Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Devising optimal integration test orders using cost–benefit analysis*#

Fanyi MENG[1], Ying WANG[1,2], Hai YU[†‡1], Zhiliang ZHU[1]

[1]*Software College, Northeastern University, Shenyang 110169, China*

[2]*State Key Lab for Novel Software Technology, Nanjing University, Nanjing 210023, China*

[†]E-mail: yuhai@mail.neu.edu.cn

Received Sept. 30, 2021; Revision accepted Jan. 11, 2022; Crosschecked Mar. 24, 2022

Abstract: Integration testing is an integral part of software testing. Prior studies have focused on reducing test cost in integration test order generation. However, there are no studies concerning the testing priorities of critical classes when generating integration test orders. Such priorities greatly affect testing efficiency. In this study, we propose an effective strategy that considers both test cost and efficiency when generating test orders. According to a series of dynamic execution scenarios, the software is mapped into a multi-layer dynamic execution network (MDEN) model. By analyzing the dynamic structural complexity, an evaluation scheme is proposed to quantify the class testing priority with the defined class risk index. Cost–benefit analysis is used to perform cycle-breaking operations, satisfying two principles: assigning higher priorities to higher-risk classes and minimizing the total complexity of test stubs. We also present a strategy to evaluate the effectiveness of integration test order algorithms by calculating the reduction of software risk during their testing process. Experiment results show that our approach performs better across software of different scales, in comparison with the existing algorithms that aim only to minimize test cost. Finally, we implement a tool, ITOsolution, to help practitioners automatically generate test orders.

Key words: Integration test order; Cost–benefit analysis; Probabilistic risk analysis; Complex network
https://doi.org/10.1631/FITEE.2100466 **CLC number:** TP311

1 Introduction

Compared with procedure-oriented programming, object-oriented (OO) programming is characterized by encapsulation, polymorphism, and inheritance. Hence, there is a significant difference between these two types of software in the creation of test strategies (Binder, 1996). OO software involves four levels of testing: method, class, inter-class, and software testing (Tai and Daniels, 1999). By testing at the method and class levels, we determine whether each module of software is in working order. However, inter-class testing ensures that all modules can collaborate with one another (Jorgensen and Erickson, 1994). In integration testing, it is challenging to determine the order in which classes are integrated and tested in inter-class testing.

A class integration test order (CITO) is closely related to the software testing efficiency, as it affects the sequence in which classes are developed and inter-class faults are detected, as well as the design of test cases and construction of test stubs (Abdurazik and Offutt, 2006).

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (Nos. 61902056, 61977014, and 61603082), the Shenyang Young and Middle-Aged Talent Support Program, China (No. ZX20200272), the Fundamental Research Funds for the Central Universities, China (No. N2017011), and the Open Fund of State Key Lab for Novel Software Technology, Nanjing University, China (No. KFKT2021B01)

Electronic supplementary materials: The online version of this article (https://doi.org/10.1631/FITEE.2100466) contains supplementary materials, which are available to authorized users

ORCID: Fanyi MENG, https://orcid.org/0000-0001-6465-3295; Hai YU, https://orcid.org/0000-0002-8024-1781

© Zhejiang University Press 2022

The main concept underlying class order testing is to ensure that non-dependent classes are assigned higher priorities for integration testing, followed by the classes that are dependent on classes that have already been tested. In this manner, the numbers of test stubs and drivers are minimized, thereby reducing test costs. Integration test orders are generated by a reverse sort of classes based on the direct relationships between them if there are no cyclical dependencies in software. Moreover, due to the structural complexity of software, testers must perform cycle-breaking operations when test stubs are introduced (Assunção et al., 2014). In this process, test stubs are used to simulate interactive behaviors between class pairs, and provide attributes and methods for classes to be tested. Input values and expected outputs should be set beforehand to make sure that the simulated behaviors are consistent with the implementation of the actual classes. Thus, the entire simulation process is time consuming and complicated (Bansal et al., 2009). To reduce test costs, prevalent integration test order strategies (Kung et al., 1995; Tai and Daniels, 1999; Le Traon et al., 2000; Briand et al., 2002, 2003; Abdurazik and Offutt, 2006; da Veiga Cabral et al., 2010; Vergilio et al., 2012; Assunção et al., 2014; Jiang et al., 2021) focus on two aspects: reducing the number of test stubs and minimizing their total complexity (Wang ZS et al., 2011). However, a class integration test order is closely related to the sequence in which software bugs are detected. Such strategies share the limitation that they cannot expose software bugs immediately, which affects the test efficiency.

In addition, inherent risk in software projects leads to budget overruns and delays in the delivery of software products (Amland, 2000). The NASA-STD-8719.13A standard (NASA, 1999) defines several types of risk, including availability risk, acceptance risk, performance risk, cost risk, and schedule risk. In this study, we focus on reliability-based risk, which represents uncertainties associated with the frequency and severity of the failures of software components. Once the inter-class integration test order is generated, the sequence of detected faults is determined. Accordingly, the rate of reduction in software risk is determined. To illustrate this situation, consider an example software containing five classes with three faults. Fig. 1a shows their dependency relationships, Fig. 1b lists the exposed locations of the faults, and Fig. 1e describes their risk index distribution. A higher risk index for a class means a higher probability of malfunction and a more serious failure consequence. Having removed edge $<\mathrm{D},\mathrm{E}>$, we can obtain several solutions to order classes without stubbing efforts. Suppose that we place the classes in order A–C–D–E–B to be integrated. Figs. 1c and 1f show the ratio of detected faults versus the integration steps and the ratio of total risk covered by this order. Clearly, all faults have been detected until the integration in the last class is complete. The area under the curve represents the weighted average of the ratio of faults detected during the integration process. In this case, the measure is 50%. Figs. 1d and 1g reflect the scenario in which the class order is changed to A–B–D–E–C. Based on this integration solution, faster fault detection and total coverage rates are obtained. Accordingly, the test efficiency increases.

Testing high-risk classes early is crucial in the integration process. In this study, we refer to the higher-risk classes with higher testing priorities as test efficiency. According to existing works (Briand et al., 2002; Abdurazik and Offutt, 2006; da Veiga Cabral et al., 2010; Vergilio et al., 2012), we consider the test stub complexity metric as a test cost measure. Different class integration test orders lead to different fault-detection efficiencies and different costs of constructing test stubs. An efficient class integration test order can significantly improve the test efficiency and reduce the test cost. Therefore, we propose a strategy to devise an optimal integration test order that balances the priority of critical classes (obtained by the risk analysis model) against stubbing complexity when breaking cycles. Our insight is that special attention should be paid to critical classes with high risk indices, which potentially induce bugs and bring severe consequences to software users. As such, we encode two criteria in cycle-breaking operations: providing higher priorities to critical classes with higher risk indices, and minimizing the total complexity of the test stubs. We have conducted experiments on six open-source projects and compared their effectiveness with 13 state-of-the-art CITO generation algorithms. The experiment results showed that our approach outperforms the baseline approaches for software of different scales. With limited test cost, we can obtain a higher benefit rate in reducing software risk

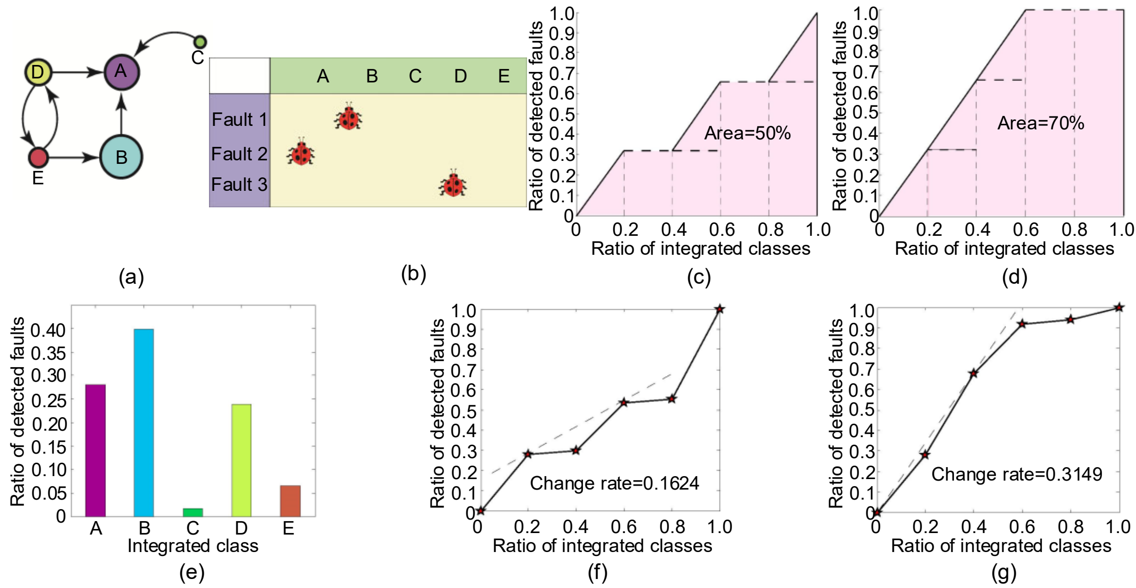

Fig. 1 An illustration of how the CITOs affect the test efficiency: (a) dependency diagram; (b) exposed faults in classes; (c) average ratio of faults detected for order A–C–D–E–B; (d) average ratio of faults detected for order A–B–D–E–C; (e) class risk distribution; (f) ratio of total risk covered by order A–C–D–E–B; (g) ratio of total risk covered by order A–B–D–E–C

compared with other approaches.

