Hindawi
Journal of Control Science and Engineering
Volume 2020, Article ID 8814394, 10 pages
https://doi.org/10.1155/2020/8814394

Research Article

Applying Software Metrics to RNN for Early Reliability Evaluation

Hao Zhang,[1] **Jie Zhang**,[2] **Ke Shi**,[3] **and Hui Wang**[4]

[1]*School of Medicine Information, Wannan Medical College, Wuhu 241003, China*
[2]*School of Computer and Information, Anhui Normal University, Wuhu 241003, China*
[3]*School of Computer Science and Technology, Hefei Normal University, Hefei 230601, China*
[4]*School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230601, China*

Correspondence should be addressed to Jie Zhang; zjzj2526@163.com

Received 21 August 2020; Accepted 30 November 2020; Published 19 December 2020

Academic Editor: Juan-Albino Méndez-Pérez

Structural modeling is an important branch of software reliability modeling. It works in the early reliability engineering to optimize the architecture design and guide the later testing. Compared with traditional models using test data, structural models are often difficult to be applied due to lack of actual data. A software metrics-based method is presented here for empirical studies. The recurrent neural network (RNN) is used to process the metric data to identify defeat-prone code blocks, and a specified aggregation scheme is used to calculate the module reliability. Based on this, a framework is proposed to evaluate overall reliability for actual projects, in which algebraic tools are introduced to build the structural reliability model automatically and accurately. Studies in two open-source projects show that early evaluation results based on this framework are effective and the related methods have good applicability.

1. Introduction

Software reliability engineering aims to improve software quality and its role covers software life cycle. Recently, more research studies believe that reliability engineering implemented in the early stages of software development can significantly reduce potential risks such as rework costs from later stages [1–4]. Structural reliability models work in the early reliability engineering, but they are often difficult to be applied in actual projects due to lack of modeling parameters. From the recent empirical research on reliability modeling, most of them belong to the traditional software growth models (SRGMs) which are based on the testing failure data. Luan and Huang [5] study the distribution of faults in large-scale open-source projects by using the Pareto distribution which can obtain better prediction curve fitting accuracy than others. Sukhwani et al. [6] implement different SRGMs on NASA's flight control software for improving the development process and version management. Aversano and Tortorella [7] present a framework to evaluate the reliability of an ERP software which is based on bug reports. Honda et al. [8] use popular SRGMs in industrial software projects and discuss the performance of each SRGM. Tamura and Yamada [9] give a hierarchical Bayesian model which emphasizes the role of the fault detection rate in the reliability analysis of several open-source projects.

Most empirical studies do not cover early reliability evaluation because of the use of nonstructural models. In structural reliability models, Markovian models have been widely concerned which are state-based and emphasize structural analysis based on a specific granularity. Typical models include Littlewood's semi-Markov process (SMP) [10], Cheung's discrete-time Markov chain (DTMC) [11], and Laprie's continuous-time Markov chain (CTMC) [12]. These models are not easy to apply in practice. Take the DTMC model as an example. The two parameters required for modeling—component reliability and control transfer probability among components—are difficult to obtain from actual projects [13]. In this study, we aim to obtain the necessary information from software codes directly without testing failure data. From the perspective of improving the engineering process, we propose a complete framework for reliability modeling and calculating in design and coding period.

Software metrics, which measure codes from different perspectives, have been applied to the quality analysis and

the defect prediction in actual projects. Shibata et al. [14] incorporate the cumulative discrete-rate risk model with time-related measurement data and verify that the new model can obtain better predictive performance than popular nonhomogeneous Poisson process (NHPP) SRGMs. Chu and Xu [15] indicate a general functional relationship between complexity metrics and software failure rate which can be used in the exponential SRGMs. D'Ambros et al. [16] give the performance of several software defect prediction methods and the factors of threat validity in practice, which are based on static source code metrics and dynamic evolution metrics. In [17], the authors summarize the existing defect prediction models based on software metrics into four categories and explain how to aggregate them to achieve significant effect on performance evaluation.

The above studies do not involve reliability analysis and calculation. Besides, some researches reveal the relationship between the cognition of code complexity and the quality control. Fiondella et al. [18] point out that the complexity metric data could be utilized in cognitive modeling, which usually has characteristics of a low collection cost and various forms. Kushwaha and Misra [19] indicate the importance of the cognitive measure of complexity and implement it as the control quantity in a more reliable software development process. We consider that the cognitive information required for early reliability analysis is already included in the code structure, code metrics, and design documents. In this empirical study, we use multiple consecutive versions of code metrics and recognize the relationship between it and reliability changes based on some tools such as RNN. Then, the early reliability evaluation in the target version is carried out in order to assist decision-making in the development process.

The rest of this paper is organized as follows. Section 2 gives the framework, the RNN model, and the formal modeling tool used in this study for reliability assessment. Section 3 proposes the experimental methods, including object selection, metric data processing, and aggregation scheme. In Section 4, the experimental results are analyzed and discussed. By comparing the performance with other traditional reliability models, we prove the effectiveness of the proposed method and draw conclusions in Section 5.

2. Framework and Approaches

2.1. Framework of This Study. First, we give our framework as shown in Figure 1 to fully describe the methods used in this empirical study. The actual metric data will be divided into six categories based on their characteristics. These data collected from different release versions of a software are used as input for a series of RNN models. When a version is identified as the current version to be evaluated, the corresponding RNN model will be trained on data from historical versions in order to separate out the defect-prone classes. The next section describes in detail the RNN we used. Based on the specific strategy in Section 3, the classification results are aggregated into the module reliability value. And a DTMC model is established with all acquired parameter values in order to calculate the overall reliability. The formal tools will be used to facilitate the application of the DTMC. We will discuss the implementation details in the following sections.

2.2. The RNN Model. We use a simple type of RNN which has one hidden layer in the framework. Figure 2 describes its main structure.

