Research on Cross-version Software Defect Prediction Based on Evolutionary Information

Zhexi Yao1*, Jiawei Song1, Yushuai Liu1, Tao Zhang2 and Jinbo Wang2

1University of Chinese Academy of Sciences, Beijing
2Technology and Engineering Center for Space Utilization, Chinese Academy of Sciences, Beijing, China
*yaozhq14@csu.ac.cn

Abstract. Software defect prediction is an important part of the software testing field. According to the characteristics of object-oriented software, this paper considers the evolution information separately in different packages and summarizes the evolution metrics that affect the defect prediction. Existing research on evolutionary information often ignores the impact of newly added and disappearing classes on software defects prediction. Based on these factors, evolution metrics are proposed and applied to defect prediction. Two evolution metrics, transition class ratio and static metric category number, are proposed for object-oriented cross-version defect prediction. Experiments are carried out in the commonly used software defect prediction set. The experimental results show that the proposed metrics have better defect prediction ability than the traditional static metric.

1. Introduction
Software defect prediction (SDP) is an important task in software testing and is an important activity to ensure software quality. SDP is mainly based on historical data to predict potential defects in software, specifically statistical analysis of historical data. The researchers have invested a lot of time and energy into this field and have achieved certain results. The results of defect prediction provide a more accurate evaluation basis for testers, which is conducive to the rational allocation of testing resources and the improvement of testing efficiency. Meanwhile, more and more researchers are engaged in this field. The application of a large number of cross-domain theories greatly promotes the development of SDP.

Defect prediction based on evolutionary information has been widely studied. The basic idea is to simulate the evolution process of software as a biological evolution process, find the relationship between software in different versions, and finally apply the relationship to SDP. The metrics based on evolutionary information are proved to have better prediction effect than the static metrics [1]. Based on the evolution information of object-oriented software, the evolution metrics are proposed. Experiments show that the proposed metric has excellent prediction effect.

2. Related works
The SDP plays an important role in software testing. In software engineering, the introduction of defects is inevitable, which may be caused in every software engineering stage from requirement analysis to software development. If a defect is detected later, the cost of fixing it will be high. Especially after the software is released, the cost of detecting and fixing defects increases dramatically.
By predicting the distribution of defects in software development, testing resources can be reasonably allocated and costs can be effectively reduced.

2.1. Cross-version software defect prediction

Cross-version defect prediction is the process of performing defect prediction between different versions of the software. Different versions of the software are related to each other. Software improves itself by changing versions, this process called software evolution. Software evolution is an important activity in the entire software life cycle. It is a dynamic and continuous process and is a general term for a series of maintenance activities after the software is delivered [2]. For example, after the software is put into use, maintenance personnel need to change the software to meet new requirements due to defect repair, increased demand, and environmental changes, or to improve the performance and reliability of the software. In software evolution, maintenance personnel may generate new defects when repairing historical defects, and may also generate new defects in added parts. Therefore, software defects are closely related to software evolution, and the defects between different versions are also correlated.

Software evolution is a long-term and continuous process, and its evolution information can reflect the changing characteristics of software as well as the changes of defects. Adding evolutionary information to SDP can improve the performance of the model. Some researchers use correlation analysis or visualization to study the relationship between various features and defects of software changes. Sliwerski [3] quantitatively analysed transactional-level changes in open source projects, pointing out that the size and time of changes are associated with the risk of introducing defects. Another part of the research is devoted to building defect prediction methods for software source code changes. Aversano [4] realized defect prediction of transaction-level changes based on the idea of text classification.

For object-oriented programs such as Java, the situation is more complicated. In the evolution of object-oriented programs, classes in the package change with versions. According to the changes in the classes, they are divided into three categories: common class, new class, and disappearing class. They are products of software evolution. In practical, the distribution of packages in object-oriented software is related to the different functions of the software. While the classes in a package vary in evolution, the functionality represented by a package tends not to change. In software evolution, the change of code in a package is often reflected in the change of different classes. Based on this characteristic, it is feasible to consider the defect prediction work separately in different packages.

2.2. Software metric

The software metric is an indicator or parameter that describes the characteristics of a software product, also known as a feature. In SDP, reasonable metrics are needed. It can be classified into static metrics and evolutionary metrics according to its production process.