The main contributions of this study are summarized as follows:

1. Integration test priority measurements

We propose a multi-layer dynamic execution network (MDEN) model, which describes the complexity of software structure from multiple dimensions, to quantify the testing priority of each class based on probabilistic risk analysis.

2. An integration test order strategy to balance test efficiency and cost

From a cost–benefit perspective, we propose a strategy to devise an optimal integration test order, which ensures that higher-risk classes can be tested earlier (to improve the test efficiency) and minimizes the complexity of the test stubs (to reduce the test cost).

3. An evaluation scheme for CITOs

We present a scheme that assesses the effectiveness of integration test order by comparing the rates of change in system-level risk due to the integration steps. With the aid of this measurement, we conduct a comprehensive comparison with previous studies.

4. A publicly available tool and dataset

We provide a publicly available tool, ITOsolution, and raw data used in our evaluation to support further replication and research.

2 Related works

In this section, we briefly review related research from two perspectives: minimizing the number of test stubs and the total complexity of test stubs.

2.1 Minimizing the number of test stubs

Kung et al. (1995) first addressed problems in CITO algorithms and proposed a methodology for generating integration test orders. First, an OO software was modeled using object relation diagrams (ORDs). Second, strongly connected components (SCCs) in the graph were identified. Third, a random strategy was used, rather than heuristic information, to remove the edges when there were two or more candidate associations for cycle breaking. Finally, the orders were derived by sorting the classes based on the dependencies among them.

Tai and Daniels (1999) integrated classes based on their major- and minor-level numbers, whereby inheritance and aggregation relationships between classes were determined by the major-level numbers, and the minor-level numbers were determined according to association relationships between the classes of the same major level. As discussed in Briand et al. (2003), this solution is sub-optimal in terms of the required number of test stubs in cases

where class associations are not involved in cycles.

A graph-based strategy to devise inter-class integration test orders was proposed by Briand et al. (2003). The product of the number of incoming dependencies of a node a and the number of outgoing dependencies of a node b determined the directed edge weight $\langle a, b\rangle$. Then, the association edge involved in SCCs with the greatest weight was removed to break the cycles. More importantly, this study reviewed three main strategies proposed by Tai and Daniels (1999), Le Traon et al. (2000), and Briand et al. (2002), providing both analytical and empirical comparisons based on five case studies.

2.2 Minimizing the total complexity of test stubs

Briand et al. (2002) presented an approach to obtain optimal integration test orders by combining coupling measurements with genetic algorithms (GAs). First, the coupling measurements were used to differentiate stubs of varying complexities. Then, the algorithms can minimize complex cost functions based on such measurements. Tentative results showed that as dependencies and cycles became more complex, genetic algorithms tended to produce results that became less consistent with each execution.

Abdurazik and Offutt (2006) modeled test dependencies among classes using a weighted object relation diagram (WORD) and used fine-grained information to estimate stub complexity based on coupling measurements. Edges and nodes were assigned weights by quantitatively analyzing nine coupling types that were introduced. Their view was that if a class is used by multiple classes, all or part of the stub for that class may be shared among all classes that use it, thus reducing the cost of stubbing. For this reason, they assessed the cost of removing nodes according to node weights.

To optimize CITO, da Veiga Cabral et al. (2010) used a Pareto ant colony algorithm to produce test orders, which represented a suitable compromise between the number of attributes and methods in the stubbing process. Research has shown that this is a viable approach and provides better results in complex cases, such as testing software that contains a large number of dependency cycles.

Vergilio et al. (2012) and Assunção et al. (2014) introduced a generic approach based on multi-objective algorithms to solve the CITO problem for both OO and aspect-oriented contexts. The results demonstrated that the characteristics of software, instantiation contexts, and number of objectives could affect the performance of the algorithms. The Pareto archived evolution strategy (PAES) (Knowles and Corne, 2000) outperformed the other algorithms, even for more complex software. Considering all software and indicators, the non-dominated sorting genetic algorithm NSGA-II (Deb et al., 2002) is the most suitable for devising CITOs in most cases.

Jiang et al. (2021) presented an integration test order strategy considering the inter-class indirect relationships caused by control coupling, and proposed a novel approach to estimate the complexity of stubs created for a transitive relationship. They showed that the results could significantly reduce the stubbing cost when generating class integration test orders considering the transitive relationship. However, they considered only the test cost by controlling the stubbing complexity created for a transitive relationship, while our approach made a trade-off between test cost and test efficiency.

Our previous work (Wang Y et al., 2018a) aimed to improve the efficiency of regression testing. We proposed a risk-based test case prioritization (Ri-TCP) algorithm based on the transmission of information flows among software components. Experiment results indicated that the Ri-TCP technique has a higher fault detection rate with serious risk indicators than the state-of-the-art regression testing approaches. Afterwards, we preliminarily applied the above software risk analysis to the integration test area (Wang Y et al., 2018b). We first presented a new strategy for mapping the data flow interactions into a multi-granular flow network model. Based on it, we identified critical classes and assigned them higher test priorities when deriving CITOs. However, our previous risk analysis model cannot accurately describe the structure and behavioral characteristics of software in the execution process. In this study, to precisely evaluate risk factors in software, we propose an MDEN model, which depicts the dynamic features of a software program in both time and space. In addition, we provide a more comprehensive scheme to evaluate the effectiveness of our approach, including a risk distribution analysis of classes in real-world software, comparison with 13 state-of-the-art CITO generation algorithms, and

the benefits in testing resource allocation created by the generated CITOs.

However, prior studies of CITO algorithms consider only stubbing efforts (test cost) while ignoring the testing priorities of critical classes when generating CITOs. As a result, they cannot expose software bugs quickly. In our work, there is a trade-off between test cost and efficiency when deriving CITOs, by introducing the proposed criteria to cycle-breaking operations: providing higher priorities to critical classes with higher risk indices, and minimizing the total complexity of the test stubs. In the following sections, we provide a detailed description of the proposed approach and discuss our evaluation results.

3 Proposed approach

Four fundamental steps are involved in the proposed approach: MDEN model construction, probabilistic risk analysis (PRA) for classes, breaking cycles, and topological sorting. Fig. 2 gives an overview of our proposed approach.

3.1 Multi-layer dynamic execution network model

To identify breakable and unbreakable dependencies, modeling the relationships between classes is considered an important premise for generating CITOs (Briand et al., 2003). Three models have been used in previous work: unified modeling language (UML) diagrams, ORD, and test dependency graph (TDG). ORD is obtained by mapping from a UML diagram, which contains inheritance, aggregation, and association relationships. These coupling relationships are considered the basis for assessing possible stubbing efforts (Abdurazik and Offutt, 2006). To gain insight into method-level information, TDG extends ORD by extracting the details from the source code. Three types of dependencies are associated with this model: class to class, method to method, and method to class (Le Traon et al., 2000).

From the perspective of complex system science, the software topological structure affects software functionality, performance, and reliability (Myers, 2003). Software behavior shows a characteristic of dynamic evolution with the execution of its functions (Cai and Yin, 2009; Xu et al., 2020). Program behavior is essentially the collection of all its execution traces in different scenarios (Bowring et al., 2004). The execution traces represent the sequential function execution paths during runtime, and scenarios are used to describe the functionality and behavior of a software program from a user-centered perspective. Traditional static models cannot accurately describe the structure and behavioral characteristics of software in the execution process. To comprehensively evaluate risk factors in software, we propose an MDEN model that depicts the dynamic features of a software program in both time and space.