As shown, the RNN propagates forward from initial state $\mathbf{s}^{(0)}$. The update equations for every time step from 1 to t are as follows:

$$\mathbf{s}^{(t)} = \tanh\left(\mathbf{U}\mathbf{x}^{(t)} + \mathbf{W}\mathbf{s}^{(t-1)} + \mathbf{b}\right), \tag{1}$$

$$\mathbf{o}^{(t)} = \text{sigmoid}\left(\mathbf{V}\mathbf{s}^{(t)} + \mathbf{c}\right), \tag{2}$$

where $\mathbf{b}$ and $\mathbf{c}$ are the bias vectors. The hyperbolic tangent function tanh $(\cdot)$ is the most commonly used activation function between input and hidden layers. And the logistic function sigmoid $(\cdot)$ is chosen for output function because we only deal with two classification problems here (defect-prone or reliable). The loss L is calculated as follows:

$$L = L^{(t)} = -\mathbf{y}^{(t)}\log \mathbf{o}^{(t)} - \left(1 - \mathbf{y}^{(t)}\right)\log\left(1 - \mathbf{o}^{(t)}\right) + \frac{\lambda}{2}\|\omega\|_2^2, \tag{3}$$

where $\lambda/2\|\omega\|_2^2$ is the item of $L2$ regularization to avoid overfitting. We first calculate the gradient $\nabla_{\mathbf{s}(t)}L$ of the last state $\mathbf{s}^{(t)}$ and update the weight matrix $\mathbf{V}$ based on controlled gradient descent. Then, we recursively calculate the gradient of all states from $\mathbf{s}^{(t-1)}$ to $\mathbf{s}^{(1)}$. The matrices $\mathbf{U}$ and $\mathbf{W}$ are updated during the iteration which is called back-propagation through time (BPTT). The dotted arrows in Figure 2 indicate its order of calculation. For the historical version used for training, all classes in each module are labelled with concrete $\mathbf{y}^{(t)}$ according to the actual test results. The application of this RNN model is detailed in Section 3.

2.3. Reliability Modeling and Formal Tools. Structural reliability models can calculate the software reliability without testing failure data. We choose the discrete-time Markov chain (DTMC) model [11] in the framework, which is the most popular one since it uses diagram similar to workflow to describe the control transfer relationships between modules. Assuming that the module N_i has the reliability degree R_i and the transfer probability $P_{i,j}$, the product $R_iP_{i,j}$ expresses the probability that N_i has been executed successfully and then transferred to N_j. This is the probability of one-step transfer in the Markov chain.

We can get the one-step probability of any pair of modules and form a matrix $\mathbf{Q}$ called one-step stochastic transfer matrix. The power $\mathbf{Q}^n$ is defined as an n-step transfer matrix. And the Neumann series of matrix $\mathbf{Q}$ is as follows:

$$\mathbf{S} = \mathbf{I} + \mathbf{Q} + \mathbf{Q}^2 + \cdots + = \sum_{k=0}^{\infty} \mathbf{Q}^k, \tag{4}$$

where $\mathbf{I}$ is the identity matrix.

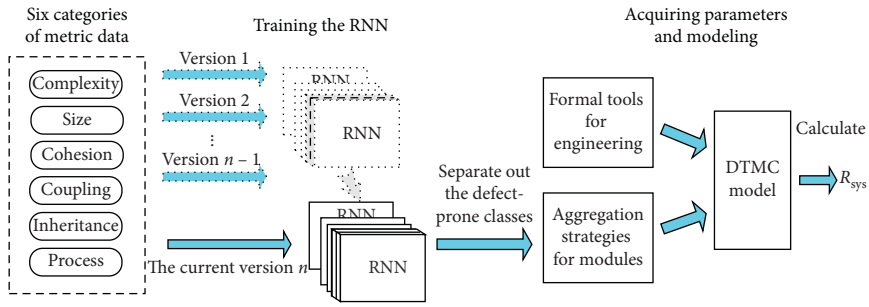

Figure 1: A framework of this study. The trained RNNs classify all classes of one module in the current evaluated version. A specified scheme receives class information and aggregates them to module reliability. The DTMC is used to calculate the overall reliability.

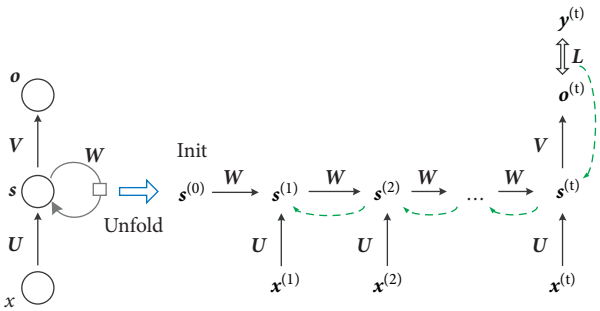

Figure 2: The many-to-one RNN model used in this study. The right is the unfolded form, where **U**, **V**, and **W** are the uniform weight matrices. The loss L between the only output $\mathbf{o}^{(t)}$ and the goal $\mathbf{y}^{(t)}$ is used in the backpropagation for updating model parameters.

Let us set the u^{th} row of **S** belongs to the starting module N_u and the v^{th} column belongs to the ending module N_v. The entry $S_{u,v}$ denotes the probability sum covers all possible transfer paths from N_u to N_v. So the system reliability R_{sys} can be computed as follows:

$$R_{\text{sys}} = S_{u,v} R_v, \tag{5}$$

where R_v is the reliability of N_v. Equation (5) is expressed as the probability of successfully reaching N_v and successfully executing N_v.

The DTMC model simply and effectively emphasizes the influence of local structure on the overall system, and it is suitable for reliability modeling in the early stage of software development. But it has difficulties in practical applications. The first is that the parameters R_i and $P_{i,j}$ are not easily known in practical applications. We will use specific strategies to solve this in Section 3.

