Using reliability risk analysis to prioritize test cases

Ying Wang\textsuperscript{a}, Zhiliang Zhu\textsuperscript{a}, Bo Yang\textsuperscript{a}, Fangda Guo\textsuperscript{b}, Hai Yu\textsuperscript{a,}\textsuperscript{*}

\textsuperscript{a}Software College, Northeastern University, NO. 195, Chuangxin Road, Hunnan District, Shenyang, PR China
\textsuperscript{b}School of Computer Science and Engineering, Northeastern University, NO. 195, Chuangxin Road, Hunnan District, Shenyang, PR China

\section*{A R T I C L E   I N F O}

Article history:
Received 31 May 2017
Revised 4 January 2018
Accepted 22 January 2018
Available online 2 February 2018

Keywords:
Regression testing
Test case prioritization
Probabilistic risk analysis
Information flow
Complex network

\section*{A B S T R A C T}

In this paper, we present a risk-based test case prioritization (Ri-TCP) algorithm based on the transmission of information flows among software components. Most of the existing approaches rely on the historical code changes or test case execution data, few of them effectively use the system topology information covered by test cases when scheduling the execution of test cases. From the perspective of code structure, the proposed algorithm firstly maps software into an information flow-based directed network model. Then, functional paths covered by each test case are represented by a set of barbell motifs. Finally, combining with probabilistic risk analysis (PRA) and fault tree model, we assign a priority to each test case by calculating the sum of risk indexes of all the barbels covered by it. Experimental results demonstrate that Ri-TCP technique has a higher detection rate of faults with serious risk indicators and performs stably in different systems, compared with the other state-of-the-art algorithms.

© 2018 Elsevier Inc. All rights reserved.

1. Introduction

Regression testing is an important guarantee for software quality, whose purpose is to ensure that the modifications in previous versions of software meet the users’ requirements (Kung et al., 1996). In the life cycle of software testing, regression testing plays a significant role, which approximately accounts for 50% of the total maintenance cost due to its high frequency of being executed (Harrold, 2009). To improve the test efficiency and reduce test effort, testers might schedule test cases in an order according to some criterion to make the critical test cases be executed preferentially, this is so called “test case prioritization technique” (Elbaum et al., 2001b; Wong et al., 1997).

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement. Test cases are the cornerstones of quality assurance where they are developed to verify the quality and behavior of a product (IEEE, 2010). Test case prioritization technique aims to achieve code coverage at the fastest rate possible, increase assurance in reliability of the system at a faster rate, or improve the fault detection ability of test case suite during the testing process (Marchetto et al., 2016). A higher fault detection rate can provide earlier feedback on the system under test, enable earlier debugging, and increase the likelihood that, if the testing period is cut short, test cases that offer the greatest fault detection capacity in the available testing time will have been executed (Hao et al., 2016).

Numerous algorithms are proposed to address the test case prioritization problem. Of these, code-based prioritization techniques have drawbacks when dealing with large-scale software because of the statement and block level information is hard to manage (Ma and Zhao, 2008). The structural complexity-based prioritization strategy is to assign weights to classes based on the complexity of system topology, and then prioritize the test cases with the goal of maximizing the total or additional covered indicators. However, they ignored the information transmission relationships between classes that are covered by test cases. Thus, approaches to scheduling test cases by comprehensively analyzing the coverage information are valuable.

Risk analysis theory is successfully applied to software testing field, to improve productivity and reduce the costs of testing. The approaches Redmill (2005), Redmill (2004), Felderer and Ramler (2013), Felderer et al. (2012) and Amland (2000) addressed risk-based testing at a general level. Redmill (2005, 2004) emphasized the human and organizational factors. Employing risk as the basis for test planning did not provide a formula for perfection. Test planners must retain responsibility, but informed use of risk could provide illuminating guidance. Felderer and Ramler (2013) and Felderer et al. (2012) showed a model-based approach to risk-based testing, with the focus on product risks affecting the quality of the product itself. In Felderer and Ramler (2013), they pre-
sented a generic risk-based testing methodology and a procedure how it can be introduced in a test process. Based on this procedure, four stages of risk-based test integration were derived, i.e., initial risk-based testing, risk-based test reporting, risk-based test planning, and optimization of risk-based testing. Risk-based testing has a high potential to improve the software development and test process as it helps to optimize the allocation of resources and provides decision support for the management (Amland, 2000). Thus, considering risk indicators when prioritizing the test cases is essential for improving test efficiency.

In this paper, we propose a strategy combining three reliability risk factors – dynamic execution probability, fault-proneness and failure consequence – to schedule test cases to be executed. By equating the functional invocations with the transmissions of information flow, the software system is mapped into a class-level directed network model. Based on the complex network theory, we decompose the functional paths into a series of barbell motifs which consists of a class node pair and an information transmission relationship contained therein. With the aid of the fault tree model, we quantitatively analyze all the state events caused by the failure of each barbell motif in the system. Then, the risk index covered by test case is treated as a basis for ordering their execution. By comparing with the other state-of-the-art techniques based on several case studies, we show that the proposed approach performs better in effectiveness and stability across different software systems. The main contributions of this approach are summarized as follows:

- A class-level directed network model based on information flow for analyzing communication relationships between modules of software.
- An evaluation scheme for quantifying the risk indexes of classes in the system using the PRA model.
- A measurement to assess the risk coverage of test cases combining fault tree analysis and barbell motifs.
- A comprehensive comparison with previous studies from the perspective of detection rate of faults with high risk index.

The remainder of this paper is organized as follows. Section 2 discusses related research and Section 3 introduces the Ri-TCP technique. In Section 4, an evaluation indicator is described. In Section 5, we provide a comparison with previous research and discuss the experiment results. Finally, we give our conclusion in Section 6.

2. Related work

2.1. Test case prioritization techniques

Considering the coverage information as a target, test case prioritization techniques produce an optimal order for maximizing the coverage rate of certain factor (e.g., branch coverage, decision coverage, or statement coverage) as early as possible (Do et al., 2010). Rothermel et al. (2001) transformed the test case prioritization problem into a solution of searching the optimal order from all possible permutations of test cases. Its formalized definition is described as follows:

**Definition 1. Test case prioritization problem.** Given a test suite \( T \), the set \( PT \) consisting of all the permutations of test cases in \( T \), and a function \( f \) from \( PT \) to the set of real numbers, find a \( T' \in PT \) such that \( (\forall T'' \in PT \setminus \{T'\}) [f(T) \geq f(T'')] \).

Yoo and Harman (2012) surveyed the area of prioritization technique and discussed open problems and potential directions for future research. According to their paper, we categorized the existing approaches into four types: coverage-based prioritization, interaction testing, cost-aware test case prioritization and prioritization approaches based on other criteria.

**Coverage-based prioritization.** By analyzing the static call graphs of JUnit test cases and the program under test, Mei et al. (2012) prioritized the test cases in the absence of coverage information operating on Java programs tested under the JUnit framework. As dynamic coverage-based techniques use actual coverage information while their approach used estimated coverage information, the former was intuitively better than the latter in terms of fault-detection effectiveness. However, by avoiding the need to instrument code and execute test cases, this approach might be more applicable than dynamic coverage-based approaches in cases where gathering coverage information was inappropriate or was not cost effective.

Jeffrey and Gupta (2006) proposed a test case prioritization technique based on the coverage requirements presented in the relevant slices of the outputs of test cases. They called this approach the “REG+OI+POI” heuristic strategy for prioritization, where REG, denotes REGular statement (branches) executed by the test case, OI denotes the Output Influencing and POI denotes the Potentially Output Influencing statements (branches) executed by the test case. The experimental results suggested that accounting for relevant slicing information, along with information about the modifications traversed by each test case, had potential when used as part of the test case prioritization process.

**Interaction testing.** Interaction testing is required when system under test involves multiple combinations of different components. Bryce and Memon (2007) also applied the principles of interaction coverage to the test case prioritization of event-driven software. They extended the notion to t-way interactions over sequences of events. Prioritization by interaction coverage of events improved the rate of fault detection in half of our experiments. The test suites that include the largest percentage of 2-way and 3-way interactions had the fastest rate of fault detection when prioritized by interaction coverage.