Each entity (method or attribute) defined in a class is considered a basic execution unit. Moreover, each action that is manually triggered by users is treated as an execution scenario. If we represent the entities as nodes and the dependencies between them

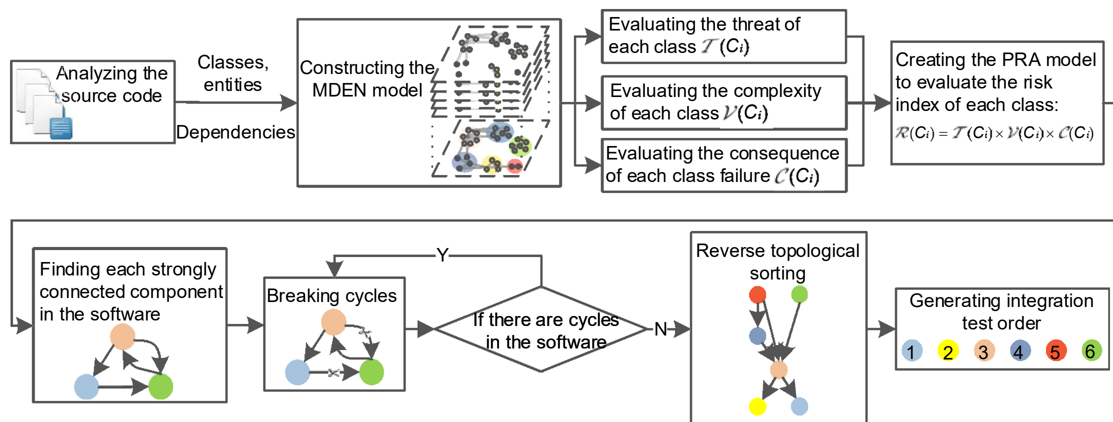

Fig. 2 Overview of our approach

as directed edges, an execution scenario is equal to a function profile composed of function execution traces; hence, a set of function profiles in various scenarios can be modeled as an MDEN.

Let $\mathcal{S}$ be any OO software and C_i any class in the software; then, $\mathcal{S} = \{C_1, C_2, \cdots, C_{\mathrm{NC}}\}$, where NC is the number of classes. For any class $C_i \in \mathcal{S}$, $C_i = \{m_1, m_2, \cdots, m_{\mathrm{NM}_i}, a_1, a_2, \cdots, a_{\mathrm{NA}_i}\}$, where m_t represents any method defined in class C_i, a_k denotes any attribute of class C_i, and NM_i and NA_i are the numbers of methods and attributes in class C_i, respectively, each scenario can be treated as a top-down function execution process triggered by users. Assume that $V_{\mathrm{et}} = \{m_1, m_2, \cdots, m_{|V_{\mathrm{et}}|}\}$ is the function entry method set; i.e., $m_i \in V_{\mathrm{et}}$ is the method whose in-degree is zero and out-degree is greater than zero in the static method-level dependency network. We define the MDEN model as follows:

Definition 1 (MDEN) Let network $G_i = (V_i, E_i)$ be the function execution profile in scenario s_i, where V_i denotes the entity set that is directly or indirectly dependent on the function entry method $m_i \in V_{\mathrm{et}}$ at runtime, and E_i represents the directed dependencies among the entities of set V_i. Then, MDEN can be written as $\mathcal{G} = \{G_i \mid i \in \{1, 2, \cdots, |V_{\mathrm{et}}|\}\}$.

If method m_t invokes abstract method m_k belonging to abstract class C_t, we assume that m_t depends on all concrete methods that implement method m_k, defined in the subclasses of C_t. In this manner, the dynamic binding mechanism can be described in MDEN. Note that we ignore the inherited entities in subclasses that are not invoked by other methods. This assumption can help avoid redundant testing without any negative effect.

As one network layer corresponds to a snapshot of the software execution process in a scenario, the degree of overlap of edges between network layers represents the frequency of continuous execution of method nodes connected by them. Let m_i and m_j be a method pair, and NM be the number of methods in the software for any $i, j \in \{1, 2, \cdots, \mathrm{NM}\}$, $i \neq j$, and $t \in \{1, 2, ..., |V_{\mathrm{et}}|\}$. The continuous execution frequencies of m_i and m_j satisfy

$$\tau(m_i, m_j) = \frac{1}{|V_{\mathrm{et}}|} \sum_{t=1}^{|V_{\mathrm{et}}|} \delta_{ij}^t, \tag{1}$$

where δ_{ij}^t is defined by

$$\delta_{ij}^t = \begin{cases} 1, \text{ if} \langle m_i, m_j \rangle \in E_t, \\ 0, \text{ otherwise.} \end{cases} \tag{2}$$

Fig. 3 shows an example software containing six classes (https://github.com/FanyiMeng-NEU/Class-Example). Five functions exist in the software: menu initialization (InitializeMenu. main(String[])), goods purchase (OrderService. buy(Goods, String)), order details display (OrderService.ShowOrderDetails(Order)), goods distribution (Distribution.distributeOrder(String)), and goods comment submission (Goods.SubmitGoods-Comments(String, String, Order)). The corresponding MDEN model of the sample code is shown in Fig. S1 (see supplementary materials for Figs. S1–S6).

For example, once the function entry Submit-GoodsComments(String, String, Order) is triggered by users, entities Goods.setComments(String), Order.getGoods(), and OrderService.orders are executed. Then, entities Order.goods and Goods. comments are used by Order.getGoods() and Goods.setComments(String), respectively. The above scenario is expressed in layer 1. Similarly, the other function profiles can be obtained by mapping the execution traces into method-level network layers. From Fig. 3, we can observe the structural relationships between modules and their execution sequences. In particular, edge Order.getOrderNO() → Order.orderNO appears in three network layers, layers 2, 3, and 5. Thus, we say that the continuous execution frequency of methods Order.getOrderNO() and Order.orderNO is 0.6.

Compared with existing models, the main improvements effected by MDEN are as follows:

1. Software is depicted at a more fine-grained level. Five types of coupling relationships between methods, attributes, and classes are taken into consideration: class to class, method to method, method to class, class to attribute, and method to attribute. Moreover, the details of dynamic execution scenarios can be mapped into function profiles.

2. In Le Traon et al. (2000)'s approach, the notion of weight is defined on classes, aimed to capture the number of cycles in which the nodes are involved. However, the aim of assigning weights to the nodes and edges based on information transmission details within the MDEN model is to estimate the execution

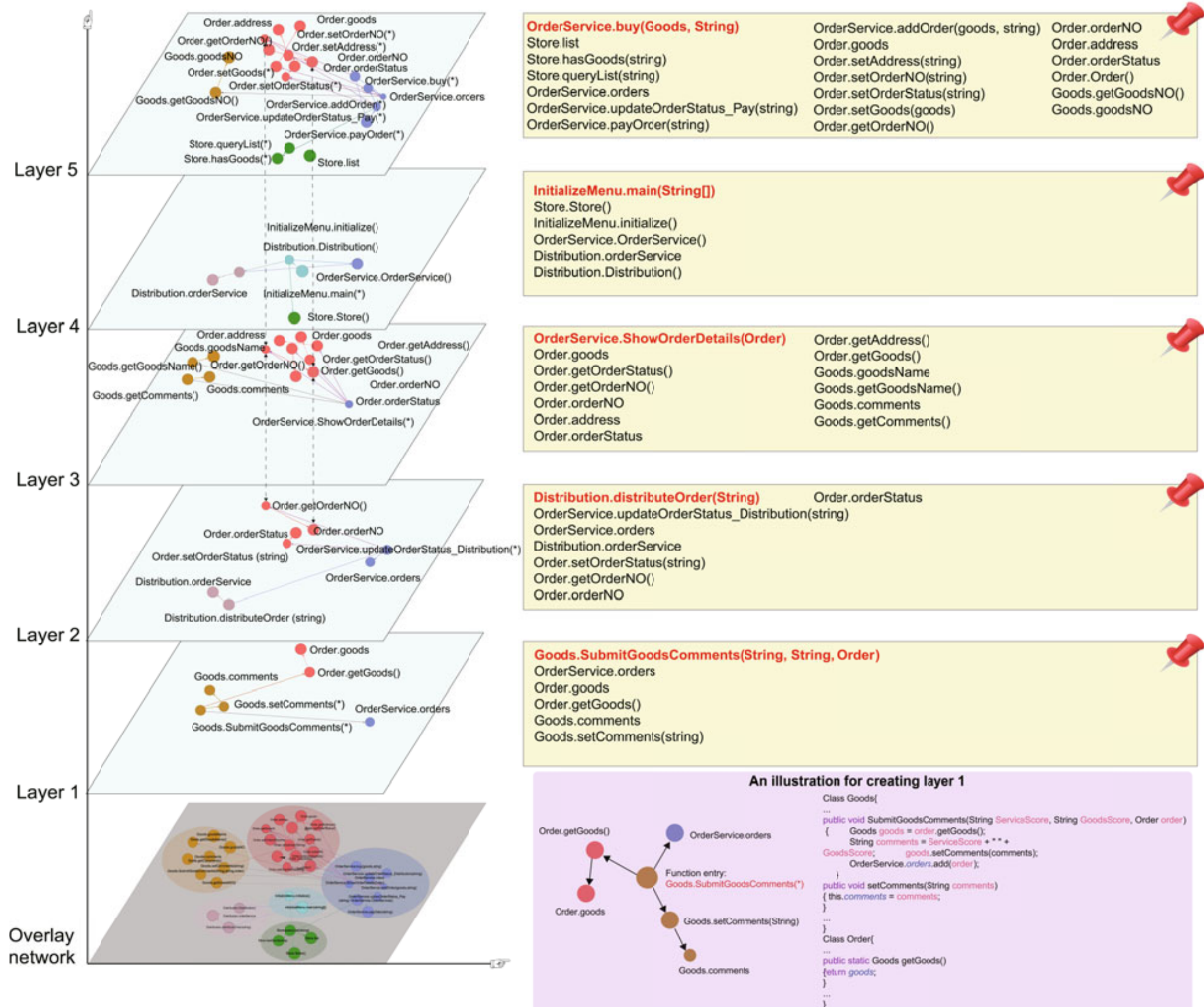

Fig. 3 The multi-layer dynamic execution network (MDEN) model for the sample code

probability, the complexity, and the consequence to the software if the nodes fail.

