–661, 2000.
[15] A. Mockus and D. M. Weiss, “Predicting risk of software changes,” *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169 – 180, April 2000.
[16] H. Kagdi and J. I. Maletic, “Software-change prediction: Estimated+actual,” in *Second International IEEE Workshop on Software Evolvability (SE)*, 2006, pp. 38–43.
[17] V. Basili, L. Briand, and W. Melo, “A Validation of Object-Oriented Design Metrics as Quality Indicators,” *IEEE Trans. on Soft. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.
[18] S. Chidamber and C. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Trans. on Soft. Eng.*, vol. 20, no. 7, pp. 476–493, 1994.
[19] T. Girba, S. Ducasse, and M. Lanza, “Yesterdays Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2004, pp. 284–293.
[20] E. Arisholm, L. Briand, and A. Foyen, “Dynamic Coupling Measurement for Object-Oriented Software,” *IEEE Trans. on Soft. Eng.*, vol. 30, no. 8, pp. 491–506, 2004.
[21] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, “Predicting the Probability of Change in Object-Oriented Systems,” *IEEE Trans. on Soft. Eng.*, vol. 31, no. 7, pp. 601–614, 2005.
[22] F. Xia, “A Change Impact Dependency Measure for Predicting the Maintainability of Source Code,” in *Proceedings of the Annual International Computer Software and Applications Conference (COMPSAC)*, vol. 2, 2004, pp. 23–24.
[23] S. Ross, *A First Course in Probability*. Macmillan College Pusblishing, Inc., 1994.
[24] T. J. McCabe, “A complexity measure,” *IEEE Trans. on Soft. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.
[25] W. Li and S. Henry, “Object-Oriented Metrics that Predict Maintainability,” *Journal of Systems and Software*, vol. 23, no. 2, pp. 111 – 122, November 1993.
[26] “Software Architecture Design, Visual UML & Business Process Modeling - from Borland,” 2007, http://www.borland.com/us/products/together/index.html.
[27] “Creole – The CHISEL Group,” 2007, http://www.thechiselgroup.org/creole/.
[28] “Rigi Group Home Page,” 2007, http://www.rigi.csc.uvic.ca/.
[29] B. Bollobas, *Graph Theory: An Introduction*. Springer-Verlag New York, 1979.
[30] G. James, D. Burley, D. Clements, P. Dyke, J. Searl, and J. Wright, *Modern Engineering Mathematics*, 3rd ed. Pearson Education Ltd, 2001.
[31] W. H. Press, *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992.

Ali R. Sharafat is a third year undergraduate student (expected graduation in 2009) at the Faculty of Mathematics majoring in Computer Science and Combinatorics & Optimization at the University of Waterloo in Canada.

He has completed internship work terms at Research In Motion a DSP Firmware Developer and the University of Waterloo as a Research Assistant. His research interests include software evolution and testing.

Mr. Sharafat was a Member of the Canadian National Team in 2004 International Physics Olympiad, is the recipient of Natural Sciences and Engineering Research Council of Canada (NSERC) Fellowship, was awarded the Fessenden-Trott Scholarship, and has consistently ranked as the top student in his class at the University of Waterloo. He is a student member of IEEE.

Dr. Ladan Tahvildari is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Waterloo, a Visiting Scientist with Centre for Advanced Studies at the IBM Toronto Laboratory, and the founder of the Software Technologies Applied Research Laboratory. She received her BASc from Iran University of Science and Technology, and her MASc and PhD from University of Waterloo in Software Engineering. She has established the area of Quality-Driven Object-Oriented Re-engineering which is a novel approach for improving maintainability and performance of object-oriented legacy systems. Her research has appeared in over 50 peer-reviewed publications. Dr. Tahvildari has been on the program and organization committees of many international IEEE/ACM conferences. She is Program Co-Chair of IEEE ICSM2007 in Paris, Working Sessions and Tools Chair of IEEE ICPC2006 in Greece, Program Co-Chair of IEEE STEP2004 in Chicago, and Workshops Chair of IEEE WCRE2004 in the Netherlands. She has served as Chair of the Computer Society (CS) in the IEEE Local Chapter since 2004. Her accomplishments have been recognized by various awards. Recently she has been honored with the prestigious Ontario's Early Researcher Award (ERA) to recognize her work in self-adaptive software.

© 2008 ACADEMY PUBLISHER