Previous studies used tools to generate software interaction test suites have been evaluated on criteria of accuracy, execution time, consistency, and adaptability to seeding and constraints. Bryce and Colbourn (2005) prioritized interaction test cases based on user specified importance. For example, an operating system with a larger user base might be more important than one with a smaller user base. After weighting each level value for each factor, they calculated the combined benefit of a given test by adding the weights of each level value selected for the test. Computational results suggest that the greedy methods for constructing biased covering arrays could be useful when testers desire a prioritized ordering of tests.

**Cost-aware test case prioritization.** Yoo et al. (2009) introduced a test case prioritization technique, which can significantly reduce the required number of pair-wise comparisons by clustering test cases. The paper demonstrated that clustering without input parameters could outperform unclustered coverage-based technologies, and discussed an automated process that could be used to determine whether the application of the proposed approach would yield improvement.

Walcott et al. (2006) presented a regression test prioritization technique that used a genetic algorithm to reorder test suites in light of testing time constraints. Experiment results indicated that our prioritization approach frequently yields higher average percentage of faults detected (APFD) values, for two case study applications, when basic block level coverage was used instead of method level coverage. The experiments also revealed fundamental trade-offs in the performance of time-aware prioritization.

**Prioritization approaches based on other criteria.** Elbaum et al. (2001b, 2001a) performed a series of experiments to explore how the three factors-program structure, test suite composition, and change characteristics—affect the fault detection rate of test suites. Using a multiple regression model, they illustrate which metric
was the best predictor of APFD (average percentage of detected). The results indicated that the high prediction capabilities of the regression model for the optimal technique opened new opportunities for the evaluation of test suite orderings, because it could accurately estimate an upper threshold for prioritization potential without knowing the location of the faults.

Pan et al. (2012) evaluated the fault-proneness index and fault propagation influence of classes with the help of the weighted class dependency network model. On this basis, they assigned testing importance weight to each class, and then schedule the execution of test cases based on the sum of covered class weights. The empirical results on several systems suggested that the structural complexity-based test case prioritization technique is more effective than the other test case prioritization techniques and has relatively high stability.

The above-mentioned approaches are all based on code coverage or structural complexity metrics to prioritize test cases and ignore the reliability risk factors. To address this problem, we propose a risk evaluation model to assess the system risk coverage rate of test cases, which fully considers the topological structure characteristics of software and information flow transmission relationships between modules. Furthermore, the high-risk test cases are executed preferentially to improve the system reliability with maximum efficiency.

2.2. Risk analysis for software testing

In Erdogan et al. (2014) presented a systematic literature review addressing the combined use of risk analysis and testing. Focusing on risk-based testing, they described each of the approaches in great detail, and categorized them into eight types. In this section, we merge the work focusing on test cases into one type, and discuss the most relevant approaches.

Approaches addressing combination at a general level. Generally, the integrality of test before software release determines the reliability and robustness of system. During the testing process, introducing the potential risk information identified from intermediate results of testing and program structure as auxiliary measures, helps to locate faults in the system, thereby improving testing efficiency (Xie et al., 2013). For this reason, Felderer and Ramler (2014) developed a procedure on how risk-based testing could be introduced in a test process and derived a stage model for its integration. Moreover, they systematically discussed the potential benefits, prerequisites and challenges to introduce it.

Amland (2000) proposed an approach to risk-based testing and how risk-based testing was carried out in a large project in a financial institution. They defined the risk indicator as the product of two factors-fault rate and failure cost, and then integrate this measurement into the testing process. The paper concluded with how practical risk-based testing experience should inform theory and provided advice on organizational requirements that were necessary to achieve success.

In Felderer and Schieferdecker (2014) proposed a taxonomy of risk-based testing, which provided a framework to understand, categorize, assess, and compare risk-based testing approaches to support their selection and tailoring for specific purposes. It was aligned with the consideration of risks in all phases of the test process and consists of three top-level classes, i.e., risk drivers, risk assessment, and risk-based test process. Especially, the taxonomy of risk-based testing had been developed by analyzing the work presented in available publications on risk-based testing.

Approaches with focus on model-based risk estimation. Foidl and Felderer (2016) investigated how the information and data of a quality assessment based on the open quality model QuaMoCo can be integrated into risk-based testing. The contribution of their approach was, on the one hand, to show the potential usage of quality models for risk-based testing by presenting two integration approaches and, on the other hand, to provide a concrete integration including tool support and an empirical evaluation for the quality model QuaMoCo.

Adorf et al. (2015) presented a novel approach to risk-based test selection, which employed a comprehensive and versatile Bayes risk (BR) model taking defect probabilities and costs into account. Their investigation clearly showed that the opinions of suitably trained and guided experts were indeed informative, and could fruitfully be used for steering a quality assurance (QA) process. By carefully analyzing quantified opinions polled from several experts, one might obtain a valuable elementary predictor useful for selecting QA-tasks to be executed in a given test run. Combining the respective risk decrements with BR criterion, the selected tasks could be prioritized. From this priority queue of QA-tasks, a risk evolution curve might be constructed, which visualized the risk as a function of (QA-process) time.

Approaches with focus on regression testing. Felderer and Fournet (2015) presented a systematic classification of available security regression testing approaches by introducing a criterion with respect to security issues. They reported that most classical regression (functional) testing approaches were code-based. However, in regression testing of security requirements and security mechanisms, model-based approaches were dominating over code-based approaches. Code-based regression testing approaches were rather applied for the identification of vulnerabilities.

Chen et al. (2002) proposed a customer-oriented and also risk-based regression test selection technique, which provided methods to obtain both targeted tests and safety tests. To reduce risk efficiently, their selection strategy obeyed two rules: selecting scenarios that tested the most critical requirement attributes, and producing the test suite that covered as many requirements attributes as possible. To assess the validity of this approach, they applied it to three components of IBM WebSphere, the case study indicated that the technique was effective in finding defects.

Zimmermann et al. (2009) proposed an approach that allows to automatically generate test cases for risk-based testing of safety-critical systems. In their paper, Markov chain test models were used to describe the stimulation and usage profile of the system under test. Also, they refined the test models in such a way that only critical test cases can be generated. Another area of the research was to find similar model transformation operators permitting more complex failure scenarios, and then analyzed the operators found in this way for effectiveness and efficiency with regard to risk-oriented failure detection.

Although risk analysis model has been proven useful in software testing, few literatures involve in the definition of reliability risk factors from perspectives of component vulnerability, execution probability and failure consequence based on system topological structure, and its application in test case prioritization technique. This motivates us to extend the standard definition of risk from the financial area to software reliability analysis field to solve the practical engineering problem.

2.3. Network model for object-oriented (OO) software

Complex network theory enables us to describe the software from global point of view, which maps system into a network model and provide a valuable analytical dimension (Li et al., 2008). In 2002, Valverde et al. (2002) were the first to introduce the complex network theory into the software topological structure analysis. They expressed the OO software system as an undirected graph, more accurately, all the classes were denoted as nodes and relationships between them (e.g., inheritance, aggregation and association) are represented as edges. Soon afterwards, De Moura et al. (2003), Wheeldon and Counsell (2003),
Valverde and Solé (2003) and Myers (2003) used directed graph to depict the software collaboration. They highlighted that edge directionality was required to uncover several network features, such as: differences between in- and out-degree distributions, the anti-correlation between large in-degree and large out-degree, and the positive assortative mixing among out-degrees.

Network model is a powerful tool for characterizing the complexity of software system (Zheng et al., 2008). Wang et al. (2016) applied this theory in devising inter-class integration test order, which combines HITS (hyperlink-induced topic search) algorithm with the class-level directed weighted network model to identify the fault-prone nodes and those who have more error propagation ability, and then considered them as test focus with higher priority to be integrated. Pan et al. proposed an evaluation model to assign testing importance weight to each class of the system, by analyzing the dynamical behaviors of error spreading in the software network (Pan et al., 2016). Furthermore, the execution priority of test case was equal to the sum of class weight it covered, which is considered as the criteria to prioritize the regression test cases. The above-mentioned network model neglect the information flow transmission process in the software and the combination failure mode of information source, target and pipeline when a functional path segment fails. Thus, how to use the software topology to assess the risk factors and how to apply them in the regression test area remain a few problems to be worth thinking.