3.2 Probabilistic risk analysis model

In economics, PRA defines risk $\mathcal{R}$ as the product of threat $\mathcal{T}$, complexity $\mathcal{V}$, and consequence $\mathcal{C}$, i.e., $\mathcal{R}=\mathcal{T}\times\mathcal{V}\times\mathcal{C}$ (Al Mannai and Lewis, 2008). $\mathcal{T}$ is the probability of a component or asset being attacked or stressed, $\mathcal{V}$ is the failure probability of a component or asset under attack, and $\mathcal{C}$ is the financial or fatality consequence if a failure occurs. The PRA model provides useful means for identifying potentially troublesome classes that require higher priority and effort during the integration test process. We define the heuristic risk indices of classes as a combination of three factors: likelihood of being executed, malfunction probability, and failure consequence. A formalized definition of the risk factor is as follows: $\mathcal{T}(C_i)$ is the probability of code within class C_i being executed dynamically; $\mathcal{V}(C_i)$ is the complexity of class C_i; $\mathcal{C}(C_i)$ is the failure consequence of class C_i, i.e., the expected damage to the software caused by class C_i. Then, $\mathcal{R}(C_i)=\mathcal{T}(C_i)\times\mathcal{V}(C_i)\times\mathcal{C}(C_i)$.

The higher the risk index of a class, the more error-prone the class is; thus, more severe software damage is caused when it fails. If we assign higher priorities to critical classes with higher risk indices, the software reliability and test efficiency are also improved.

1. Threat $\mathcal{T}(C_k)$

In any function profile $G_i \in \mathcal{G}$, let $P_i =$

$\{p_1^i, p_2^i, ..., p_{|P_i|}^i\}$ be the execution trace set from the entry to the exit methods, function $g_i(m_t)$ the number of paths passing through method m_t, and $f(s_i)$ the execution probability of scenario s_i. Then, the execution probability of method m_t in G_i is equal to the ratio of $g_i(m_t)$ to the total number of execution paths in set P_i; $f(s_i)$ can be estimated as the frequency at which scenario s_i is covered by the test cases in the test case suite. Furthermore, the execution probability of class C_k is defined as the average execution probability of all methods belonging to the class in different scenarios:

$$\mathcal{T}_k(m_t) = \frac{\sum_{i=1}^{|V_{\text{et}}|} \varphi_i(m_t) \cdot f(s_i)}{\sum_{t=1}^{\text{NM}} \sum_{i=1}^{|V_{\text{et}}|} \varphi_i(m_t) \cdot f(s_i)}, \tag{3}$$

$$\varphi_i(m_t) = g_i(m_t)/|P_i|, \tag{4}$$

$$\mathcal{T}(C_k) = \frac{1}{\text{NM}_k} \sum_{t=1}^{N_k} \mathcal{T}_k(m_t). \tag{5}$$

2. Complexity $\mathcal{V}(C_k)$

We analyze the complexity of the class from two perspectives: complexity within the class, and complexity caused by the dynamic coupling relationships between classes. Dynamic coupling relationships denote the activities of the invocations between connected class pairs. Let the complexity $\mathcal{V}(C_k)$ of class C_k be the average failure rates of all methods defined in the class itself. For any method m_t in the software, the failure of the method may be caused by its own complexity, or by exceptions returned from other dependent methods. In other words, in a function profile, the failures of all nodes that are reachable by method m_t can lead to an exception for method m_t. Let $Q_t = \{m_1, m_2, \cdots, m_{|Q_t|}\}$ be the method set of directly or indirectly reachable methods m_t, S the sample space of failure reasons, and $\text{EA} = \{A_{m_1}, A_{m_2}, \cdots, A_{m_{|Q_t|}}, A_{m_t}\}$ a complete event group, where event A_{m_i} represents the failure of method $m_i \in Q_t$ at some point, which is caused by its own complexity. Then, all events in EA are independent of one another. Moreover, as all methods are executed in order based on the invocations between them, no event pairs occur simultaneously in a scenario, i.e.,

(1) $A_{m_i} \cap A_{m_j} = \varnothing$, $m_i, m_j \in Q_t, i \neq j$;

(2) $A_{m_1} \cup A_{m_2} \cup \cdots \cup A_{m_{|Q_t|}} \cup A_{m_t} = S$.

According to Bayes' total probability formula (Walters and Ludwig, 1994), the complexity of method m_t can be calculated by

$$\mu(m_t) = \sum_{i=1}^{|Q_t|} \mu(m_t|A_{m_i})\mu(A_{m_i}) + \mu(m_t|A_{m_t})\mu(A_{m_t}), \tag{6}$$

where prior probability $\mu(A_{m_i})$ represents the complexity of method m_i due to its own complexity, and posteriori probability $\mu(m_t|A_{m_i})$ denotes the complexity of method m_t introduced by the exception of method m_i. Thus, $\mu(m_t|A_{m_t}) = 1$. We discuss the definitions of $\mu(A_{m_i})$ and $\mu(m_t|A_{m_i})$ below:

(1) The Halstead model (Lipow, 1982) is adopted to measure the failure rate of method m_k, which is defined as

$$\begin{aligned}\mu(A_{m_k}) &= \frac{\mathcal{B}_k}{\sum_{i=1}^{|\text{NM}|} \mathcal{B}_i} = \frac{\text{VL}_k/3000}{\sum_{i=1}^{|\text{NM}|} (\text{VL}_i/3000)} \\ &= \frac{\mathcal{N}_k \log_2(\eta_k + \varphi_k)}{\sum_{i=1}^{|\text{NM}|} \mathcal{N}_i \log_2(\eta_i + \varphi_i)},\end{aligned} \tag{7}$$

where $\mathcal{B}_k$ is the number of faults in method m_k, VL_k is the code volume of method m_k, $\mathcal{N}_i$ is the total usage of all operators and operands appearing in the implementation, and η_i and φ_k represent the numbers of unique operators and operands in the code, respectively. Note that in our study, all constants and variables defined in classes are considered operands, and operators consist of arithmetic, logical, and relational operational symbols.

(2) Suppose that $p_{ti} = m_t \rightarrow m_1 \rightarrow m_2 \rightarrow \cdots \rightarrow m_{l_{ti}} \rightarrow m_i$ is the shortest path from method m_t to method m_i in static software topology, and that l_{ti} denotes the number of nodes passed by path p_{ti} (except methods m_t and m_i); as such, posteriori probability $\mu(m_t|A_{m_i})$ is given by

$$\mu(m_t|A_{m_i}) = \tau(m_t, m_1)\tau(m_1, m_2) \cdots \tau(m_{l_{ti}}, m_i). \tag{8}$$

According to Eq. (8), the complexity of class C_k satisfies

$$\mathcal{V}(C_k) = \frac{1}{\text{NM}_k} \sum_{i=1}^{N_k} \mu(m_i). \tag{9}$$

We adopt Floyd (1962)'s algorithm to search for all shortest paths between all method pairs, and the time complexity of the entire process is $O(\text{NM}^3)$.

3. Consequence $\mathcal{C}(C_k)$

Software execution is equivalent to the exchange of information flows between methods (Henry and Kafura, 1981). Under normal working conditions, we denote the total information flows transmitted in the network layer by $G_t = (V_t, E_t)$, corresponding to scenario s_t as $\mathcal{E}(G_t)$. More precisely, $\mathcal{E}(G_t)$ is the sum of communications between methods in G_t. Let $\upsilon_t(m_i, m_j)$ be the number of paths passing the directed edge $\langle m_i, m_j \rangle$ in execution trace set $P_t = \{p_1^t, p_2^t, ..., p_{|P_t|}^t\}$, where $m_i, m_j \in V_t$ and $\langle m_i, m_j \rangle \in E_t$. Then, we have

$$\mathcal{E}(G_t) = \sum_{i=1}^{\mathrm{NM}} \sum_{j=1}^{\mathrm{NM}} \upsilon_t(m_i, m_j). \tag{10}$$