The second is the construction of the DTMC model. Most DTMC modeling is based on directed graphs. In fact, there are currently no tools to support the automatic creation of such graph. As the number of modules increases, graphical representations and calculations will become more complex and difficult. Moreover, using only directed arcs is not enough to represent all relationships between modules in a local structure, such as parallelism.

We have proposed easy-to-use formal tools in our previous studies [20] for DTMC modeling. The basic idea is to use an algebraic expression $N_i \oplus N_j$ instead of the arc $N_i \longrightarrow N_j$ in the diagram. The operator $\oplus$ denotes the act of motivating between modules, which usually means the generation of control transfer. The advantage of this formal expression is that it is precise and unambiguous, especially when dealing with more complex and larger-scale situations.

In [20], we introduce more operators to express more complex relationships so that these algebraic operators can form a complete algebraic system. It is formally equivalent to ordinary algebraic expression, which can be automatically parsed by the automat of the formal language such as LL, LR, and SLR. Using this, we can automate the calculation of the DTMC model. We will explain in Section 4 how to use these tools to automatically build DTMC models and calculate the reliability of actual projects.

3. Experimental Design

The detailed experimental design and process are presented in this section. First, the actual software projects selected as the research objects are listed and the reasons are explained. Then processing methods for the used metric data are proposed. Finally, we aggregate the reliability values of software modules based on specific strategies.

3.1. Projects and Datasets. In this study, we use two open-source projects—jEdit [21] and Apache Ant [22]. There are popular and mature with strong developments and supports behind it. Both of them have the same version length in the PROMISE repository [23] which is the most important software metrics database. Table 1 shows the use of PROMISE's data in our scheme.

In Table 1, we set the current version of the two projects to 4.3 and 1.7, respectively. We have comprehensively considered the update of the data in the database and the impact on the effectiveness of this empirical study, although neither jEdit 4.3 nor Ant 1.7 is the last stable version. The two projects are properly sized and representative for development technology. In addition to the metric data, we also need to analyze the structural information. When we carry out reliability engineering at early stage, we consider ourselves as developers and designers. So we can get the necessary structural information from the design documentation and source codes.

Table 1: Training sets and test sets used in the RNN.

Project	Historical version (for training)				Current version (for test)
jEdit	3.2	4.0	4.1	4.2	4.3
Ant	1.3	1.4	1.5	1.6	1.7

Table 2 lists the structural information of the two target versions, including the description, and scale of some modules is used as example. We have also marked these packages with N_1, N_2, etc. Here, the granularity of module division is defined at the package level. We consider that it is appropriate to analyze the structure at the package level for projects developed in Java. And the corresponding level can be found in other language environments.

There are 23 packages (modules), 496 files, and 492 classes in jEdit 4.3. Ant 1.7 includes 15 packages (modules), 785 files, and 745 classes. Similarly, we need to seek the structural information of earlier versions. Furthermore, we need to check the previous version of one module in order to seek more detailed information at the design stage. It usually works because of the limited changes in modules between versions. As the coding continues, we can continuously adjust the metric data of one module.

4. Metric Data for RNN

Software metrics are generally classified into three categories: traditional metrics, object-oriented (OO) metrics, and process metrics [24]. Sometimes the traditional and OO are called code metrics. The PROMISE's data are, respectively, collected at the method level, class level, and file level. The number of lines of code (LOC) and the cyclomatic complexity (CC) are still valid at the method level within one class, which are used for measuring the basic size of codes. At the class and file level, the Chidamber–Kemerer (CK) [25] measurements are widely used, such as cohesion among methods (CAM), response for a class (RFC), and depth of inheritance tree (DIT). In addition, Darcy and Kemerer [26] emphasize perspectives of encapsulation and coupling, and the typical measurement element is coupling between object classes (CBO). Moser et al. [27] present eight metrics of process characteristics, and these are improved in the Madeyski–Jureczko (MJ) [28] measurements.

In this study, metric data from the PROMISE library are divided into six categories: complexity, coupling, cohesion, inheritance, size, and process. Table 3 lists all relevant metric elements we used in the RNN model as shown in Figure 2.

For one module, we identify defect-prone classes from the six aspects in Table 3. That is, we have to train six RNNs to solve these two classification problems.

Take the RNN-c_2 as an example. This is the model for the coupling category which is marked as c_2. We first initialize $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ randomly and then set $\mathbf{b}$, $\mathbf{c}$, and $\boldsymbol{s}^{(0)}$ to 0. Then an input series consisting of metric data vectors is used for training the RNN-c_2 model. Table 4 shows the input series when processing the BrowserView class in project jEdit.

The input series is a definite sequence ordered by four version numbers which corresponds to four vectors $\mathbf{x}^{(1)} \sim \mathbf{x}^{(4)}$. And the vector $\mathbf{x}^{(5)}$ is treated as test data. We define the same sequence for all classes in all modules and use $\mathbf{x}^{(5)}$ as test input too. The same training method is used in RNNs of the other five categories. The whole size of the training set of jEdit is 492 and that of Ant is 745.

Table 2: Structural information of the two target versions.

Project	Package	Description	Files	Classes	Mark
jEdit 4.3	Browser	File system browser	10	10	N_1
	Bsh	Bean shell	115	106	N_2
	Buffer	Buffer event listener	18	18	N_3
	Bufferio	I/O request for buffering	6	6	N_4
	Total	23 packages (modules)	496	492	
Ant 1.7	Dispatch	Actions dispatch for a task	3	3	N_1
	Filters	Input and output filtering	22	22	N_2
	Helper	Help functions	5	5	N_3
	Input	Input handler	6	6	N_4
	Total	15 packages (modules)	785	745	

Table 3: Metric division in this study.