3. Ri-TCP methodology

The most common definition of risk in software projects is in terms of exposure to specific factors that present a threat to achieving the expected outcomes of a project. Software risk management focuses on six factors: asset, threat, vulnerability, safeguards, consequence and probability (Verdon and McGraw, 2004). The asset or object of the protection efforts, can be a system component, data, or even a complete system. The threat, or danger source, is invariably the danger a malicious agent poses and that agent’s motivations (financial gain, prestige, and so on). Threats manifest themselves as direct attacks on system security. A vulnerability is a defect, which enables an attacker to bypass security measures. Pfleeger and Pfleeger (2002) defines vulnerability as a weakness in the security system that might be exploited to cause loss or harm. Safeguard is the management, operational, and technical controls prescribed for an information system. Consequence is the impact on the organization, which can be monetary or tied to reputation, or it might result in the breach of a law, regulation, or contract. Probability is the likelihood that a given event will be triggered.

We consider asset as software system and see safeguard as testing scheme, in this manner, the risk analysis is incorporated into software testing process. The view of risk we adopt is from the financial management theory. To bring risk-based testing down to the source-code level, three risk factors—threat, vulnerability, and consequence—defined in our approach are different from the common concepts in risk management. Threat (hereinafter “execution probability”) is the likelihood that a component being executed; vulnerability (hereinafter “fault-proneness”) is the potential for a component fault; and consequence is the negative impact on system if the component fails. Consequently, risk exposure is determined by the above code characteristic dimensions.

The overall framework of Ri-TCP methodology is shown in Fig. 1. By analyzing the system under test, we construct the class-level network model. On this basis, the execution probabilities, vulnerabilities and failure consequences of nodes and edges in the software network are calculated using risk analysis theory. Furthermore, we evaluate the risk coverage weight combining the fault tree model with probabilistic risk analysis. Finally, the test case order is generated based on the additional greedy algorithm.

3.1. Class-level directed network model based on information flow

Let $S$ be any OO software system, $C_i$ be any class in system $S$, and $m_p$ and $a_q$ be any method and attribute in class $C_i$, respectively, then we have $S = [C_1, C_2, \ldots, C_k]$, $C_i = \{m_p, p \in \{1, 2, \ldots, NM_i\} \cup \{a_q, q \in \{1, 2, \ldots, NA_i\}\} = \{m_{i1}, m_{i2}, \ldots, m_{iNM_i}\} \cup \{a_{i1}, a_{i2}, \ldots, a_{iNA_i}\}$, where NC represents the total number of the classes in the system, NM$_i$ and NA$_i$ denote the number of methods and attributes in class $C_i$, respectively. To describe the topological structure of software system from the point of view of information transmission, we introduce the concepts below.

**Definition 2. Invocation and information transmission relationships between classes.** For any method $m_i \in C_i$, $m_j \in C_j$ and $i \neq j$, the information transmission relationship $m_i \rightarrow m_j$ exists iff at least one of the following three conditions are satisfied:

1. method $m_i$ calls $m_j$ and passes parameters to $m_j$, which is denoted as $m_i \xrightarrow{p} m_j$.
2. method $m_j$ invokes $m_i$ and that $m_i$ returns a value to $m_j$, which is written as $m_i \xrightarrow{R} m_j$.
3. method $m_j$ accesses the global variable $a_p$ and calls $m_i$, also, the value of $a_p$ is updated in the body of method $m_i$, which is represented as $m_i \xrightarrow{a_p} m_j$. It means that $m_i$ gives a feedback on $a_p$ to $m_j$.

---

**Fig. 1.** Overview of Ri-TCP’s architecture.
Here, $N_{ij}(m_{it}, m_{jk})$ represents the information flow transferred from method $m_{it}$ to $m_{jk}$ by one invocation, i.e., it is equal to the number of parameters or return value passed from $m_{it}$ to $m_{jk}$:

**Definition 3. Information flow.** The information transmission is the process that one method directly or indirectly passes parameters to another by invoking operations. If we denote the directed edge from class $C_p$ to $C_q$ as $< C_p, C_q >$, then the information flow transferred between them is:

$$L(C_p, C_q) = \sum_{m_{pi} \in C_p, m_{qj} \in C_q} N_{ij}(m_{pi}, m_{qj}) \times \text{Times}_{ij}$$

where $L(C_p, C_q)$ denotes the total information flow transferred from $C_p$ to $C_q$, i.e., it is the weight assigned to edge $< C_p, C_q >$; and Times$_{ij}$ is the number of transmissions from $m_{pi}$ to $m_{qj}$. Thus, the total information flow transferred in the system is:

$$TF = \sum_{C_i, C_j \in C} L(C_i, C_j)$$

**Definition 4. Class-level directed network model based on information flow, CDNMI F.** If we consider the classes as nodes and the information transmission relationships between them as edges, then the software system can naturally be described as a complex network, which is denoted as $G = (V, E, Z)$, where $V = \{C_i | i = \{1, 2, \ldots, NC\}\}$ is the node set, $E = \{< C_i, C_j > | i, j = \{1, 2, \ldots, NC\}, i \neq j\}$ is the directed relationship set and $Z = (z_{ij})_{NC \times NC}$ is the weighted adjacency matrix of network $G$, i.e., $z_{ij} = L(C_p, C_q)$. If $m_{pi} \in C_p$, $m_{qj} \in C_q$ and $p \neq q$, there is a directed edge $< C_i, C_j > \in E$ in network $G$ if the information transmission relationship $m_{pi} \rightarrow m_{qj}$ exists.

Consider a code snippet shown in Fig. 2. Fig. 3(a) describes the invocation and information transmission relationships between classes. Let $Cd.c(x, y)$ denote method $c(x, y)$ belonging to class $Cd$ and $Ca.a$ be the global variable defined in class $Ca$, where $x$ and $y$ are the parameters required by $c(x, y)$. For instance, method $Ce.b()$ invokes $Ce.c(x)$ and passes a parameter $x$ to it, thus we have $Ce.b() \xrightarrow{1} Ce.c(x)$. Method $Ch.b()$ is called by $Cb.b()$ and then $Cb.c()$ returns a value to it, i.e., $Cb.c() \xrightarrow{1} Ca.b()$. $Cd.c(x, y)$ uses global variable $Cd.b$ and depends on $Ce.a(x)$, meanwhile, $Cd.b$ is updated in the body of method $Ce.a(x)$, i.e., $Ce.a(x)$ gives a feedback on $Cd.b$ to $Cd.c(x, y)$. Thus, edge $Ce.a(x) \xrightarrow{1} Cd.b \xrightarrow{1} Cd.c(x, y)$ exists in the network. Fig. 3(b) shows its corresponding CDNMI F model, where the edge weight represents the information flow transferred between classes and Table 1 explains the details of the information transmission relationships.

From the definition of CDNMI F model, we can see that the dependency relationships between modules can be treated as the pipelines used to transfer the information flows. And the failure of nodes can be intuitively considered as the situation that information flows are blocked in the pipelines during the runtime. Thus, the risk factor of each component affects the system reliability. With the help of CDNMI F model, we can further analyze the importance of the node's logical location in the software network.

---

**Table 1**

<table>
<thead>
<tr>
<th>Edge</th>
<th>$L(G_i, G_j)$</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Cb \rightarrow Ca$</td>
<td>1</td>
<td>$Cb.c() \xrightarrow{1} Ca.b()$</td>
</tr>
<tr>
<td>$Cc \rightarrow Ca$</td>
<td>1</td>
<td>$Cc.b() \xrightarrow{1} Ce.a(x)$</td>
</tr>
<tr>
<td>$Cc \rightarrow Cd$</td>
<td>1</td>
<td>$Cc.c(x) \xrightarrow{1} Ce.b()$</td>
</tr>
<tr>
<td>$Cc \rightarrow Ce$</td>
<td>1</td>
<td>$Cc.c(x) \xrightarrow{1} Ce.a(x)$</td>
</tr>
<tr>
<td>$Cd \rightarrow Ca$</td>
<td>1</td>
<td>$Cd.b() \xrightarrow{1} Ce.a(x)$</td>
</tr>
<tr>
<td>$Cd \rightarrow Ce$</td>
<td>1</td>
<td>$Cd.b() \xrightarrow{1} Ce.a(x)$</td>
</tr>
<tr>
<td>$Ce \rightarrow Ce$</td>
<td>2</td>
<td>$Ce.b() \xrightarrow{1} Ce.c(x)$</td>
</tr>
<tr>
<td>$Ce \rightarrow Cd$</td>
<td>3</td>
<td>$Ce.b() \xrightarrow{1} Cd.c(x, y)$</td>
</tr>
</tbody>
</table>

**3.2. Probabilistic risk analysis model**

NASA-STD-8719.13A (NASA, 1999) standard defines several types of risk for software system such as, availability risk, acceptance risk, performance risk, cost risk, schedule risk, etc. In this paper, we focus on the reliability risk, which is a function of the anticipated probability that the software product will fail at run-
time, the potential severity of resulting consequences, and the uncertainties associated with the frequency and severity (Goseva-Popstoianova et al., 2003). By synthesizing these concepts, we adopted probabilistic risk analysis (PRA) model to assess the software system reliability.