According to the NASA-STD-8719.13A standard (Goseva-Popstojanova et al., 2003), risk severity considers the worst-case consequence of a failure determined by the degree of software damage and mission loss that can ultimately occur (NASA, 1999). In our study, we suppose that if method m_k fails, the faults will be propagated via invocation relationships to the other methods in network layer G_t. In other words, the failure of method m_k leads to the blocking of information flows; therefore, the loss of flows can be treated as the failure consequence of method m_k in this scenario. Suppose that $G_t^k = (V_t^k, E_t^k)$ is the failure network caused by method m_k in scenario s_t, obtained by removing m_k as well as all nodes and edges that can help reach m_k from network G_t. Clearly, in scenario s_t, $\mathcal{C}_t(m_k)$ is the difference in information flows between networks G_t and G_t^k. Let $\mathcal{E}(G_t^k)$ be the residual information flows in failure network G_t^k, $P_t^k = \{p_1^{kt}, p_2^{kt}, ..., p_{|P_t^k|}^{kt}\}$ the execution trace set working well in G_t^k, and $\upsilon_t^k(m_i, m_j)$ the number of paths passing through the directed edge $\langle m_i, m_j \rangle$ in set P_t^k, where $m_i, m_j \in V_t^k$ and $\langle m_i, m_j \rangle \in E_t^k$. As shown in Eq. (11), the failure consequence of class C_k can be defined as the average failure consequence of all methods belonging to the class in different scenarios:

$$\begin{aligned}\mathcal{C}(C_k) &= \frac{1}{\mathrm{NM}_k} \sum_{t=1}^{\mathrm{NM}_k} \mathcal{C}(m_t) = \frac{1}{\mathrm{NM}_k} \sum_{t=1}^{\mathrm{NM}_k} \sum_{i=1}^{|V_{\mathrm{et}}|} \mathcal{C}_i(m_t) \\ &= \frac{1}{\mathrm{NM}_k} \sum_{t=1}^{\mathrm{NM}_k} \sum_{i=1}^{|V_{\mathrm{et}}|} \left(\mathcal{E}(G_i) - \mathcal{E}(G_i^t)\right) / \mathcal{E}(G_i),\end{aligned} \tag{11}$$

where

$$\mathcal{E}(G_i^t) = \sum_{i=1}^{\mathrm{NM}} \sum_{j=1}^{\mathrm{NM}} \upsilon_t^k(m_i, m_j). \tag{12}$$

We employ the MDEN model in Fig. 3 as an example to further illustrate the risk analysis. Table 1 lists the statistics of class risk indices in sample code. From the threat perspective, class OrderService containing nine entities is executed in all five scenarios. In particular, attribute OrderService.orders appears in three network layers, layers 1, 2, and 5, and there are seven execution traces passing through method OrderService.addOrder(Goods, String) in layer 5. As a result, the execution probability of class OrderService is relatively high. For the complexity, class OrderService contains 10 operators and five operands used by other classes a total of 156 times. In particular, because it depends on a relatively complex class Order, the incoming complexity results in an increase in its own complexity. For failure consequence, methods Order.getOrderNO(), Order.orderNO, and Order.orderStatus appear in three scenarios. Of these, the failure of Order.getOrderNO leads to a 67%, 71%, and 35% loss in the information flows in layers 2, 3, and 5, respectively. A comprehensive comparison indicates that class Order ranks first in failure consequence. As the threat, complexity, and failure consequences of class OrderService are all relatively high in the software, the class should be assigned a higher test risk index.

In our study, the PRA model is regarded as a flexible framework for identifying potentially troublesome classes that require higher priority and effort during the integration test process. The proposed approach is not an exclusive one for quantifying three indexes of the PRA model. Specifically, the complexity index $\mathcal{V}(C_k)$ of the model can be replaced with other effective metrics to quantify the complexity of each class, such as cyclomatic complexity (Bang et al., 2015; Weyuker, 1988).

Table 1 Statistics of class risk indices in sample code

Class	$\mathcal{T}(C_i)$	$\mathcal{V}(C_i)$	$\mathcal{C}(C_i)$	$\mathcal{R}(C_i)$
Store	0.0315	0.1310	0.0933	0.0107
InitializeMenu	0.1205	0.1000	0.0810	0.0272
OrderService	0.3131	0.5341	0.1417	0.6601
Distribution	0.1205	0.0244	0.2644	0.0217
Order	0.2629	0.1104	0.2671	0.2160
Goods	0.1515	0.1000	0.1524	0.0643

3.3 Algorithm for generating CITOs

To reduce software risk and improve reliability, the test effort should be focused on classes with higher risk indices. In this subsection, we propose a strategy for generating CITOs, which assigns higher priorities to higher-risk classes while guaranteeing that the total test stub complexity is minimized. In this manner, we can detect and correct bugs as early as possible, reduce expenses, and improve software quality.

3.3.1 Complexity measurements for test stubs

A stub is a placeholder that implements the necessary partial functionality of a class for its compilation and integration (Sharma and Sibal, 2013). Suppose that class C_i is dependent on C_j. Then, we should simulate an object of C_j for class C_i in a scenario in which C_i is being tested when C_j has not yet been tested. The simulated object is considered a stub and written as $\mathrm{Stub}(C_i, C_j)$, which is composed of the attributes and methods invoked by class C_i. However, stub construction is treated as an expensive and error-prone operation. Moreover, the costs of stub construction can be evaluated based on the following measurements proposed by Briand et al. (2002) to control the total test efforts:

$$\begin{cases} \mathrm{SCplx}(C_i, C_j) \\ = \sqrt{\alpha \cdot \overline{\mathrm{ZA}(C_i, C_j)}^2 + \beta \cdot \overline{\mathrm{ZM}(C_i, C_j)}^2}, \\ \overline{\mathrm{ZA}(C_i, C_j)} = \frac{\mathrm{ZA}(C_i, C_j)}{\mathrm{ZA}_{\max} - \mathrm{ZA}_{\min}}, \\ \overline{\mathrm{ZM}(C_i, C_j)} = \frac{\mathrm{ZM}(C_i, C_j)}{\mathrm{ZM}_{\max} - \mathrm{ZM}_{\min}}, \end{cases} \tag{13}$$

where $C_i, C_j \in \mathcal{S}$, $i \neq j$, $\alpha + \beta = 1$, $\mathrm{SCplx}(C_i, C_j)$ represents the complexity of the test stub used to simulate the behaviors of C_j relative to C_i's dependencies, $\mathrm{ZA}(C_i, C_j)$ denotes the number of attributes belonging to class C_j and accessed by class C_i, $\mathrm{ZM}(C_i, C_j)$ is the number of methods defined in class C_j invoked by class C_i, and $\mathrm{ZA}_{\max}$, $\mathrm{ZA}_{\min}$, $\mathrm{ZM}_{\max}$, and $\mathrm{ZM}_{\min}$ represent the maximum and minimum values of $\mathrm{ZA}(C_i, C_j)$ and $\mathrm{ZM}(C_i, C_j)$, respectively. In other words, $\overline{\mathrm{ZA}(C_i, C_j)}$ and $\overline{\mathrm{ZM}(C_i, C_j)}$ are the normalization values of $\mathrm{ZA}(C_i, C_j)$ and $\mathrm{ZM}(C_i, C_j)$, respectively.

3.3.2 Integration test order

The goal of generating integration test order is to ensure that when one class is tested, most classes that depend on it have also been tested. The reverse ordering of classes based on the direction of relationship between classes can help reduce the number of test stubs and guarantee the completeness of integration testing. All cycles composed of class relationships can be found according to the topology of the class-level network. To obtain an acyclic dependency network, we should break the cycles by removing edges from the software. Once a dependency has been broken, a test stub should be constructed (Huang and Lyu, 2005).

Definition 2 (Class-level dependency network) By merging the network layers in $|V_{\mathrm{et}}|$ scenarios, a complete method-level network $G^{\mathrm{m}} = (V^{\mathrm{m}}, E^{\mathrm{m}})$ can be obtained, which satisfies $V^{\mathrm{m}} = V_1 \cup V_2 \cup ... \cup V_{|V_{\mathrm{et}}|}$. Furthermore, if we shrink all entities in a class into one node and merge all relationships between a class pair into one edge, the class-level dependency network $G^{\mathrm{c}} = (V^{\mathrm{c}}, E^{\mathrm{c}})$ is formed.

Definition 3 (Dependent path) Suppose that SV is a collection of nodes with an in-degree of zero and an out-degree greater than zero, and that EV is a collection of nodes with an in-degree greater than zero and an out-degree of zero. For any class $C_i \in$ SV, class $C_j \in$ EV, $i \in \{1, 2, \cdots, |\mathrm{SV}|\}$, and $j \in \{1, 2, \cdots, |\mathrm{EV}|\}$, if there is a path p_{ij} from C_i to C_j in the software, p_{ij} is defined as a dependent path between C_i and C_j.

Definition 4 (Dependent depth) Suppose that class $C_1 \in$ SV, class $C_n \in$ EV, and dependent path $p_{ij} = C_1 \to C_2 \to \cdots \to C_k \to \cdots \to C_{n-1} \to C_n$; then, the dependent depth of C_k satisfies $D_k = |n - k| + 1$.