Existing static metrics mainly include process-oriented McCabe metric [5], Halstead [6] metric and object-oriented CK metric [7], which reflect the static characteristics of software. The CK measure comprehensively considers the characteristics of inheritance, coupling, and cohesion in object-oriented programs, and effectively measures the correlation between program characteristics and software defects. At present, the above metrics are widely used in SDP and have achieved good prediction results. The commonly used 20 code metrics are given in Table 1, which have been used in previous studies [8, 9]. These code metrics can be divided into five categories based on the aspect of the metric: complexity, coupling, cohesion, inheritance, and scale. It can be seen that the metrics in the same category are related to each other. When one metric increases, the other metrics in the same category tend to increase.

In practical applications, evolutionary information is also very important for SDP. Aiming at the characteristics of the software evolution process, some evolution metric, such as static metric change [10,11] or version information [12], are also proposed. [12] proposes that the change measures of static metric can be used as evolutionary measures, but the relationship between the measures is
ignored. It can be seen that different static metrics in the same category are related. It may be inefficient to study only the number of static metrics in transition. This paper separately analyzes the evolution characteristics of software from the perspective of adding new classes and disappearing classes in each package. At the same time, the changes of static metrics in different category are designed as the evolution metric.

Table 1. The category of static metric

| categories       | metrics                        | symbols | description                                                                 |
|------------------|--------------------------------|---------|-----------------------------------------------------------------------------|
| complexity       | Average Method Complexity      | amc     | E.g., number of JAVA byte codes                                            |
|                  | Average McCabe                 | avg_cc  | Average McCabe’s Cyclomatic Complexity values of methods in the same class |
|                  | Maximum McCabe                 | max_cc  | Maximum McCabe’s Cyclomatic Complexity values of methods in the same class |
| coupling         | Afferent couplings             | ca      | How many other classes use the specific class                              |
|                  | Coupling Between Methods       | cbm     | Total number of new/redefined methods to which all the inherited methods are coupled |
|                  | Coupling Between Object classes| cbo     | Increased when the methods of one class access services of another         |
|                  | Efferent couplings             | ce      | How many other classes is used by the specific class                       |
|                  | Inheritance Coupling           | ic      | Number of parent classes to which a given class is coupled                 |
| cohesion         | Cohesion among Methods of class| cam     | Summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods |
|                  | Lack of Cohesion in Methods    | lcm     | Number of pairs of methods that do not share a reference to an instance variable |
|                  | Lack of Cohesion in Methods, different from LCOM | lcm3 | If m, a are the number of methods, attributes in a class number and \( \mu(a) \) is the number of methods accessing an attribute, then \\
|                  |                                 |         | \( lcom3 = \left(1 - \frac{\sum_{j=1}^{a} \mu(a_j) - m}{1 - m}\right) \) |
| inheritance      | Depth of Inheritance Tree      | dit     | Provides the position of the class in the inheritance tree                 |
|                  | Measure of Aggregation         | moa     | Count of the number of data declarations (class fields) whose types are user defined classes |
|                  | Measure of Function Abstraction| mfa     | Number of methods inherited by a class plus number of methods accessible by member methods of the class |
| scale            | Lines of Code                  | loc     | Measures the volume of code                                               |
|                  | Number of Children             | noc     | Measures the number of immediate descendants of the class                 |
|                  | Response for a Class           | rfc     | Number of methods invoked in response to a message to the object           |
|                  | Number of Public Methods       | npm     | Counts all the methods in a class that are declared as public             |
|                  | Weighted Methods per Class     | wmc     | The number of methods in the class (assuming unity weights for all methods) |
|                  | Data Access Metric             | dam     | Ratio of the number of private (protected) attributes to the total number of attributes |
3. Evolution defect prediction

3.1. Evolutionary metric

Existing researches mainly focus on the classes remained, while there is little research on the newly added classes and disappearing classes. But the changes in the package are the most direct by the newly added and disappearing classes.