In financial and engineering field, PRA model is considered as a powerful tool to quantify the heuristic risk factors of components or assets in the complex system or project (Bedford and Cooke, 2001). Using PRA model, we gain insight into the characteristics of complex system and locate the error-prone parts, in this manner, the early warnings can be provided before the system failure occurs. PRA model involves three indicators: threat, vulnerability and consequence, and that its standard definition is described as (Jonkman et al., 2003):

$$R(i) = T(i) \times V(i) \times C(i).$$

where $T(i)$ is the probability of an attack or stress being applied to component or asset $i$, $V(i)$ is the probability that the failure of the component or asset $i$ occurs, and $C(i)$ is the financial or fatality consequence if the failure occurs. We apply the above theory to evaluate the risk in an OO system and redefine the PRA model as follows:

- $T(C_i)$ - execution probability of class $C_i$, i.e., probability of class $C_i$ being executed;
- $V(C_i)$ - fault-proneness of class $C_i$, i.e., probability of class $C_i$ containing faults;
- $C(C_i)$ - consequence to system if class $C_i$ fails.

Hence, the risk of class $C_i$ can be calculated by $R(C_i) = T(C_i) \times V(C_i) \times C(C_i)$. The software risk is determined by failure events, occurrence possibility of failure behaviors and potential effect of failure. Combining with CDNMIF model, we can evaluate the execution probability, fault-proneness and failure consequence of nodes and edges in the system network.

3.2.1. Execution probability $T$

Suppose that all the source codes will be executed. In practice, user behaviors have a certain randomness. Core modules of the software are located in the hubs of the system network, as a result, they are executed frequently (Myers, 2003). In this approach, we simulate the running process of software based on a random walk model, thereby characterizing the dynamic information transmission behaviors between modules. The modeling process follows these two rules:

1. Growth: With the operations of users, edge and node scales continue to expand.
2. Preferential execution: Users tend to execute the core modules of system. The betweenness centrality of a node indicates the probability that this node is passed through by the information transmission paths in the system. This means that the higher betweenness centrality a node has, the more important its logical location is in the system network (Ma et al., 2011), thereby being executed more frequently. We jump to the next node to be executed, which is adjacent to the node being executed currently, according to the following probability:

$$\Pi(G_i) = \frac{BC_i}{\sum_j BC_j}, \quad BC_i = \sum_{s \neq i} \frac{n_{si}}{g_{si}},$$

where $NG$ represents the number of neighbors of the currently executing node, $BC_i$ denotes the betweenness centrality of the class $C_i$ which is connected to the currently executing class, $g_{si}$ is the number of shortest path from class $C_i$ to $C_s$, $n_{si}$ is the total number of shortest path from class $C_i$ to $C_s$ and passing through $C_i$. To guarantee the randomness and diversity of user operations and the integrity of function execution, we design the random walk model below.

First, select a starting node of any function randomly. Note that we consider the classes whose in-degree is zero and out-degree is greater than zero as the seeds of user operations. Function execution is a top-down process. Along the information transmission relationships between classes, we treat the neighbor nodes to which the currently executing class points, as the next step to be executed. Once the seed node is decided, we record the executed paths, and repeat the above steps for at least $LT$ times, until all

---

**Fig. 3.** An illustration for establishing the CDNMIF model.
the nodes and edges have been accessed. As shown in Eqs. (5) and (6), the execution probabilities of class $C_i$ and edge $< C_i, C_j >$ can be calculated as the ratio of the number of times that nodes and edges have been executed to the total number of accesses to all the nodes and edges, respectively. Algorithm 1 describes the calculation process of execution probabilities for nodes and edges in CDNMIF model. Its time complexity is $O(|E|)$, where $|E|$ is the number of edges. Here, we let $LT = 2000$.

$$T(C_i) = \frac{T_{Ci}}{\sum_{j=1}^{N_C} T_{C_j}}$$ (5)  

$$T(C_i, C_j) = \frac{T_{Eij}}{\sum_{p=0}^{N_C} \sum_{q=1}^{N_C} T_{Epq}}$$ (6)  

3.2.2. Fault-proneness (vulnerability) $V$

From the point of view of the complex system science, the complexity of the system topological structure determines the fault-proneness of software (Myers, 2003). In this paper, we use the concepts of "fan-in" and "fan-out" proposed by Henry and Kafura (1981) to measure the complexity of system. Let $l_{th}$ be lines of code in method $m_q$, $Fin_{pq}$ be the sum of the number of parameters passed into method $m_q$ and the number of return values received by $m_q$, and $Fout_{pq}$ be the number of parameters passed from method $m_q$. Then the fault-proneness of nodes and edges can be expressed as:

$$V(C_i) = \sum_{m_q \in C_i} lexh(m_q) \times (Fin(m_q) \times Fout(m_q))^2$$ (7)  

$$V(C_p, C_q) = \sum_{m_q \in M_{pq}} lexh(m_q) \times (Fin(m_q) \times Fout(m_q))^2$$ (8)  

where $M_{pq}$ is the method set that transfer information flow between class $C_p$ and $C_q$. More accurately, we consider the fault-proneness of edge $< C_p, C_q > \in E$ as the sum of the complexity of all the methods transferring information flows between class $C_p$ and $C_q$ and treat the fault-proneness of node $C_i \in S$ as the sum of the complexity of the methods defined in itself. The time complexity of the calculation process is $O(NM)$, where $NM$ is the total number of methods in the system.

3.2.3. Failure consequence $C$

A failure is a deviation of the system or system component behavior from its required behavior (Goseva-Popstojanova and Trivedi, 2000). On the other hand, a fault refers to an error (or anomaly) in the software, which may cause the a failure of system or system component. Thus, faults represent problems that developers see, while failures represent problems that the users (human or computer) see (Hamill and Goseva-Popstojanova, 2009). Faults can be introduced at any phase of the software life cycle, that is, they can be tied to any software artifact (e.g., design or source code). Different types of software faults lead to different extent of the damage to the system. According to NASA-STD-8719.13A standard (NASA, 1999), risk severity considers the worst-case consequence of a failure determined by the degree of system damage and mission loss that could ultimately occur (Goseva-Popstojanova et al., 2003). In terms of the ripple effect on software network, we suppose that if method $m_q \in C_i$ fails, then the errors will inevitably affect the methods that directly or indirectly depend on it, thereby blocking the transmission of information flows. To simplify the problem’s discussion, we adopt the assumption proposed by Pan and Li (2012) : (1) all the errors only affect the correctness of the system; (2) errors will be spread by information transmission.

Based on the points discussed above, we define failure consequence $C(m_k)$ as the maximum flow loss of system caused by the failure of method $m_k$.

$$C(m_k) = TF - \sum_{m_j \in M'k} N_{lf}(m_j, m_k) \times Times_{jk}$$ (9)  

$$\bar{M}'_k = M \setminus M'_k$$ (10)  

where $M$ represents the method set in the system, $M'_k$ is the method set that are reachable to method $m_k$ in the software network; $\bar{M}'_k$ denotes the difference set between $M$ and $M'_k$, i.e., it is a collection of working methods when method $m_k$ fails; $TF$ is the total information flows transferred in the system under normal working conditions, which can be qualified by Eq. (2); $N_{lf}(m_j, m_k)$ is the number of parameters passed from method $m_j \in \bar{M}'_k$ to $m_k \in M'_k$ by one invocation; and $Times_{jk}$ is the number of communications between $m_j$ and $m_k$. Combining with Algorithm 1, we can estimate the communication frequency between method pairs as follows:
(1) If methods $m_i$ and $m_k$ belongs to the same class $C_p$, then $\text{Time}_{jk}$ is equal to the number of times that class $C_q$ have been accessed by random-walking, i.e., $\text{Time}_{jk} = \text{timeN}[p]$.
(2) If method $m_j \in C_p$, $m_k \in C_q$ and $p \neq q$, then $\text{Time}_{jk}$ is considered as the number of executions of edge $<C_p, C_q>$ in Algorithm 1, i.e., $\text{Time}_{jk} = \text{timeE}[i]$.