Definition 5 (Maximum dependent depth) Suppose that there are q dependent paths passing through class C_k, and that D_{ki} is the dependence depth of C_k along the ith dependent path, $k \in \{1, 2, \cdots, |V_{\mathrm{c}}|\}$, and $i \in \{1, 2, \cdots, q\}$; then, the maximum dependence depth satisfies $D^k_{\max} = \max\{D_{k1}, D_{k2}, \cdots, D_{kq}\}$.

Let $\Psi_{ij} = \{\mathrm{cp}_1, \mathrm{cp}_2, \cdots, \mathrm{cp}_{|\Psi_{ij}|}\}$ be the cycle set passing directed edge $\langle C_i, C_j \rangle$ in class-level dependency network G^{c}. If edge $\langle C_i, C_j \rangle$ is removed from $\mathrm{cp}_t \in \Psi_{ij}$, we obtain a directed path $p^t_{ji} = C_j \to C_k \to \cdots \to C_r \to C_i$. Suppose that O_t is the test order devised by sorting classes along path p^t_{ij} according to their topologies, and that O'_t is the test order generated by sorting classes in descending order in p^t_{ij} based on their risk indices. When removing

$\langle C_i, C_j\rangle$ from cp_t, $\mathrm{SCplx}(C_i, C_j)$ can be considered the test cost, and the degree of high-risk classes being integrated preferentially, denoted by PR_t, can be treated as a test benefit. As a result, using cost–benefit analysis, the benefit rate of removing edge $\langle C_i, C_j\rangle$ is quantified as follows:

$$\mathrm{Bf}_{ij}^t = \frac{\mathrm{PR}_t}{\mathrm{SCplx}\,(C_i, C_j)}. \tag{14}$$

Here, PR_t is equal to the ratio of the degree of "high-risk class being tested first" in order O_t to that of order O'_t, calculated by

$$\mathrm{PR}_t = \sum_{k=1}^{r_t} \frac{\dfrac{2(r_t-k+1)\mathcal{R}_k^{O_t}}{r_t(r_t+1)}}{\sum\limits_{n=1}^{r_t}\dfrac{2(r_t-n+1)\mathcal{R}_n^{O'_t}}{r_t(r_t+1)}}, \tag{15}$$

where r_t represents the number of classes in cycle cp_t, $\mathcal{R}_k^{O_t}$ is the risk index of the class ranked k^{th} in order O_t, and $\mathcal{R}_n^{O'_t}$ denotes the risk index of the class ranked n^{th} in order O'_t.

All cycles must exist in the SCCs of the directed graph (Kung et al., 1995). For this reason, before performing cycle-breaking operations, we should weigh each edge within the SCCs detected in the class-level dependency network. Let weight W_{ij} of edge $\langle C_i, C_j\rangle$ be the maximum benefit value of being removed from all cycles passing through it. Thus,

$$W_{ij} = \max\{\mathrm{Bf}_{ij}^1, \mathrm{Bf}_{ij}^2, ..., \mathrm{Bf}_{ij}^{|\Psi ij|}\}, \tag{16}$$

where W_{ij} denotes the maximization of the cycle-breaking operation benefit rates.

Cycle-breaking operations determine the complexity of constructing test stubs (test cost) and the priority of high-risk classes to be tested (test efficiency). In our strategy, we encode two criteria in cycle-breaking operations to balance test cost and test efficiency using the benefit rate Bf_{ij}^t defined in Eq. (14): providing higher priorities to critical classes with higher risk indices, and minimizing the total complexity of the test stubs.

The steps for the strategy in generating CITOs are as follows:

(1) Merge all network layers in the MDEN model $\mathcal{G}$; then, obtain class-level dependency network $G^{\mathrm{c}} = (V^{\mathrm{c}}, E^{\mathrm{c}})$.

(2) Use Tarjan's algorithm to traverse all nodes in network G^{c}; then, a collection of the SCC can be found, where $\mathrm{SCC} = \{\mathrm{sc}_1, \mathrm{sc}_2, \cdots, \mathrm{sc}_{|\mathrm{SCC}|}\}$. According to the definition of SCC, all cycles must exist in the SCC of the directed networks.

(3) For each $\mathrm{sc}_i \in \mathrm{SCC}$, we repeat steps i and ii:

i. Find and record all cycles in sc_i, and assign a weight to each edge of each cycle according to Eq. (16).

ii. By adhering to the following principles, we remove edges from the network to break cycles:

(a) If there is only one edge $\langle C_i, C_j\rangle$ with the greatest weight in the network, $\langle C_i, C_j\rangle$ is deleted.

(b) If there is more than one edge with the greatest weight, we delete the edge $\langle C_i, C_j\rangle$, whose corresponding $\mathrm{Stub}(C_i, C_j)$ has lower complexity.

(c) If there is more than one edge with the greatest weight, and the test stub complexity values corresponding to all edges are equal, the edge whose starting node has the highest risk index is removed.

(d) When there are no more cycles in sc_i, stop the operation of removing edges.

(4) Traverse all nodes in the network and calculate their maximum dependent depths. The nodes with the same $D_{\max}^k$ are sorted by their risk indices in descending order, while the other nodes are sorted by their $D_{\max}^k$ values in ascending order. Finally, we obtain the inner-class integration test order O_{test}.

Take $\mathrm{SCC}_1 = \{\mathrm{OrderService}, \mathrm{Goods}, \mathrm{Order}\}$ as an example, which contains three cycles, i.e., OrderService → Goods → OrderService, Order → Goods → Order, and OrderService → Order → Goods → OrderService. From the statistics in Table 2, we can see that removing edge $\langle \mathrm{Order}, \mathrm{Goods}\rangle$ requires the minimal effort to boost the priority of class Order with a higher index, thereby obtaining larger gains. Compared with Stub(OrderService, Goods), constructing Stub(Goods, OrderService) incurs lower costs. However, we ultimately eliminate edge $\langle \mathrm{OrderService}, \mathrm{Order}\rangle$ to guarantee that higher-risk class OrderService is tested earlier. As shown in Fig. S2, by topologically sorting classes in the acyclic network, test order Order → OrderService → Goods → Distribution → Store → InitializeMenu is naturally devised.

Table 2 SCC statistics in the example software

Edge$\langle C_i, C_j\rangle$	$\omega(C_i, C_j)$	SCplx(C_i, C_j)	Bf	NM	NA
⟨Order, Goods⟩	2	0.02	0.5394	1	0
⟨OrderService, Goods⟩	1	0.18	0.1832	3	0
⟨Goods, OrderService⟩	2	0.02	0.1607	1	0
⟨OrderService, Order⟩	1	0.32	0.1031	4	0
⟨Goods, Order⟩	1	0.24	0.0136	1	0

4 Evaluation model

4.1 Strategy for optimizing the allocation of testing resources

Focusing the testing on critical classes by optimally allocating testing resources can be considered a mitigation strategy for software risk (Huang and Lyu, 2005). As test costs rise, the failure probability of classes is reduced accordingly. Intuitively, if a component has been comprehensively tested, the risk associated with its use should be less than the one that has not been well tested (Frankl and Weyuker, 2000). To improve the efficiency of integration testing, we propose a scheme for allocating testing resources to classes based on their test benefit rates in risk reductions; i.e., greater test efforts should be focused on high-risk classes while less effort should be invested in low-risk ones. However, software failure probability can be eliminated only by an infinitely long testing; thus, the exponential function is applicable in representing the software reliability growth model. In our study, we extend the fault-discovery model (Monden et al., 2013) to express the cumulative magnitude of software risk reduction after the first $\hat{N}$ classes in the test order have been tested during the integration process:

$$\begin{cases}\mathcal{R}'_{\hat{N}}=\sum\limits_{i=1}^{\hat{N}}\mathcal{T}(C_i)\mathcal{C}(C_i)\mathcal{V}(C_i)(1-\mathrm{e}^{-\theta_i\mathrm{DA}_i}),\\ \theta_i=b_0/S_i,\ \mathrm{DA}_i=\phi_i\mathrm{DA}_i^{\max},\end{cases}\tag{17}$$

where $\hat{N}$ is the total number of classes that have been tested, θ_i is the ease coefficient of detecting faults in class C_i per unit effort, b_0 is a constant, S_i is the size of class C_i, DA_i is the test resource allocated to class C_i, $\mathrm{DA}_i^{\max}$ is the test resource demand for fully testing class C_i, and ϕ_i is the ratio of DA_i to $\mathrm{DA}_i^{\max}$.