Category	Metric elements	Mark
Complexity	AMC, MAX_CC, AVG_CC	c_1
Coupling	CBO, CA, CE, IC, CBM	c_2
Cohesion	LCOM, LCOM3, CAM	c_3
Inheritance	DIT, MOA, MFA	c_4
Size	WMC, NOC, RFC, NPM, LOC, DAM	c_5
Process	NR, NDC, NML, NDPV	c_6

Table 4: Input series of the RNN-c_2 (e.g., the class BrowserView of jEdit).

Coupling metric	Version				
	v3.2 $\mathbf{x}^{(1)}$	v4.0 $\mathbf{x}^{(2)}$	v4.1 $\mathbf{x}^{(3)}$	v4.2 $\mathbf{x}^{(4)}$	v4.3 $\mathbf{x}^{(5)}$
CBO	13	18	25	24	28
CA	8	11	14	15	15
CE	10	14	20	16	21
IC	0	1	1	1	1
CBM	0	4	4	4	4

4.1. Aggregation Strategies. In this section, we propose specific strategies to aggregate the RNNs' result into the reliability degree of one module and calculate the reliability value of the target projects.

The training RNNs classify all the classes into two types: defeat-prone and reliable. So the total number of defeat-prone classes can be counted in each aspect. For the defeat-prone class (DPC), we give the following definitions to mark the training data series:

(i) For a certain version, if a class has bug commit, it is defined as DPC

(ii) If a metric data of a class is obviously abnormal, the class is defined as DPC under the category including the metric as listed in Table 3

We assume that the historical version is aware of bug reports when we set jEdit 4.3 and Ant 1.7 as the current versions to be evaluated. This means that the classes in the current version are under development and there are no bug reports yet. This setting reflects the difficulty of early reliability assessment and is also in line with the actual situation.

The reliability influence (RI) of the c_i category in one module is defined by

$$\mathrm{RI}(c_i) = \left[1 - \left(\frac{N_{\mathrm{dpc}}(c_i)}{N_{\mathrm{all}}}\right)\right] * 100\%, \quad (6)$$

where $N_{\mathrm{dpc}}(c_i)$ is the number of defeat-prone classes in the module, N_{all} is the total number of module classes, and c_i indicates the specified aspect (from c_1 complexity to c_6 process). As shown in Table 4, the input vector $\mathbf{x}^{(5)} = [28, 15, 21, 1, 4]^{\mathrm{T}}$ only currently tests if the class BrowserView is defeat-prone from the coupling aspect. This still needs to complete the statistics from the remaining five aspects.

The RI value relates reliability to statistics. And the aggregation strategy is closely related to the calculation of the module reliability. Zhang et al. [17] point out that the summation strategy can often achieve the best performance when constructing models predict defect rank or count. Here, we use the summation strategy to aggregate the reliability influence (RI) of six categories (aspects) into the reliability of individual modules. It can be calculated as follows:

$$R_{\mathrm{module}} = \sum_i (r_i * RI(c_i)), \quad (7)$$

where r_i is the weight of the c_i category. Let r_i be equal to 1/6 here since every category represents a different logic and any one of them is important.

In the actual development, reliability engineering often needs to be implemented by the project manager. Module developers just calculate the R_{module} value based on (6) and (7). We introduce formal tools which are described in Section 2 to apply R_{module} into a DTMC model. We assume that module developers can submit R_{module} but also related algebraic expressions based on their understanding of the system structure.

For example, the developer of the browser (N_1) in jEdit 4.3 should submit (i) the module reliability value R_1 and (ii) the algebraic expression $N_1 \oplus N_{23}$. The expression $N_1 \oplus N_{23}$ replaces the directed arc to describe the control transfer flow between N_1 and N_{23}. In jEdit 4.3, the core package is recorded as N_{23}. This indicates that tasks via N_1 will be transferred to the core module.

The submitted algebraic expressions which confirm the design intent of developers link all modules in the workflow. The project manager and architect can also modify expressions directly based on their overall understanding. As a typical formal method, the algebraic expressions used here are precise and unambiguous, which is a lightweight and easy-to-use tool for software engineers.

As the actual managers of this study, we can collect an expression set finally which implicitly contains two key parameters required for the DTMC model: R_i and $P_{i,j}$. We have already discussed the calculation process of R_i which is the reliability of the i^{th} module in the target open-source projects. The transfer probability $P_{i,j}$ can be estimated by the Java static code analysis tools. For simplicity, the probability is equally divided by all possible transfer $N_i \oplus N_j$ in this study. A LR parser will be deployed for scanning and parsing all expressions in order to calculate the overall project reliability.

5. Results and Discussion

The experimental results and corresponding discussions are presented in this section.

5.1. Results. When we apply our framework of early reliability evaluation, the important problem is what kind of structure granularity is appropriate for the two practical projects. Early reliability engineering should be started at the design stage of software system which usually includes function module differentiation and architecture deployment. The structure information is contained in the module division and the relationship between modules. This means that we need to find the appropriate module granularity for structural analysis.

jEdit and Ant are both Java OO projects. As mentioned in Section 3, we suggest that the Java OO project should be analyzed at the package level since functions provided by one Java package are relatively independent. In addition, the metric data collected at the package level cover all collected at the class and method level in the PROMISE repository.

Figure 3 presents the distributions of eight modules of jEdit 4.3 in four metrics. The overall distribution of four metric data of all is also attached for comparison. As shown, different modules (packages) show significant differences, which indicate the diversity of function, complexity, and coding style. These differences denote possible quality gaps. This is in line with the fact that different modules of the jEdit project were developed by peoples located in several places. Besides, some mature packages are also used in the jEdit project, whose development cycle goes beyond the project itself. The Ant project has the same situation.

We use TensorFlow 1.4 to build the model RNN-c_i for six metric categories c_1–c_6. Table 5 lists classification results by six RNN-c_i for all classes in one package (e.g., the browser package of jEdit 4.3). 1 in the table represents defeat-prone, and 0 represents reliable.

By using (6) and (7), we calculate the R_{module} value of the browser package of jEdit 4.3 as 88.33%. Similarly, the R_{module} values of the remaining 22 packages in jEdit 4.3 and the R_{module} values of 15 packages in Ant 1.7 are also obtained.