Thus, the failure consequence of class $C_i$ is the average failure consequence of all the methods defined in itself:

$$C(C_i) = \frac{\sum \text{m} \cdot \text{c}(\text{m}_i)}{N M_i}$$  

Furthermore, the failure consequence of dependency relationship $<C_p, C_q>$ is the average failure consequence of methods that transfer information flows between classes $C_p$ and $C_q$:

$$C(C_p, C_q) = \frac{\sum \text{m} \cdot \text{c}(\text{m}_i)}{|M_{pq}|}$$  

**Algorithm 2** Failure consequences of methods in system.

**Require:** Adjacency matrix $A[NM][NM]$ of method-level network $G_m$

**Ensure:** Consequence $\text{c}(\text{m}_i)$

1. Initialize ArrayList table[NM] to record reachable node sets
2. for $i = 0 \rightarrow NM$ do
3. Initialize array vs[NM] to record if the nodes have been visited
4. table[i] $\leftarrow$ ArrayList $<\rightarrow$ ()
5. search(A, i, table[i])
6. table[i].remove(0)
7. end for
8. Calculate the total information flow $TF$ in network $G_m$
9. $\text{c}(\text{m}_i) \leftarrow \text{table}[k].\text{size}() \times \text{Time}_{jk}/TF$
10. function $\text{search}([\text{matrix}], \text{start}, \text{ArrayList})$
11. vs[start] = 1
12. $\text{ArrayList}.\text{add}($start$)$
13. for $i = 0 \rightarrow |V|$ do
14. if Matrix[start][i] = 0 & vs[i] = 0 then
15. $\text{search}([\text{matrix}], i, \text{ArrayList})$
16. end if
17. end for
18. end function

As shown in Algorithm 2, to improve the operation efficiency, the search strategy of our algorithm integrates a depth-first traversal with the effective pruning mechanism. Then, the failure consequences of all the methods are obtained by one execution. Accordingly, its time complexity is $O(NM^2)$.

### 3.3 Test case priority

Before assigning priorities for test cases, we introduce the following concept:

**Definition 5. Barbell motif.** During the runtime of the system, each step of information transmission is implemented by the collaboration of three components: source, information-pipeline and target. The three components constitute a barbell motif of "node-edge-node" in the software network.

The fundamental cause of the system failure lies in the failure of an information delivery implemented by one barbell motif. However, the failure of a barbell corresponds to $2^3 = 8$ possible state events: (1) Event(Source F, Pipeline N, Target N); (2) Event(Source N, Pipeline F, Target N); (3) Event(Source N, Pipeline N, Target F); (4) Event(Source F, Pipeline F, Target N); (5) Event(Source F, Pipeline N, Target F); (6) Event(Source N, Pipeline F, Target F); (7) Event(Source F, Pipeline F, Target F), where $N$ represents the component is working properly and $F$ denotes the component is at fault. From perspective of source codes, the failures of nodes and edges indicate that the output results deviate from the anticipate requirements.

To quantitatively analyze the risk coverage of test cases, we decompose the functional paths covered by test cases into a series of barbell motifs. Furthermore, with the aid of fault tree model, we provide the formal semantics of causal chain of the system failure. Taking the network model described in Fig. 3 as an example, if a test case covers the functional paths passing through classes $Ca, Ch, Cd$, and $Ce$, then these information transmission relationships can be refined into three barbells shown in Fig. 4.

Fault tree analysis (FTA) is a top down, deductive failure analysis in which an undesired state of a system is analyzed using Boolean logic to combine a series of lower-level events (Bucci et al., 2008). Fault tree is a causality graph, which uses event, logical gate and transfer symbols to express the cause-and-effect relationships between possible events in the system. The input events of a logical gate are the “reasons” of its output events, and that its output events are the “results” of the corresponding input events. In this manner, we obtain the state set event of a barbell motif under various working conditions. Fig. 5 shows the possible combinations of event states that lead to the failure occurrence, where the OR gate means that the output occurs if any input occurs.

Let $\forall(\text{C}_p). \exists(\text{C}_p, \text{C}_q) > 0$ and $\forall(\text{C}_q)$ be the probabilities that class $C_p$, edge $<C_p, C_q>$ and class $C_q$ have defects, respectively; $\text{T}(\text{C}_p), \text{T}(<\text{C}_p, \text{C}_q>)$ and $\text{T}(\text{C}_q)$ denote the probabilities of class $C_p$, edge $<C_p, C_q>$ and class $C_q$ being executed; and $\text{C}(\text{C}_p), \text{C}(<\text{C}_p, \text{C}_q>)$ and $\text{C}(\text{C}_q)$ be the failure consequences of class $C_p$, edge $<C_p, C_q>$ and class $C_q$, respectively. Any component has two execution statuses: being executed or not. For a barbell, there are $2^3 = 8$ event trees under different execution conditions. Due to space constraints, Fig. 5 describes the scenario that all the nodes and edges will be executed, i.e., the execution probability of any barbell motif $B$ in the system is equal to $\text{T}(\text{C}_p) \cdot \text{T}(<\text{C}_p, \text{C}_q>) \cdot \text{T}(\text{C}_q)$. For illustration, consider a barbell that class $C_p$ and edge $<C_p, C_q>$ have no defects and class $C_q$ contains errors. As shown in Fig. 5, its fault-proneness equals the product of the three event probabilities, i.e., $\forall(\text{C}_p, \text{C}_q) = (1 - \text{T}(\text{C}_p)) \cdot (1 - \text{T}(<\text{C}_p, \text{C}_q>) \cdot \text{T}(\text{C}_q))$. Obviously, its corresponding failure consequence is $\text{C}(\text{C}_p, <\text{C}_p, \text{C}_q>) = 0$. Let “+” denote that there are no errors in the component and “−” indicate that the component contains defects. Combining with the PRA model, we obtain the risk index of barbell $B$ under this case:

$$\text{Risk}(\text{C}_p, <\text{C}_p, \text{C}_q>)$$
Fig. 5. The fault tree model of a barbell motif under the condition that $T^b_i = T(C_i) \cdot T(< C_p, C_q >) \cdot T(C_q)$.

\[
R_i^b = \sum_{t=1}^{NT_i} R_i^b = \left[ \sum_{t=1}^{NT_i} \left( 1 - V(C_i) \right) \cdot \left( 1 - V(< C_p, C_q >) \right) \cdot V(C_q) \right] \cdot C(C_q) 
\]  

(13)

where $NT_i$ represents the number of barbell motifs covered by test case $TC_i$.

Assigning higher priority to the critical test cases helps to detect the more severe faults in the early testing steps, thereby improving the system reliability and test efficiency. In this paper, we prioritize the test cases combining the additional greedy algorithm with their risk coverage weights. The aim of the Ri-TCP technique is to maximize the risk coverage while considering the duplicated covered parts. Based on the feedback mechanism, the proposed strategy iteratively selects the test case which covers the maximum sum of the risk indexes of barbell motifs that have not already been consumed by previously selected elements. In other words, we require update the coverage information for each unselected test case to find the “next best”. Above all, the algorithm description of Ri-TCP is given below.

4. Evaluation indicators

4.1. Fault-detection efficiency

To measure the effectiveness of the prioritized list of test cases in detecting faults, we adopt the average percentage of faults detected per cost (APFDc) (Le Hanh et al., 2001) as an evaluation indicator:

\[
APFDc = \frac{\sum_{i=1}^{\left| \mathcal{F} \right|} \left( s_{f_i} \times \sum_{j=1}^{\left| \mathcal{F} \right|} c_{T_f} - \frac{1}{2} \times c_{T_f} \right) }{\sum_{i=1}^{\left| \mathcal{F} \right|} c_{T_f} \times \sum_{i=1}^{\left| \mathcal{F} \right|} s_{f_i}} 
\]  

(18)

$\mathcal{F}$: $\mathcal{F} = \{ f_1, f_2, \ldots, f_{\left| \mathcal{F} \right|} \}$ represents the set of faults that detected by test case suite, where $\left| \mathcal{F} \right|$ is the total number of faults in $\mathcal{F}$.