(1) Transition class proportion

For two adjacent versions of the software, defects in the previous version Vt-1 may be fixed or may continue to exist in current version Vt. Commonly, the changes of classes in the same package may aim to add functionality or to reduce defects. Therefore, the reduced class ratio between Vt-1 and Vt is used as the evolution metric of Vt-1. Similarly, for the metrics added in Vt, there is a certain possibility that it is to add new functions. The increased proportion is used to characterize the evolution information in Vt and participated in the defect prediction. Then the first evolution metric is proposed. The calculation formula is shown in (1) and (2) below.

\[ Pac = \frac{\text{The number of added classes in the package}}{\text{The total number of classes in the package}} \]  

(1)

\[ Pdc = \frac{\text{The number of classes disappeared in the package}}{\text{The total number of methods in the package}} \]  

(2)

(2) Static metric category number.

Static metrics reflect the static nature of the software, and each evolution version can be represented by certain static metrics. For the same class in two adjacent versions of Vt-1 and Vt, the comparison shows how the various code metrics have changed. In an evolutionary cycle, a class is considered to have changed significantly if it changed in most of the static metrics. We believe that the more evolved a class is, the greater its correlation with defects will be. At the same time, the evolutionary metrics based on the category of static metrics are proposed to represent the defects in these two versions of the software. The second evolution metric is proposed: based on the category of code metrics in Table 1, the changes of static metric between two adjacent versions are considered, and the number of categories is summarized as the new evolution metric. The calculation is shown in formula (3).

\[ Scc = \sum \text{categories of changed static metric} \]  

(3)

3.2. Cross-release defect prediction based on evolution metric

This paper studies the prediction of defects across versions, and the overall structure is shown in figure 1. As is shown in Figure 1, the evolution process of software is simulated by the change of defect data sets between adjacent versions, and the evolution metric is finally proposed to provide help for defect prediction. The entire prediction process is as follows. First, for two adjacent versions of a project Vt-1, and Vt, the classes in these two versions may not be exactly the same. Then Vt-1 and Vt need to be pre-processed to get their common classes, and the data related to evolution metrics can be extracted. The Vt-1 version of the common class and the Vt version of the common class are represented in Figure 1. The Pac and Scc are extracted from the adjacent version of the software as the newly added evolution metric of the pre-processed Vt-1; the Pdc and Scc are used as the evolution metric of the pre-processed Vt version. Finally, defect prediction is carried out for the pre-processed Vt-1 and Vt respectively.
4. Experiments

The defect data set on object-oriented software is selected from the commonly used database in SDP. Experiments are performed to verify the effectiveness of the evolutionary metrics proposed. The experiments are carried out under Weka 3.7 and eclipse respectively.

4.1. The data set

The experiment selected 36 versions of 10 open source projects, all of which are developed by the Java language. And they are also the data set commonly used in defect prediction [12, 13]. The basic information of each data set is shown in Table 2, which contains the basic information of the project, including project name, version number, number of classes and defect rate.

| Project  | Version | Number of classes | Defective classes(%) | Project  | Version | Number of classes | Defective classes(%) |
|----------|---------|-------------------|----------------------|---------|---------|-------------------|----------------------|
| Ant      | 1.3     | 125               | 16                   | Jedit   | 3.2     | 272               | 33.09                |
|          | 1.4     | 178               | 22.47                |         | 4.0     | 306               | 24.51                |
|          | 1.5     | 293               | 10.92                |         | 4.2     | 367               | 13.08                |
|          | 1.6     | 351               | 26.21                |         | 4.3     | 492               | 2.23                 |
|          | 1.7     | 745               | 22.28                |         | 1.0     | 135               | 25.18                |
| Camel    | 1.0     | 339               | 3.83                 | Log-4j  | 1.1     | 109               | 33.94                |
|          | 1.2     | 608               | 35.53                |         | 1.2     | 205               | 92.19                |
|          | 1.4     | 872               | 16.63                | Lucene  | 2.0     | 195               | 46.67                |
|          | 1.6     | 965               | 19.48                |         | 2.2     | 247               | 58.3                 |
| Xerces   | 1.2     | 440               | 16.14                |         | 2.4     | 340               | 59.7                 |
|          | 1.3     | 453               | 15.23                | Poi     | 1.5     | 237               | 59.49                |
|          | 1.4     | 588               | 74.32                |         | 2.0     | 314               | 11.78                |
| Synapse  | 1.0     | 157               | 10.19                |         | 2.5     | 385               | 64.41                |
|          | 1.1     | 222               | 27.03                |         | 3.0     | 442               | 63.57                |
|          | 1.2     | 256               | 33.59                | Xalan   | 2.4     | 723               | 15.21                |
| Velocity | 1.4     | 196               | 75                   |         | 2.5     | 803               | 48.19                |
|          | 1.5     | 214               | 66.35                |         | 2.6     | 885               | 46.44                |
|          | 1.6     | 229               | 34.06                |         | 2.7     | 909               | 98.79                |