In Table 1, 4.3 is the current version to be evaluated in the jEdit project and 1.7 is the current version in the Ant project. This means that the input series for jEdit are $3.2 \longrightarrow 4.0 \longrightarrow 4.1 \longrightarrow 4.2$, and series for Ant are $1.3 \longrightarrow 1.4 \longrightarrow 1.5 \longrightarrow 1.6$. The length of both series is 4. For

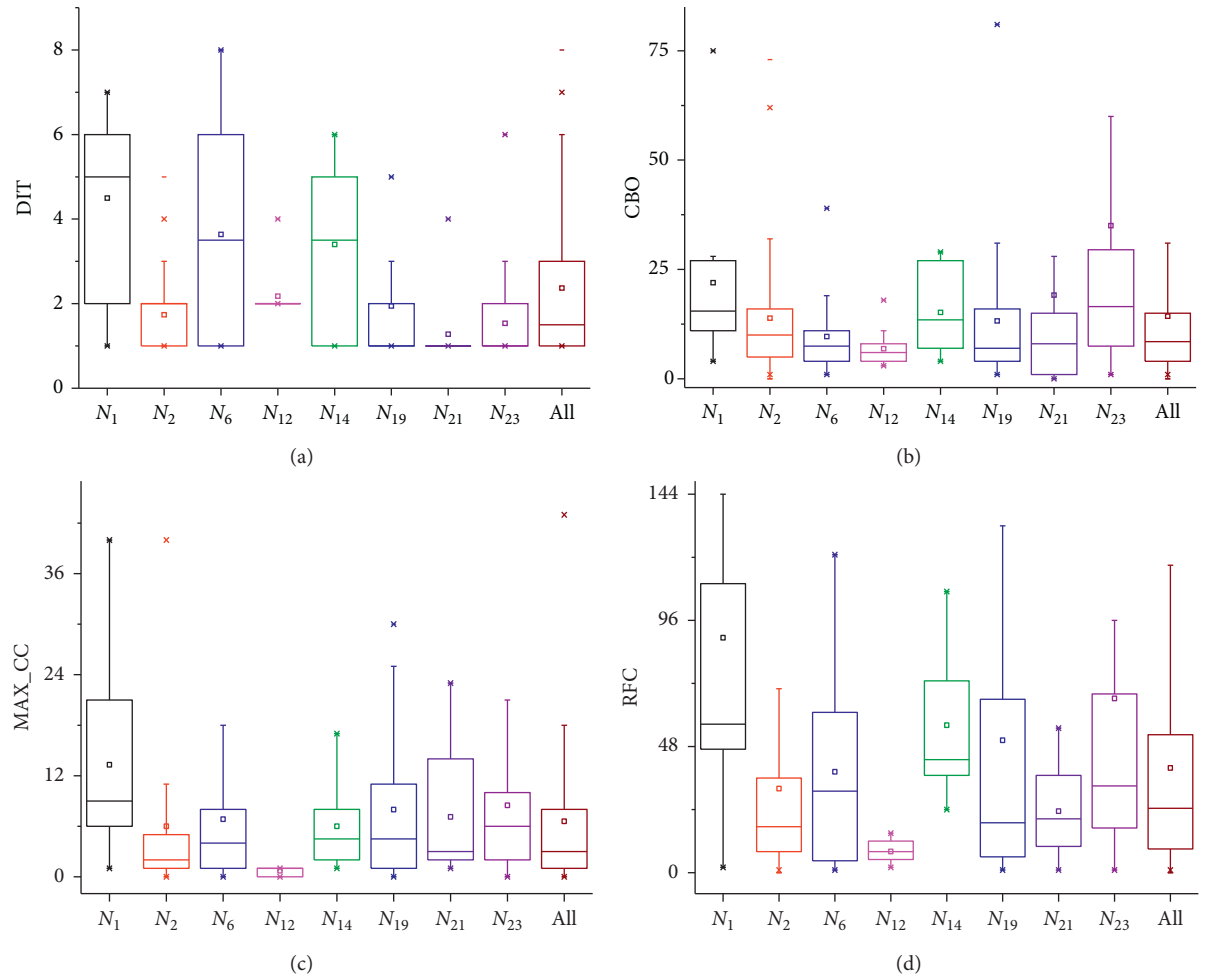

Figure 3: Boxplots of data distribution in four metrics (e.g., eight different packages and overall packages of jEdit 4.3): (a) the inheritance category represented by DIT; (b) the coupling category represented by CBO; (c) the complexity category represented by MAX_CC; (d) the size category represented by RFC.

Table 5: Classification results by RNN-c_i (e.g., the browser package of jEdit 4.3).

Class name	Classification results by six RNN-c_i					
	c_1	c_2	c_3	c_4	c_5	c_6
BrowserCommandsMenu	0	0	0	0	0	0
BrowserIORequest	0	0	0	0	0	0
BrowserListener	0	0	0	0	0	0
BrowserView	0	0	1	0	0	0
FileCellRenderer	0	0	0	0	0	0
VFSBrowser	1	0	1	0	0	0
VFSDirectoryEntryTable	0	1	0	0	0	1
VFSDirectoryEntryTableModel	1	0	0	0	0	0
VFSFileChooserDialog	0	0	0	0	0	1
VFSFileNameField	0	0	0	0	0	0

comparison, we can also set jEdit 4.2 and Ant 1.6 as the version to be evaluated. The corresponding input series are $3.2 \longrightarrow 4.0 \longrightarrow 4.1$ and $1.3 \longrightarrow 1.4 \longrightarrow 1.5$, respectively. The length of both series is 3. In extreme cases, we have the inputs $3.2 \longrightarrow 4.0$ for jEdit 4.1 and $1.3 \longrightarrow 1.4$ for Ant 1.5. The length of both series is 2.