$s_{f_i}$: the severity of any fault $f_i \in \mathcal{F}$. If class $C_i \in V_c$ contains fault $f_i$, then $s_{f_i}$ is equal to the risk index of $C_i$.

c_{T_f}$: the costs of executing the $i$–th test case in order TCO. Suppose that $c_{T_f}$ is proportional to its execution time.

$T_{f_i}$: the ranking of the test case covering fault $f_i \in \mathcal{F}$ in order TCO.
Algorithm 3 Ri-TCP algorithm.

Require: The bytecode files of the software system under test.

The coverage information of the test case suite.

Ensure: TCO: Test case order.
1: Extract all the information transmission relationships between classes by analyzing the bytecode files of the software.
2: Map system into the CDNMIF model according to the dependencies.
3: Calculate the execution probabilities, fault-proneness and failure consequences of nodes and edges in the software network.
4: Decompose each test case into a series of barbell motifs.
5: Evaluate the risk coverage weight combining fault tree analysis with PRA model.
6: Combining the risk coverage with the additional greedy algorithm, we generate the test case order TCO.

4.2. System risk reduction rate

Suppose that a detected fault would be fixed and the risk index of tested modules would be dropped to 0. We introduce a metric called system risk reduction rate (SRRR) to measure the effectiveness of various prioritization techniques in terms of the rate of system risk clearance.

Once the top $i - 1$ classes of order TCO have been tested, the percentage of residual system risk is:

$$R_t = 1 - \frac{R_{clear}}{R_{total}} = 1 - \frac{\sum_{i} R(TG)}{\sum_{T} R(TG)}$$

(19)

According to the mapping relationships between the number of test cases having been executed $i \in |T|$ and residual system risk $R_t$, the rate of reduction in system risk can be calculated by the least square method (LSM):

$$SRRR = \frac{\left(\sum_{t=1}^{i} t^2\right) \left(\sum_{t=1}^{i} R_t^2\right) - \left(\sum_{t=1}^{i} t\right) \left(\sum_{t=1}^{i} R_t\right)}{i \left(\sum_{t=1}^{i} t^2\right) - \left(\sum_{t=1}^{i} t\right)^2}$$

(20)

A more efficient test case execution order leads to a lower residual amount and a higher rate of reduction in system risk. As a result, SRRR can be considered evaluation criteria for TCP techniques.

5. Experiments and data analysis

5.1. Research questions and experimental design

Based on the context of our study, we addressed the following research questions:

- **RQ1**: How is the distribution of code risk indexes covered by test cases?
- **RQ2**: Compared with the other state-of-the-art approaches, what are the advantages of the proposed TCP technique in effectiveness and stability?
- **RQ3**: Combining with the results from the other empirical studies, what are the factors affecting the success of TCP techniques?

Four open source software including Jmeter, Apache-ant, Joda-time and JFreeChart are adopted as our experimental subjects to verify the validity of the proposed approach. Note that we obtain the first two systems from Software-artifact Infrastructure Repository (SIR), whereas the other objects are from their own repositories. In particular, the four systems contain seeded and real faults and test case suites that have been widely used in research on fault localisation, test case suite selection, minimization and prioritization (Zhang et al., 2009; Fang et al., 2014; Rachatusumrit and Kim, 2012; Hao et al., 2014). The faults in Joda-time and JFreeChart software are identified by mining the bug fixing log files uploaded by original developers, according to the method proposed in (Just et al., 2014). Table 2 lists the detailed information about the experimental data. Following the orthogonal defect classification criteria (Chillarege et al., 1992), 15.1% of defects in the four systems are assignment type and the others are interface type.

Dependency Finder is a powerful tool that can extract multi-granular dependencies between modules from software. With the help of it, we map the systems into CDNMIF models as shown in Fig. 6. Furthermore, combining with jUnit tool, the test coverage information for each test case is obtained. On this basis, to address research question RQ1, we analyze the risk coverage distribution of test cases for all the systems.

To address research question RQ2, we compare Ri-TCP strategy with 9 coarse-grained TCP algorithms and 2 fine-grained TCP techniques and then discuss the experimental results. Table 3 shows the detailed descriptions of 11 comparable algorithms. Of these, T2 technique orders test cases randomly, in this paper, we obtain its average value of AFPD metric after repeating the experiment 50 times; T3 technique orders the test cases by maximizing the rate of severe-fault detection under given fault locations, thus, T3’s ranking is considered as the optimal result; and T8 and T9 techniques are proposed by Pan et al. (2012) and Ma and Zhao (2008), respectively; the former uses a structural complexity metric as the ranking criteria and the later prioritizes the test cases based on testing-importance measurement. All the simulations were performed on a personal computer with the following hardware environment: 3.7GHz CPU, 12 GB memory, and a 1TB HDD. The software operating environment was Windows 8.1 and the compiler platform was Eclipse 4.5.0.

To answer RQ3, combining with the empirical studies of approaches (Ma and Zhao, 2008; Rothermel et al., 2001; Pan et al., 2012; Zhang et al., 2013; Elbaum et al., 2002), we discuss the main factors that affect the effectiveness of the TCP techniques, i.e. the granularity of the coverage information (e.g., statement or method level strategies), the validity of the adopted fault proneness predic-

---

tors (e.g., complexity measurements or historical changes), and the coverage criteria (e.g., the total and additional strategies).

5.2. Results and discussion

5.2.1. Risk analysis for experimental systems

Fig. 7 shows the risk coverage distribution of test cases for all the systems. Clearly, few test cases cover high-risk function paths. If we define the high-risk level test cases as the ones whose risk coverage rates are greater than 10%, then, on average, the ratio of high-risk level test cases to the total number of test cases in four systems is 21%. Thus, the risk coverage distribution of test cases conforms to the Pareto principle, i.e., 80 percent of critical codes in the system are covered by 20 percent of test cases.

Taking Jmeter 1.8 software as an example, the No. 13 test case has the highest rate of risk coverage. By analyzing the source code, we find that test case InterleaveControl$Test covers 9 barbell motifs. More accurately, seven faults in classes InterleaveControl, HTTPSampler and PowerTable – Model can be detected with the its execution, which accounts for 36.8 percent of total. If the covered function paths cannot work as expected, then the software will lose 35 percent of information flows, which highly affects the system reliability. As a result, this test case should be assigned a higher priority to be executed.

Similarly, in software Jmeter 1.8.1, the No.22 test case ProxyControl$Test involves 83 functional paths and its risk coverage rate is nearly to 30% of total, thereby it has a higher fault detection ability. By contrast, the No.1 test case GzipTest in Apache-ant 1.3 software only covers 2 low-risk barbell motifs, which leads to a lower execution priority. From an overall perspective, 80% of system risk is covered by six critical test cases. These means that there is a sharp rise in system reliability when half of the test cases have been ex-
Table 3
Comparable test case prioritization techniques.

<table>
<thead>
<tr>
<th>Label</th>
<th>Prioritization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Untreated</td>
<td>Original ordering of test cases</td>
</tr>
<tr>
<td>T2</td>
<td>Random</td>
<td>Test cases are ordered randomly</td>
</tr>
<tr>
<td>T3</td>
<td>Optimal</td>
<td>Test cases are ordered to maximize the rate of severe fault detection</td>
</tr>
<tr>
<td>T4</td>
<td>Total method coverage</td>
<td>The test case with higher number of covered methods has higher priority</td>
</tr>
<tr>
<td>T5</td>
<td>Additional method coverage</td>
<td>Test cases are prioritized by the number of covered methods which are not yet covered by the executed ones</td>
</tr>
<tr>
<td>T6</td>
<td>Total diff method coverage</td>
<td>Test cases are prioritized based on the number of covered methods which differ from those methods in the previous release (e.g. modified or added methods)</td>
</tr>
<tr>
<td>T7</td>
<td>Additional diff method coverage</td>
<td>Test cases are prioritized using both feedback and modification information</td>
</tr>
<tr>
<td>T8</td>
<td>Structural complexity coverage</td>
<td>The technique is presented by Pan et al. [24], which prioritizes the test cases by maximizing the structural complexity coverage</td>
</tr>
<tr>
<td>T9</td>
<td>Testing-importance coverage</td>
<td>The technique is presented by Ma et al. [8], which orders the test cases based on the testing-importance metric</td>
</tr>
<tr>
<td>T10</td>
<td>Total statement coverage</td>
<td>Assigning higher priorities to the test cases covering more statements</td>
</tr>
<tr>
<td>T11</td>
<td>Additional statement coverage</td>
<td>The prioritization greedily selects a test case that yields the greatest statement</td>
</tr>
</tbody>
</table>

Fig. 7. Risk coverage distribution of test cases.
executed. Consequently, prioritizing the test cases based on their risk coverage information can significantly improve the test efficiency.