4.2. The experiment design

A feature selection experiment is performed to verify the effectiveness of the algorithm in this paper. The evaluation methods is based on the CFS (the feature selection method based on correlation) algorithm. It is designed to compare the evolutionary metrics and static metrics proposed in this paper. The CFS method can evaluate the independent prediction ability of each feature and the redundancy between each feature and other features, and select the optimal feature subset with the combination of the optimal priority search algorithm. The BestFirst search method is used in the experiment. The

![Figure 1. Cross-release defect prediction based on evolution metric](image-url)
attribute selection results are shown in Table 3. The statistical analyse results of Table 3 are listed in Table 4.

### Table 3. The result of selected features

| Project | selected features                          | Project | selected features                          |
|---------|-------------------------------------------|---------|-------------------------------------------|
| ant     | 1.3:moa mfa Pdc Sec ; 1.4:rfc dam cam Sec | Poi     | 1.5:dit noc lc0m3 loc moa mfa ic cbm max_cc avg_cc; 2.0:rfc Sec |
|         | 1.4:dit rfc ce dam ic cbm Pdc; 1.5:ce lc0m3 mfa Pac |         | 2.0:rfc lc0m3 ic cbm; 2.5:dit ebo rfc lc0m ce lc0m3 loc mfa ic cbm max_cc avg_cc Sec |
|         | 1.5:rfc mfa cbm Pdc; 1.6:dit rfc moa mfa ic Pac Sec |         | dit rfc Pdc Sec; rfc moa cam avg_cc Sec |
|         | 1.6:rfc mfa max_cc Pdc; 1.7:rfc mfa |         | Lucene 2.0:wmc moa ic Pdc; 2.2:dit rfc Pac Sec |
| Camel   | 1.0:dit noc cbo loc moa mfa avg_ce Pac Sec; 1.2:noc cbo moa mfa cam cbm avg_ce Sec |         | 2.2:cb0 rfc ce npm ic cbm; 2.4:wmc moa ic Pac |
|         | 1.2:wmc noc ca ce npm moa ic avg_ce; 1.4:noc cbo lc0m ca ce loc moa ic cbm ame avg_ce Pac Sec |         | 2.0:wmc moa ic Pac |
|         | 1.4:wmc noc ebo moa max_cc avg_ce Sec; 1.6:wmc noc ca ic Sec | Jedit   | 3.2:dit noc rfc lc0m ca ce npm lc0m3 dam moa cam ic cbm ame max_ce avg_ce Pdc Sec; 4.0:dit noc rfc lc0m ca ce npm lc0m3 dam moa cam ic cbm ame max_ce avg_ce Pac Sec |
| log4j   | 1.0:wmc cbo ce moa Pdc Sec; 1.1:wmc npm Sec |         | 4.0:wmc npm dam moa ic cbm avg_ce Pdc; 4.2:dit noc rfc lc0m ca ce npm lc0m3 dam moa cam ic cbm ame max_ce avg_ce Sec |
|         | 1.1:dit npm dam mfa ic cbm max_cc; 1.2:wmc cbo ce moa Pac |         | 4.2:dit ce moa mfa Pdc; 4.3:wmc ebo rfc ca ce npm moa max_ce avg_ce |
|         | 1.2:wmc cbo ce moa Pac | xalan   | 2.4:rfc Pac; 2.5:moa cbm Sec |
| velocity| 1.4:dam moa cam ame avg_ce Sec; 1.5:lc0m3 dam Pac Sec |         | 2.5:wmc noc cbo rfc lc0m ca ce lc0m3 loc dam moa cam ic cbm max_ce avg_ce Sec ; 2.6:rfc Pac Sec |
|         | 1.5:dit noc cbo rfc lc0m ce npm lc0m3 dam moa cam ic cbm ame Sec ; 1.6:lc0m3 dam moa cam avg_ce Pac |         | 2.6:wmc dit noc rfc ca ce npm lc0m3 loc dam moa mfa cam ic cbm max_ce avg_ce Pdc Sec; 2.7:loc moa cam Pac Sec |
| synapse | 1.0:dit ce moa mfa max_ce; 1.1:ce mfa cbm avg_ce | Xerc3es | 1.2:ce Pdc; 1.3:dit mfa |
|         | 1.1:noc cbo rfc ce Sec; 1.2:dit ce moa mfa Pac Sec |         | 1.3:dit cbo rfc ce mfa ic cbm max_ce Pdc Sec; 1.4:ce ic Sec |