Table 6 lists part of the R_{module} calculation results when taking jEdit 4.1, 4.2, 4.3 and Ant 1.5, 1.6, 1.7 as the version to be evaluated, respectively.

Here we present the first six modules in the two projects and boldface some items. The bold items in Table 6 describe the special circumstances that may be encountered during the calculation. There are three situations as follows:

(i) The R_{module} value of bufferio/bufferset in jEdit 4.3: these two modules are both newly added. In fact, they are derived from the module buffer in previous versions. The simplification of one module greatly increases the reliability value of the module buffer. This is in line with the original design. The value of the two new modules can only be estimated simply by not having a time series. The estimation rules refer to the DPC rules in Section 3.

(ii) The R_{module} value of dispatch in Ant 1.7: it belongs to the real new function module in the last version. Its value also needs to be estimated without using the RNNs.

Table 6: Part of the R_{module} values when taking jEdit 4.1, 4.2, 4.3 and Ant 1.5, 1.6, 1.7 as the version to be evaluated, respectively.

jEdit Package	$v4.1$	$v4.2$	$v4.3$
Browser	86.39	87.80	88.75
Bsh	88.41	91.84	94.38
Buffer	87.45	89.33	95.81
Bufferio	—	—	**93.54**
Bufferset	—	—	**94.11**
Gui	85.23	89.32	91.26
Ant Package	$v1.5$	$v1.6$	$v1.7$
Dispatch	—	—	**96.87**
Filters	92.35	95.64	97.02
Helper	89.12	91.08	94.14
Input	91.56	93.25	96.32
Launch	—	**91.39**	92.61
Listener	96.97	88.55	92.74

(iii) The R_{module} value of launch in Ant 1.6: in version 1.6, it belongs to the new functional module. But we still want to use RNNs in version 1.7. For this case, we construct a sequence of full length with the same value.

We calculate the overall reliability for three versions of the two Java projects by the DTMC model and the formal tools. From the perspective of reliability engineering, we need module developer to provide the relationship between modules in addition to R_{module}, which can be described conveniently by algebraic expressions. In the case of jEdit 4.3, an expression set which replaces the graphic representation of a DTMC model can be established as follows:

$$\{N1 \oplus N4, N1 \oplus N23, N2 \oplus N23, N3 \oplus N12, \&, N21 \oplus N23, N22 \oplus N23, N23 \oplus N24\}. \quad (8)$$

In the set, N_{24} is a virtual module constructed to indicate the termination state. It corresponds to the end node in the DTMC model. The R_{module} value of N_{24} is equal to 100%. Most modules are migrated to the core module N_{23}, and $N_{23} \oplus N_{24}$ indicates that business termination requirement is initiated only by N_{23}. In cases of jEdit 4.1, 4.2 and Ant 1.5, 1.6, 1.7, we have also constructed a set of algebraic expressions and used virtual modules to represent terminal nodes. These expression sets are automatically parsed by the LR parser to complete the calculation of system reliability.

We use different input series for verification. In addition, two traditional reliability models—the classical G-O model [29] and Huang's model [30]—have used for performance comparison. Both of them belong to the growth model. The former is representative of the classical model, and the latter has excellent predictive performance because of the integration of testing effort. These two models use testing failure data which can be obtained according to bug reports from the official website [21, 22]. That is, these models cannot be used for early reliability evaluation.

Figure 4 presents the evaluation results of the proposed framework for jEdit 4.1, 4.2, 4.3 and Ant 1.5, 1.6, 1.7 with different learning rates. We first set the initial learning rate η to 5 and the initial regularization rate λ to 1 in both projects. As shown, the curve is not as expected. Then two smaller learning rates—0.5 and 0.05—have been gradually experimented. We found that as the learning rate η becomes smaller, the curve trend is more and more consistent with the trend of the G-O model and Huang's model. As versions increase, the gap between the proposed method and traditional models is also narrowing. This reflects that a longer input series length will achieve better prediction performance, where the input series length of both of two target versions (jEdit 4.3 and Ant 1.7) is 4.

In Figure 5, we further experimented with different regularization rates to adjust the goodness of fit of the RNNs, which is based on the parameter λ in (3). We got the evaluation result 95.466% for jEdit 4.3 with $\eta = 0.05$, $\lambda = 0.02$, which is very close to the calculation result of Huang's model (96.224%). Similarly, the evaluation result 93.073% for Ant 1.7 with $\eta = 0.05$, $\lambda = 0.01$ is very close to the result of Huang's model (93.577%).

5.2. Discussion. We use two traditional reliability models for comparison in Figures 4 and 5. Curves of the proposed method are basically consistent with curves of traditional models. In fact, these traditional models are based on the testing failure data, which is different from our method based on software structure analysis and code metrics. It is unfair to directly compare the performance with traditional models. Results close to traditional models also show the effectiveness of the proposed method in this study. This means that reliability engineering could be implemented at the early development stage if the software metric data are used properly.