5.2.2. Effectiveness analysis

Table 4 lists the statistical results of Ri-TCP and 11 comparable techniques and Fig. 8 shows the APFDc boxplots for all the systems. From Fig. 8 we can see that the APFDc values of Ri-TCP technique are remarkably close to those obtained by T3 technique and higher than all the other 10 comparable results. Specially, in Joda-time and JFreeChart software, the APFDc value of the proposed approach reaches the maximum fault detection rates. Take Joda-time 2.5, for instance, 18 real errors can be detected by only 5 test cases, i.e., TempTest, TestBaseSingleFieldPeriod, TestDateTimeComparator, TestLocalDateTime_Basics and TestInstant_Basics. Among of them, TempTest, TestBaseSingleFieldPeriod and TestInstant_Basics can cover 10, 3 and 3 severe faults, respectively, and their executed sequence determined by Ri-TCP strategy is same as the order generated by T3 technique. Generally, the fine-grained coverage based prioritization technique is recognized more effective than the coarse-grained coverage based ones in fault detection (Do et al., 2006). However, as T10 and T11 techniques don’t consider the severities of errors, APFDc metric gives a slightly higher reward to Ri-TCP strategy.

Fig. 9 describes the comparison of the results between different software versions. For Apache-ant software, Ri-TCP technique gains the most dominant advantage on APFDc measurement. In Apache-ant 1.3 software, the No.31 test case CommandlineJavaTTest has the most risk coverage rate. Among the functional paths covered by it, 3 barbell motifs contain defects. Consider a barbell consisted of class ProjectHelper, PatternSet and the dependency between them. The execution probability of its source node, pipeline and target node are 0.17, 0.07 and 0.13, respectively. Thus, this barbell is passed though by the main function paths which are executed frequently. Moreover, the fault seeded in the barbell is interface type, its failure will lead to the blockages in information transmission or unexpected outputs. More precisely, it affects 36.2% of total information flows transferred in the system. Executing test

---

**Table 4**

APFDc values obtained by all the comparable techniques.

<table>
<thead>
<tr>
<th>System</th>
<th>Version</th>
<th>Ri-TCP</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
<th>T11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jmeter</td>
<td>1.8</td>
<td>0.923</td>
<td>0.930</td>
<td>0.860</td>
<td>0.654</td>
<td>0.886</td>
<td>0.890</td>
<td>0.894</td>
<td>0.910</td>
<td>0.884</td>
<td>0.907</td>
</tr>
<tr>
<td>Jmeter</td>
<td>1.8.1</td>
<td>0.940</td>
<td>0.948</td>
<td>0.596</td>
<td>0.707</td>
<td>0.881</td>
<td>0.887</td>
<td>0.936</td>
<td>0.936</td>
<td>0.904</td>
<td>0.915</td>
</tr>
<tr>
<td>Apache-ant</td>
<td>1.3</td>
<td>0.908</td>
<td>0.939</td>
<td>0.421</td>
<td>0.529</td>
<td>0.659</td>
<td>0.689</td>
<td>0.845</td>
<td>0.788</td>
<td>0.809</td>
<td>0.854</td>
</tr>
<tr>
<td>Apache-ant</td>
<td>1.4</td>
<td>0.969</td>
<td>0.984</td>
<td>0.36</td>
<td>0.817</td>
<td>0.664</td>
<td>0.704</td>
<td>0.740</td>
<td>0.760</td>
<td>0.894</td>
<td>0.908</td>
</tr>
<tr>
<td>Joda-time</td>
<td>2.5</td>
<td>0.936</td>
<td>0.936</td>
<td>0.466</td>
<td>0.735</td>
<td>0.757</td>
<td>0.741</td>
<td>0.721</td>
<td>0.650</td>
<td>0.591</td>
<td>0.881</td>
</tr>
<tr>
<td>Joda-time</td>
<td>2.6</td>
<td>0.941</td>
<td>0.941</td>
<td>0.703</td>
<td>0.714</td>
<td>0.762</td>
<td>0.783</td>
<td>0.836</td>
<td>0.861</td>
<td>0.887</td>
<td>0.864</td>
</tr>
<tr>
<td>Joda-time</td>
<td>2.7</td>
<td>0.946</td>
<td>0.946</td>
<td>0.753</td>
<td>0.586</td>
<td>0.743</td>
<td>0.757</td>
<td>0.827</td>
<td>0.818</td>
<td>0.892</td>
<td>0.877</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1.0.13</td>
<td>0.923</td>
<td>0.923</td>
<td>0.833</td>
<td>0.692</td>
<td>0.733</td>
<td>0.738</td>
<td>0.813</td>
<td>0.821</td>
<td>0.91</td>
<td>0.858</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1.0.14</td>
<td>0.948</td>
<td>0.948</td>
<td>0.662</td>
<td>0.640</td>
<td>0.718</td>
<td>0.725</td>
<td>0.928</td>
<td>0.828</td>
<td>0.918</td>
<td>0.849</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>0.927</td>
<td>0.944</td>
<td>0.695</td>
<td>0.675</td>
<td>0.756</td>
<td>0.668</td>
<td>0.827</td>
<td>0.819</td>
<td>0.89</td>
<td>0.879</td>
</tr>
</tbody>
</table>

![Fig. 8. APFDc boxplots for all the systems.](image-url)

(a) Jmeter

(b) Apache-ant

(c) Joda-time

(d) JFreeChart
case CommandlineJavaTest preferentially can effectively control the error spreading scope.

Table 5 describes the statistics of risk reduction rates caused by all the baseline prioritization techniques for nine the systems when 50 percent of test cases have been executed. Fig. 10 shows the comparison of reduction in system risk during the execution process of test cases. Clearly, Ri-TCP technique aims to maximize the risk coverage, as a result, the risk reduction rates caused by it, is greater than that of the other comparable techniques. By combining all the analysis results, we say that Ri-TCP strategy has a stable performance and high severe fault detection ability, as well as maximizes the overall reliability.

5.2.3. Stability analysis

To verify the prioritization stability of Ri-TCP technique, as shown in Table 6, we compare the mean square error (MSE) of APFD$_c$ values obtained by twelve baseline schemes performed on four software systems. In all the cases, the results of Ri-TCP technique are extremely close to the optimal ones obtained by T3 strategy, and the average MSE of APFD$_c$ metric is lower than the other comparable ones. Overall, the performance of Ri-TCP prioritization was highly stable with the different software systems.

Furthermore, as shown in Table 7, we adopt ANOVA (Analysis of Variance) for APFD$_c$ values between Ri-TCP and T3 technique. Obviously, all the effect-size values are low and p-values for all the systems are higher than 0.05. Thus, there are no significant differences in the fault detection ability between Ri-TCP and the optimal prioritization scheme. From the perspectives of effectiveness and stability, Ri-TCP strategy obtains satisfactory results when being performed on the test case suites and software system with different scales, and it is more appropriate for the detection of assignment type and interface type faults.