We suppose that the maximum test cost $\mathrm{DA}_i^{\max}$ of class C_i is proportional to the number of its entities and the complexity of test stubs constructed for class C_i (i.e., the larger the class size, the higher the test cost). Let μ represent the cost of testing each entity, including the time, test cases, and manpower expended on it; then we obtain

$$\mathrm{DA}_i^{\max}=\mu(|C_i|+\mathrm{Stub}_i),\tag{18}$$

where Stub_i is the number of entities in stubs simulated for the use of class C_i. If there is no need to construct stubs for class C_i, $\mathrm{Stub}_i=0$. Furthermore, after the first $\hat{N}$ classes of the test order have been tested, the ratio of the magnitude of the remaining risks in the software to the total risk satisfies

$$\varpi_{\hat{N}}=1-\frac{\mathcal{R}'_{\hat{N}}}{\mathcal{R}_{|V^{\mathrm{c}}|}},\tag{19}$$

where $\mathcal{R}_{|V^{\mathrm{c}}|}$ represents the total risk of all classes in the software, and $\mathcal{R}_{|V^{\mathrm{c}}|}=\sum\limits_{i=1}^{|V^{\mathrm{c}}|}\mathcal{T}(C_i)\mathcal{C}(C_i)\mathcal{V}(C_i)$.

If adequate test resources are allowed for each class, the reduction in the total risk to the software after testing all classes in order can be quantified as

$$\mathcal{R}'_{|V^{\mathrm{c}}|}=\mathcal{R}_{|V^{\mathrm{c}}|}\cdot\left(1-\exp\left(-\frac{b_0}{\sum\limits_{t=1}^{|V^{\mathrm{c}}|}S_t\mu\left(\sum\limits_{i=1}^{|V^{\mathrm{c}}|}S_i+\sum\limits_{k=1}^{N_{\mathrm{s}}}|\mathrm{Stub}_k|\right)}\right)\right),\tag{20}$$

where N_{s} denotes the total number of established test stubs. Consequently, we obtain a rough estimate of $b_0\cdot\mu$ as

$$b_0\cdot\mu=-\frac{\ln\left(1-\dfrac{\mathcal{R}'_{|V^{\mathrm{c}}|}}{\mathcal{R}_{|V^{\mathrm{c}}|}}\right)}{1+\sum\limits_{k=1}^{N_{\mathrm{s}}}|\mathrm{Stub}_k|\Big/\sum\limits_{i=1}^{|V^{\mathrm{c}}|}|S_i|}.\tag{21}$$

In our study, the value of $\mathcal{R}'_{|V^{\mathrm{c}}|}\big/\mathcal{R}_{|V^{\mathrm{c}}|}$ is set to 0.99. In other words, we assume that software risk decreases by 99% after all classes have been fully tested.

Let Φ be the reduction in the overall software risk when integration testing is complete. Then, we have

$$\Phi=\mathcal{R}'_{|V^{\mathrm{c}}|}.\tag{22}$$

We need to optimize the allocation of testing

resources to each class, which satisfies

$$\begin{cases} \max\{\Phi\} \\ = \max\{\mathcal{R}'_{|V^c|}\} \\ = \max\left\{\sum_{i=1}^{|V^c|} \mathcal{T}(C_i)\mathcal{C}(C_i)\mathcal{V}(C_i)\left(1-\mathrm{e}^{-\theta_i \mathrm{DA}_i}\right)\right\}, \\ \sum_{i=0}^{|V^c|} \mathrm{DA}_i = B_0, 0 \le \mathrm{DA}_k \le \mathrm{DA}_k^{\max}, \end{cases} \tag{23}$$

where B_0 represents the total test cost allocated to all classes in the software. To maximize the test benefit rates, the following two principles should be observed upon allocation:

(1) Consider the expected risk reduction created by each class as a benefit. Moreover, treat the required test efforts as costs.

(2) With limited resources, try to satisfy the needs of classes with higher benefit rates while testing the others less comprehensively.

Then, the test benefit rate of class C_k can be quantified by the ratio of benefit to cost:

$$\gamma_k = \frac{\mathcal{R}(C_k)}{\mathrm{DA}_k^{\max}} = \frac{\mathcal{R}(C_k)}{\mu(|C_k| + \mathrm{Stub}_k)}. \tag{24}$$

Assume O_γ denotes the sequence of classes in descending order of their test benefit rates γ_k. To reduce software risk as quickly as possible, we allocate the maximum required resource $\mathrm{DA}_k^{\max}$ to the top-ranked classes following the order in O_γ, until the remaining cost is lower than the maximum needed resource $\mathrm{DA}_i^{\max}$ of the given class C_i. Then, as shown in Eq. (25), the residual budgets are equally assigned to classes whose test benefit rates are lower than that of class C_i:

$$\mathrm{DA}_i = \begin{cases} \mathrm{DA}_i^{\max}, \text{ if } B' \ge \mathrm{DA}_i^{\max}, \\ \dfrac{B'}{|V^c| - i + 1}, \text{ otherwise}, \end{cases} \tag{25}$$

where B' is the residual budget after testing the top $i-1$ classes in order O_γ. Let i be the ranking of classes in O_γ; then, we have

$$B' = B_0 - \sum_{t=0}^{i-1} \mathrm{DA}_{i-1}. \tag{26}$$

4.2 Rate of risk reduction

Once the top $i-1$ classes of order O_{test} have been tested, the ratio of residual software risk is

$$\varpi_i = 1 - \frac{\mathcal{R}'_i}{\mathcal{R}_{|V^c|}}. \tag{27}$$

According to the mapping relationships between the number of classes having been tested $i \in V^c$ and the residual software risk ϖ_i, the rate of reduction in software risk can be calculated using the least squares method (LSM):

$$R_{\varpi_i} = \frac{\sum_{t=1}^{i} t^2 \sum_{t=1}^{i} \varpi_i^2 - \sum_{t=1}^{i} t \sum_{t=1}^{i} \varpi_i}{i \sum_{t=1}^{i} t^2 - \left(\sum_{t=1}^{i} t\right)^2}. \tag{28}$$

Under the condition of controlling the total complexity of the test stubs, a more efficient integration test order leads to a lower residual amount and a higher rate of reduction in software risk. As a result, both Φ_i and R_{Φ_i} can be considered evaluation criteria for integration test orders.

4.3 Fault detection efficiency of integration test order

The cost-cognizant metric, average percentage of faults detected per cost ($\mathrm{APFD_c}$), has been proposed to evaluate the effectiveness of test case prioritization techniques in detecting faults (Goseva-Popstojanova et al., 2003). To assess the fault-detection efficiency of the integration test orders obtained by our strategy, we redefine $\mathrm{APFD_c}$ based on the risk severity of classes and stubbing efforts:

$$\mathrm{APFD_r} = \frac{\sum_i^{|\mathcal{F}|}\left(\mathrm{sf}_i\left(\sum_{j=\mathrm{TF}_i}^{|V^c|} \mathrm{ct}_j - \frac{1}{2}\mathrm{ct}_{\mathrm{TF}_i}\right)\right)}{\sum_{i=1}^{|V^c|} \mathrm{ct}_i \sum_{i=1}^{|\mathcal{F}|} \mathrm{sf}_i}. \tag{29}$$

$\mathcal{F} = \{f_1, f_2, \cdots, f_{|\mathcal{F}|}\}$ represents the set of faults that exist in the software, where $|\mathcal{F}|$ is the total number of faults in $\mathcal{F}$. sf_i is the severity of any fault $f_i \in \mathcal{F}$. If class $C_t \in V^c$ contains fault f_i, sf_i is equal to the risk index of C_t. ct_i is the test cost of integrating the ith class into the test order. Suppose that ct_i is the complexity of the test stub established to integrate the ith class. If the ith class can be integrated directly, $\mathrm{ct}_i = 0$. TF_i is the ranking of the class containing fault $f_i \in \mathcal{F}$ in the test order.

Clearly, the value of $\mathrm{APFD_r}$ ranges from 0 to 1. For an integration test order, a higher $\mathrm{APFD_r}$ value corresponds to a more efficient detection of severe faults.

5 ITOsolution tool

The proposed methodology has been implemented as a tool called ITOsolution (runnable JAR file: https://github.com/FanyiMeng-NEU/ITOsolution-Runnable-Jar), which can automatically provide integration test order solutions using cost–benefit analysis. ITOsolution integrates DependencyFinder (http://depfind.sourceforge.net/) to analyze the source code of Java projects and employs the Prefuse visualization toolkit (http://prefuse.org/) to display the multi-layer dynamic execution network model mapped from software. Fig. S3 shows the main features (Demo: https://www.bilibili.com/video/BV1Ty4y1L7KP) of ITOsolution. For a software program under test, ITOsolution can construct its corresponding MDEN model (Fig. S3a), derive a risk distribution diagram (obtained based on the PRA model) (Fig. S3b), provide the intermediate results of cycle-breaking operations (Fig. S3c), and give the reduction curve during integration (Fig. S3d):

1. ITOsolution requires byte-code files and source code files of the software under test as inputs. The byte-code files allow the abstract syntax tree of code to be obtained. Scanning the source code files, the operators and operands of source code can be extracted to evaluate their complexity based on Halstead measurements. Furthermore, as shown in Fig. S3a, the MDEN model is generated automatically.