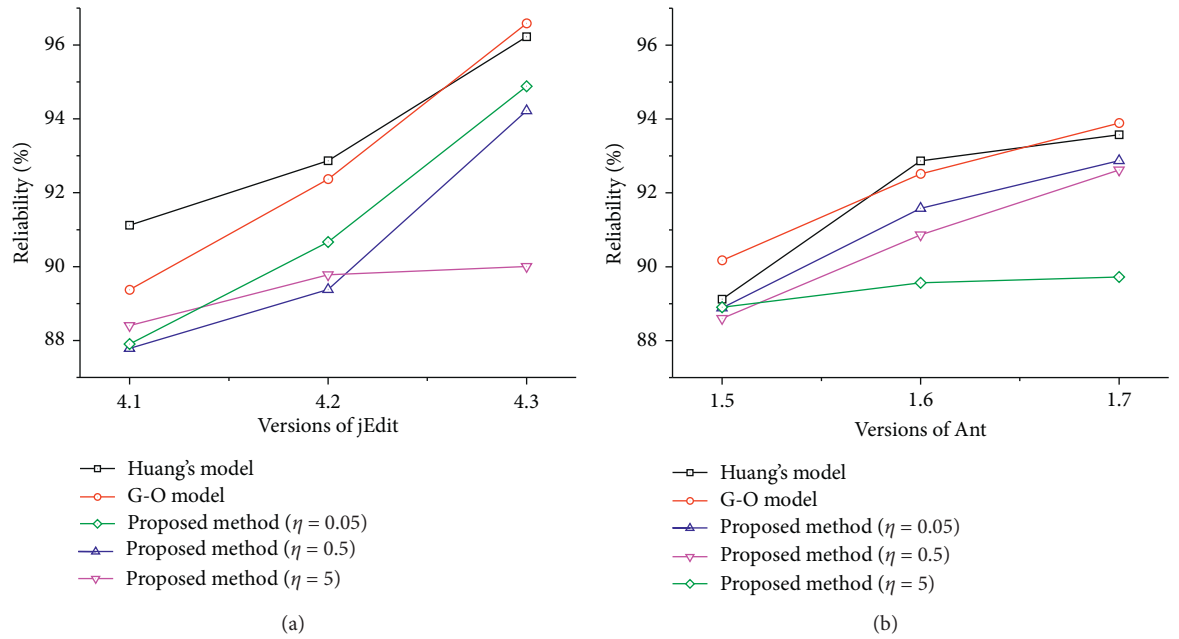

FIGURE 4: Reliability curves of the proposed method with different learning rates: (a) the learning rate η is set to 5, 0.5, and 0.05 on the target jEdit; (b) η is set to 5, 0.5, and 0.05 on the target Ant.

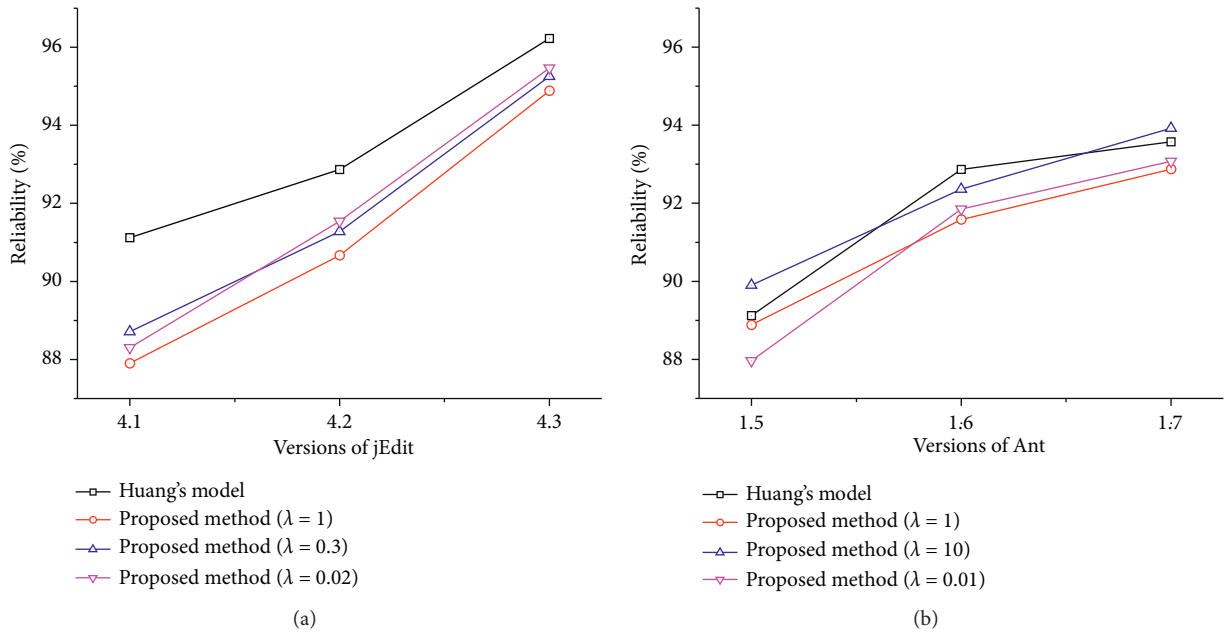

FIGURE 5: Reliability curves of the proposed method with different regularization rates when η is set to 0.05: (a) the regularization rate λ is set to 0.3 and 0.02 on the target jEdit; (b) λ is set to 10 and 0.001 on the target Ant.

As shown in Figures 4 and 5, the values of the proposed method are lower than those of traditional models usually. The reason is that the calculation process established by the framework of Figure 1 is dedicated to finding those defect-prone classes which can not necessarily lead to software failure. In other words, amplification of software code defects leads to conservative results of reliability evaluation in this study. In contrast, traditional models are relatively optimistic because they believe that the reliability curve of one software project will continue to increase with bugs fixing.

We tried different learning rates and regularization rates in the experiment. It can be seen that when the learning rate η is 5, a very different curve is obtained in both target projects. This slow-growing curve shows that the update of the RNN does not match the changes in the input series well. We then gradually reduced the learning rate and experimented with a wide range of regularization rates to optimize

the fitting problem. We found that under the data scale of this study, the optimal value for the parameter λ is at 10^{-2} magnitude.

The applicability of this study is explained as follows. The DTMC model used in the framework is the most important structural model at present [1–4, 9, 10]. Formal tools to match this model have been fully verified in our previous studies [20]. The software metric data needed for modeling can be obtained from public repository such as PROMISE or calculated directly from source codes. The metric elements and its classification used in the experiment are generally accepted in most literature studies of software defect prediction. In our follow-up research, the correlation analysis will be applied to eliminate redundant information in similar metrics.