### Table 4. The statistical results

| ScC | moa | Pdc+Pac | ic | ce | cbm | avg_ce | cbo | mfa | dit | rfc |
|-----|-----|---------|----|----|-----|--------|-----|-----|-----|-----|
| 30  | 28  | 15+12   | 21 | 20 | 19  | 18     | 18  | 18  | 18  | 15  |
| dam | noc | max_ce  | cam| lecom3| npm | wmc | loc | lecom| ame |
| 14  | 14  | 13      | 12 | 12 | 11  | 11     | 7   | 7   | 6   |
It can be seen from Table 3 that the metrics proposed have a large number of occurrences, which means the metrics have a higher correlation with the defect. In Table 4, the Scc and Pdc+Pac own the first and third places, which show that they have greater predictive power than the static metric.

5. Conclusions
This paper proposes a cross-version defect prediction method based on software evolution history. The evolution of software is considered as the process of species evolution. This paper mainly uses the evolutionary information of the newly added and disappearing classes, as well as the category change of the static metrics to predict the defects of the existing classes. On the premise of summarizing evolution connotation, this paper proposes two kinds of evolution metrics. The experimental results verify the validity of the software evolution metrics proposed in this paper.

References
[1] Nagappan N and Ball T 2005 Use of relative code churn measures to predict system defect density. Proc. The 27th Int’l Conf. on Software Engineering. pp 284-292.
[2] Chidamber S R and Kemerer C F 1994 A metrics suite for object oriented design. IEEE Transactions on Software Engineering. pp 476-493.
[3] Sliwerski J, Zimmermann T and Zeller A 2005 When do changes induce fixes? ACM Sigsoft Software Engineering Notes. pp 1-5.
[4] Aversano L, Cerulo L and Grosso C D 2007 Learning from bug-introducing changes to prevent fault prone code. Proc of the Int’l Workshop on Principles of Software Evolution. pp 19-26.
[5] McCabe T J 1976 A complexity measure. IEEE Transactions on Software Engineering. pp 308-320.
[6] Halstead M H 1977 Elements of software science (New York, USA: Elsevier)
[7] Chidamber S R and Kemerer C F 1994 A metrics suite for object oriented design. IEEE Transactions on Software Engineering. pp 476-493.
[8] Jureczko M and Madeyski L 2010 Towards identifying software project clusters with regard to defect prediction. Proc of the 6th Int’l Conf. on Predictive Models in Software Engineering. pp 9-19.
[9] He Z, Peters F, Menzies T and Yang Y 2013 Learning from open-source projects: An empirical study on defect prediction. Proc of the ACM/IEEE Int’l Symp on Empirical Software Engineering and Measurement. pp. 45-54.
[10] Nagappan N and Ball T 2005 Use of relative code churn measures to predict system defect density. Proceedings of the 27th International Conference on Software Engineering pp. 284-292.
[11] Hassan A E 2009 Predicting faults using the complexity of code changes. Proceedings of the 31st International Conference on Software Engineering. pp. 78-88.
[12] Dan W D, Qing W 2016 Improving the Performance of Defect Prediction Based on Evolution Data. Journal of Software.
[13] Xia X, Lo D and Pan S J 2016 HYDRA: Massively compositional model for cross-project defect prediction. IEEE Transactions on Software Engineering. pp: 977-998.