5.2.4. Discussion

The factors affecting prioritization success are the granularity of the coverage information, the effectiveness of the adopted fault proneness predictors (e.g., complexity metrics or historical changes), and the coverage criteria (e.g., the total and additional strategies). In this section, we discuss the above factors combining with the results from the other empirical study approaches.
Total and additional TCP techniques This dimension involves whether or not a technique employs feedback and is accounted for in the difference between “total” and “additional” techniques (Rothermel et al., 2001). In Zhang et al. (2013) explored the effectiveness of these two types of techniques. By a series of experiments, they concluded that in most cases, the additional techniques outperformed total techniques, but were not significantly different from each other. Moreover, the feedback mechanism had no effect when employing coarse granularity TCP techniques. There is a possible explanation for this. Techniques at the method level employing feedback give higher priority to test cases that execute uncovered methods, discarding methods already executed independently of the section or percentage of code in those methods that has actually been covered. If those partially covered functions are faulty, but their faulty sections have not yet been covered and the tests executing those functions are given low priority by techniques with feedback, then APFD values for techniques employing feedback could be lower. The results in our approach are consistent with (Zhang et al., 2013). The reason why the Ri-TCP technique has a better performance using the feedback mechanism, is that although the barbell motifs covered by test cases have the same source and target classes, we considered them as different ones if the pipelines between them formed by different method invocations. In this manner the risk coverage rate is extended (cover a larger surface).

Fine and coarse granularity TCP techniques Elbaum et al. (2002) approach concerned the relationship between fine and coarse granularity prioritization techniques. By investigation, they found that the coarse granularity techniques generally have relatively higher variation in the techniques’ performance across subjects. However, statement-level techniques were not significantly better than their corresponding method-level techniques. From the results of our empirical study, we can see that for all the method-level TCP strategies (T4–T9), there are only one-sixth of cases that perform slightly better than the statement-level techniques T10 and T11. The APFD$\times$ values of strategies combining fault proneness predictors (T8, T9 and Ri-TCP) ranked at the top. Similar conclusions were drawn in Ma and Zhao (2008), Pan et al. (2012) and Elbaum et al. (2002); the employed fault proneness measurement did significantly improve the severe fault detection ability of the coarse granularity prioritization techniques.

The coarser analysis used by method-level techniques renders them less costly and less intrusive than statement-level techniques. However, this coarser level of analysis could also have caused a substantial loss in the effectiveness of these techniques, offsetting efficiency gains (Elbaum et al., 2002). Satisfyingly, the proposed Ri-TCP strategy combining the risk index metric of barbell motif pattern with feedback mechanism effectively increases the risk coverage surface, and achieves a better performance than the finer granularity techniques.

5.3. Threats to validity

Programming language type. In this paper, we analyze the Java software systems using DependencyFinder tool, which can extract the dependency relationships at different levels of granularity. To facilitate obtaining data, all the adopted experimental subjects are coded in Java. Note, however, that the proposed Ri-TCP technique are equally applicable to the software implemented in other language.

Raw data. Test coverage reports for each test case provided by djUnit tool only contain the class-level testing information. To get more fine-grained coverage details, we use DependencyFinder tool to find the relationships between test cases and the classes in the
system, thereby obtaining the barbell motifs covered by test cases. As the correctness of coverage information significantly affects the experimental results, we provide the raw data to support further replication and research.

Experimental subjects. Ri-TCP prioritizes test cases based on the risk factors covered by them. Thus, the structural information of experimental subjects covered by their corresponding test case suites may affect the results. For instance, if a part of test cases cover quite a bit of high-risk barbell motifs of the system, then this may lead to higher fault detection and system risk reduction rates. Although the rankings of techniques did vary somewhat among experimental subjects, similarities did occur across all the programs, which leads to a higher stability of Ri-TCP strategy.

PRA model. We adopted the complexity metric proposed by Henry and Kafura to determine the fault-promineness of classes. However, in the source codes, there are several other factors that can potentially impact the fault-promineness of classes, such as the actual usage count of polymorphic references in a method, the intra-class coupling of methods, and so on. Nevertheless, on the basis of CDNMIF model, the concepts of “fan-in” and “fan-out” are appropriate to reflect the complexity of information flows transferred between components.

6. Conclusions

In this paper, we abstract the structural information of system covered by test cases into a series of barbell motifs by mapping the software into a CDNMIF model. Using fault tree analysis, we obtain all the possible ways leading to the failure of information transmissions between class pairs. Furthermore, the risk coverage of test cases are quantified based on PRA model. We schedule the execution of test cases combining the risk indexes of covered barbell motifs with the additional greedy algorithm. In this manner, the high-risk faults can be detected as early as possible, thereby the test efficiency is improved.

By comparing with the state-of-the-art TCT algorithms, we can draw the following conclusions:

- The fine granularity prioritization techniques that assign priorities to the test cases based on statement coverage rate (T10 and T11), generally have better performance on fault detection efficiency. However, APFD, metric didn’t reward them as they prioritized test cases without considering the fault severity measurement.

- Among the coarse granularity prioritization approaches, Ri-TCP technique performed slightly better than T8 and T9 techniques which also combined with the complexity metrics to schedule the test case execution, but significantly outperformed the method of diff method coverage-based prioritization schemes. Hence, the proposed risk index coverage indicator improves the prioritization technique’s ability of severe fault detection.

- Ri-TCP technique can maximize the system risk reduction rate. Especially, this advantage will be more obvious if the test case suite covers more high-risk barbell motifs of the system.

- The prioritization techniques based on historical code change information (T6, T7) are more appropriate to regression testing, while the complexity metric based techniques (T8, T9 and Ri-TCP) can be applied to solve the non-regression testing problem.

In future work, we plan to continue applying risk analysis to software testing field, in particular for those cross-project systems. For instance, the testing for cross-project systems have the disadvantages of insufficiency and low-efficiency. From the perspective of software topological structure, the high risk code patterns between the host software and the third-party libraries can be characterized and detected. With the help of the common code patterns, we can generate efficient test cases and provide appropriate test scheme for high risk cross-project interfaces.

Acknowledgments

The authors are grateful to all reviewers for the positive and valuable comments and suggestions regarding our manuscript. We improved the original version of this paper according to their high-quality feedback. This research was supported by the National Natural Science Foundation of China (Grant Nos. 61374178, 61402092, 61603082), The Online Education Research Fund of MOE Research Center for Online Education, China (Qtone education, Grant No. 2016ZD306), and the Ph.D. Start-up Foundation of Liaoning Province, China (Grant No. 201501141).

References

Ying Wang is a Ph.D. student in the Software College, Northeastern University, China. Her research interests include software refactoring, software architecture, software testing, and complex networks.

Zhiliang Zhu received his M.S. in computer applications and Ph.D. in computer science from Northeastern University, China. He is a fellow of the China Institute of Communications. His research interests include information integration, complexity software systems, network coding and communication security, chaos-based digital communications, applications of complex network theories, and cryptography. He has authored and co-authored over 130 international journal papers and 100 conference papers. In addition, he has published five books, including Introduction to Communication and Program Designing of Visual Basic .NET. Moreover, he is the recipient of nine academic awards at national, ministerial, and provincial levels. Prof. Zhu has served in different capacities at many international journals and conferences. He currently serves as co-chair of the 1st–9th International Workshop on Chaos–Fractals Theories and Applications. He is a senior member of the Chinese Institute of Electronics and the teaching guiding committee for Software Engineering under the Ministry of Education.

Bo Yang is a masters student in the Software College, Northeastern University, China. Her research interests include software refactoring, software architecture, software testing, and software risk analysis.

Fangda Guo is currently working toward the Ph.D. degree from the College of Computer Science and Engineering, Northeastern University, Shenyang, China. His research interests include software architecture database and graph data management, compressed sensing, signal and image processing, machine learning, computer vision, pattern recognition, and human-computer interaction.

Hai Yu received his B.E. in electronic engineering from Jilin University, China, in 1993 and his Ph.D. in computer software and theory from Northeastern University, China, in 2006. He is currently an associate professor of software engineering at the Northeastern University, China. His research interests include complex networks, chaotic encryption, software testing, software refactoring, and software architecture. He also serves as an associate editor for the International Journal of Bifurcation and Chaos, a guest editor for Entropy, and a guest editor for the Journal of Applied Analysis and Computation. In addition, he was a lead guest editor for Mathematical Problems in Engineering during 2013. Moreover, he has served in different roles at several international conferences, such as associate chair for the 7th IWCTFA in 2014, program committee chair for the 4th IWCTFA in 2010, chair of the Best Paper Award Committee at the 9th International Conference for Young Computer Scientists in 2008, and program committee member for the 3rd–9th IWCTFA and the 5th Asia Pacific Workshop on Chaos Control and Synchronization.