6. Conclusion

In this study, we propose a complete framework in order to implement early reliability engineering in two open-source Java projects. We use the RNN model to process metric data for identifying defeat-prone classes in one package and calculate the module reliability of the package based on the definition of reliability influence and aggregation strategies. Then we introduce formal tools to automatically build the structural reliability model and calculate the overall reliability. Experiments show that results of the proposed method can approach the results of traditional reliability models which use failure data. Our method works at the software design and coding stage and can adapt to any structural changes and code changes in these stages. This study provides ideas for the practical application of the structural reliability model which plays an important role in reliability engineering. The next step of empirical research should be carried out for larger-scale, longer-period open-source software projects. Furthermore, structural analysis methods could be used to optimize the framework of this paper to reveal the impact of structural changes on reliability assessment in the process of software project version change.

Data Availability

The source code used to support the findings of this study is available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This work was supported by the Humanities and Social Science Foundation for Anhui Higher Education Institutions of China (SK2016A0964), the Natural Science Foundation for Anhui Higher Education Institutions of China (KJ2019A0482), and the Major Project of Quality Engineering for Anhui Higher Education Institutions of China (2020ZDXSJG369).

References

[1] F. Febrero, C. Calero, and M. Á. Moraga, "A systematic mapping study of software reliability modeling," *Information and Software Technology*, vol. 56, no. 8, pp. 839–849, 2014.

[2] H. Mei, G. Huang, L. Zhang, and W. Zhang, "ABC: a method of software architecture modeling in the whole lifecycle," *Science China-Information Sciences*, vol. 44, no. 5, p. 564, 2014.

[3] A. D. Plessis, K. Frank, M. Saglimbene, and N. Ozarin, "The thirty greatest reliability challenges," in *Proceedings of the Reliability and Maintainability Symposium*, pp. 1–6, Palm Springs, CA, USA, 2014.

[4] T. Dan, M. Galster, P. Avgeriou, and W. Schuitema, "Past and future of software architectural decisions—a systematic mapping study," *Inform. Software Tech.*vol. 56, no. 8, pp. 850–872, 2014.

[5] S.-P. Luan and C.-Y. Huang, "An improved pareto distribution for modelling the fault data of open source software," *Software Testing, Verification and Reliability*, vol. 24, no. 6, pp. 416–437, 2014.

[6] H. Sukhwani, J. Alonso, K. S. Trivedi, and I. Mcginnis, "Software reliability analysis of nasa space flight software: a practical experience," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security*, pp. 386–397, Vilnius, Lithuania, 2016.

[7] L. Aversano and M. Tortorella, "Analysing the reliability of open source software projects," in *Proceedings of the 10th International Joint Conference on Software Technologies*, pp. 348–357, Benevento, Italy, 2016.

[8] K. Honda, N. Nakamura, H. Washizaki, and Y. Fukazawa, "Case study: project management using cross project software reliability growth model," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*, pp. 41–44, Vienna, Austria, 2016.

[9] Y. Tamura and S. Yamada, "Reliability analysis considering the component collision behavior for a large-scale open source solution," *Quality and Reliability Engineering International*, vol. 30, no. 5, pp. 669–680, 2014.

[10] B. Littlewood, "Software reliability model for modular program structure," *IEEE Transactions on Reliability*, vol. 28, no. 3, pp. 241–246, 1979.

[11] R. C. Cheung, "A user-oriented software reliability model," *IEEE Transactions on Software Engineering*, vol. 6, no. 2, pp. 118–125, 1980.

[12] J.-C. Laprie, "Dependability evaluation of software systems in operation," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 701–714, 1984.

[13] S. S. Gokhale, "Architecture-based software reliability analysis: overview and limitations," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 1, pp. 32–40, 2007.

[14] K. Shibata, K. Rinsaka, and T. Dohi, "Metrics-based software reliability models using non-homogeneous poisson processes," in *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 52–61, IEEE Computer Society, Raleigh, NC, USA, 2006.

[15] Y. Chu and S. Xu, "Exploration of complexity in software reliability," *Tsinghua Science and Technology*, vol. 12, no. 1, pp. 266–269, 2007.

[16] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4, pp. 531–577, 2012.

[17] F. Zhang, A. E. Hassan, S. Mcintosh, and Y. Zou, "The use of summation to aggregate software metrics hinders the

performance of defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 476–491, 2017.

[18] L. Fiondella, A. Nikora, and T. Wandji, "Software reliability and security: challenges and crosscutting themes," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops*, pp. 55-56, Ottawa, Canada, 2016.

[19] D. S. Kushwaha and A. K. Misra, "Cognitive complexity metrics and its impact on software reliability based on cognitive software development model," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 2, pp. 1–6, 2006.

[20] J. Zhang, Y. Lu, and G. L. Liu, "Algebraic approach of software reliability estimation based on architecture analysis," *Systems Engineering and Electronics*, vol. 37, no. 11, pp. 2654–2662, 2015.

[21] http://www.jedit.org.

[22] http://ant.apache.org.

[23] http://openscience.us/repo/index.html.

[24] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, "Software fault prediction metrics: a systematic literature review," *Information & Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[25] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 11, pp. 197–211, 1994.

[26] D. P. Darcy and C. F. Kemerer, "OO metrics in practice," *IEEE Software*, vol. 22, no. 6, pp. 17–19, 2005.

[27] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the ACM/IEEE, International Conference on Software Engineering*, pp. 181–190, Cape Town, South Africa, 2008.

[28] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? an empirical study," *Software Quality Journal*, vol. 23, no. 3, pp. 393–422, 2015.

[29] A. L. Goel and K. Okumoto, "Time-dependent error-detection rate model for software reliability and other performance measures," *IEEE Transactions on Reliability*, vol. 28, no. 3, pp. 206–211, 1979.

[30] C.-Y. Huang, S.-Y. Kuo, and M. R. Lyu, "An assessment of testing-effort dependent software reliability growth models," *IEEE Transactions on Reliability*, vol. 56, no. 2, pp. 198–211, 2007.