2. Fig. S3b shows the tooltip of the risk distribution diagram of the example code.

3. By removing the edges, which creates more integration test benefits, the tool eliminates all the cycles from the software, as shown in Fig. S3c. Then the established test stubs are listed in the table (provided by the tool). By default, the coefficients α and β defined in Eq. (13) are set to be 0.5. This means that the testers need to make the same effort to emulate the attributes and methods in test stubs. However, there is an additional function to allow users to adjust the parameters according to their preferences. More importantly, users can save the raw data such as the class diagram, class dependency matrix, attribute coupling matrix, and method coupling matrix, for further replication and research purposes.

4. After the CITO solution is devised, the tool automatically generates the software risk reduction curve due to the integration steps (as shown in Fig. S3d). In this manner, the user can easily assess the results.

6 Experimental analysis and discussion

6.1 Research questions and experimental design

The probabilistic risk analysis for classes determines their integration test priorities. In the evaluation, we should first analyze the risk distribution of all classes in open-source software, to understand how effective the proposed risk analysis model is in evaluating the threat, failure probability, and failure consequence of each class (RQ_1). Furthermore, as discussed in Section 2, several algorithms have been proposed to generate CITOs with minimized stubbing efforts. Therefore, we should consider them as baseline approaches to evaluate the effectiveness of our approach (RQ_2). In addition, because our approach is proposed based on cost–benefit analysis, we should verify whether the scheme for allocation of testing resources is meaningful during the testing process (RQ_3). Under the condition of controlling test costs, a more efficient integration test order leads to a lower residual amount and a higher rate of reduction in software risk.

To address the above concerns, we design three research questions:

RQ_1: Does the proposed risk analysis model effectively identify critical classes for integration test?

RQ_2: Compared with other approaches, what are the advantages of the proposed strategy in generating CITO?

RQ_3: From the perspective of test costs and benefits, does the scheme for allocation of testing resources really make sense?

We selected subjects according to the following two criteria:

1. To evaluate the effectiveness of our strategy, six projects, namely, DNS 1.2.0 (http://www.xbill.org/dnsjava/), ANT 1.9.4 (http://ant.apache.org/) BCEL 5.0 (http://commons.apache.org/proper/commons-bcel/), Jmeter 1.8.1 (http://jakarta.apache.org/jmeter), Xml-security 1.0.5D2 (http://xml.apache.org/security), and Joda-time 2.8.2 (http://github.com/JodaOrg/joda-time) were used as experiment subjects. They differed in size,

complexity, and application domain, which ensures, to some extent, the generalization of the conclusions we obtained in this study. Existing approaches (Briand et al., 2002, 2003; da Veiga Cabral et al., 2010; Jiang et al., 2011, 2021) also adopted the above six subjects, and provided the corresponding generated integration test orders. Following the related works to select these experiment subjects can facilitate discussion and comparison of the experiment results.

2. To evaluate the fault-detection efficiency of the proposed risk analysis model, we considered only the open-source projects Jmeter, Xml-security, and Joda-time as subjects, because they have publicly accessible issue trackers that allow us to identify real bugs, whereas the three other projects are closed-source ones that contain limited information for evaluation. The first two projects were obtained from the Software-artifact Infrastructure Repository (SIR) (http://sir.unl.edu/), and the third from its own repository.

Table 3 describes the demographics of these six subjects (raw data: https://github.com/FanyiMeng-NEU/Experimental-Objects); their corresponding class-level dependency networks are shown in Fig. S4. Combining Table 3 with Fig. S4, we provided the statistics of our experiment subjects. We can tell that the six selected subjects differed in size (the number of classes ranged from 25 to 285) and complexity (the number of cycles varied from 16 to 416 091). Thus, we could evaluate the algorithm from different perspectives including size, complexity, and cycle density.

To address RQ_1, we used the total risk indices of classes covered by test cases to guide the scheduling of their execution order, and then compared them with seven state-of-the-art test case prioritization techniques based on the measurement of severe-fault detection efficiency.

Table 3 Statistics of the experiment subjects

Software	$\lvert V_c \rvert$	$\lvert E_c \rvert$	N_c	N_f	Fault type	N_{ts}
Jmeter 1.8.1	285	709	105	11	Seeded	26
Xml-security 1.0.5D2	219	776	977	10	Seeded	18
Joda-time 2.8.2	156	713	5536	15	Real	133
DNS 1.2.0	61	276	16	–	–	–
ANT 1.9.4	25	83	654	–	–	–
BCEL 5.0	45	294	416 091	–	–	–

$\lvert V_c \rvert$ denotes the number of classes. $\lvert E_c \rvert$ denotes the number of edges between classes. N_c, N_f, and N_{ts} denote the total numbers of cycles, faults, and test cases, respectively

To address RQ_2, we compared the CITO results of the six software programs obtained by our algorithm with those generated by the approaches of Tai and Daniels (1999), Le Traon et al. (2000), Briand et al. (2002, 2003), Abdurazik and Offutt (2006), Jiang et al. (2011), and Assunção et al. (2014) from multiple perspectives. The above state-of-the-art CITO generation algorithms were considered as baselines, because we can use the same stubbing complexity metric $\mathrm{SCplx}(C_i, C_j)$ defined in Eq. (13) to evaluate their test costs. Such a metric is not applicable to the approach proposed by Jiang et al. (2021), which considered the inter-class indirect relationships caused by control coupling. All simulations were performed on a personal computer in the following hardware environment: 3.7 GHz CPU, 12 GB memory, and a 1 TB HDD. The software operating environment was Windows 8.1 and the compiler platform was Eclipse 4.5.0. Note that Tai and Daniels (1999), Le Traon et al. (2000), and Briand et al. (2003)'s strategies aim to minimize the number of test stubs, and that Briand et al. (2002), Abdurazik and Offutt (2006), Jiang et al. (2011), and Assunção et al. (2014)'s algorithms devise integration test orders to minimize the total test stub complexity.

6.2 Case studies for RQ_1

6.2.1 Risk distribution of the software

The proposed ITOsolution tool can extract coupling relationships between classes from an OO software program and automatically construct the MDEN model. Using this, we analyzed the source codes of Jmeter, Xml-security, and Joda-time. Furthermore, as shown in Fig. S5, the probability density distribution of risk indices in the three software programs were derived by risk analysis based on the PRA model. Table 4 lists the normalized statistical results of the risk indices, threat, complexity, and consequence of all classes. Here, $\mathcal{R}_{\max}$, $\mathcal{R}_{\min}$, $\mathcal{R}_{\mathrm{med}}$, $\mathcal{T}_{\max}$, $\mathcal{T}_{\min}$, $\mathcal{T}_{\mathrm{med}}$, $\mathcal{V}_{\max}$, $\mathcal{V}_{\min}$, $\mathcal{V}_{\mathrm{med}}$, $\mathcal{C}_{\max}$, $\mathcal{C}_{\min}$, and $\mathcal{C}_{\mathrm{med}}$ are the maximum, minimum, and median values of $\mathcal{R}(C_i)$, $\mathcal{T}(C_i)$, $\mathcal{V}(C_i)$, and $\mathcal{C}(C_i)$, respectively, and NL represents the number of dynamic execution network layers in the MDEN model mapped from the software.

From Fig. S5, we can see that few classes in the three software programs had high-risk indices. Taking the Jmeter software program as an example,

Table 4 Statistics of the risk factors

Factor	Jmeter	Xml-security	Joda-time
NL	52	124	333
$\mathcal{R}_{\max}$	0.1509	0.1091	0.1573
$\mathcal{R}_{\min}$	6.70×10^{-10}	1.50×10^{-8}	0.0017
$\mathcal{R}_{\text{med}}$	0.0033	0.0042	0.0065
$\mathcal{T}_{\max}$	0.0517	0.0277	0.1811
$\mathcal{T}_{\min}$	2.23×10^{-4}	3.64×10^{-4}	2.36×10^{-6}
$\mathcal{T}_{\text{med}}$	0.0035	0.0046	0.0064
$\mathcal{V}_{\max}$	0.0291	0.0258	0.0656
$\mathcal{V}_{\min}$	3.29×10^{-6}	1.82×10^{-5}	3.52×10^{-4}
$\mathcal{V}_{\text{med}}$	0.0031	0.0040	0.0057
$\mathcal{C}_{\max}$	0.0382	0.1140	0.1926
$\mathcal{C}_{\min}$	3.04×10^{-5}	7.14×10^{-5}	2.65×10^{-6}
$\mathcal{C}_{\text{med}}$	0.0032	0.0039	0.0045

the risk indices of its classes were distributed in the interval $[6.70\times10^{-10}, 0.1509]$. Of these, there were 19 classes whose risk indices were higher than 0.01, whereas the risk indices of 228 classes in the software were lower than 0.003, accounting for 80% of the total. Similar situations prevailed in the two other software programs. In other words, only a limited fraction of nodes had relatively high fault rates and the capability of error propagation; therefore, they should have been tested as early as possible.

Table S1 (see supplementary materials for Tables S1–S10) presents the statistical results of the top five and bottom five classes in the software ranked by the risk index. Metrics including out-degree K_{out}, in-degree K_{in}, message passing coupling MPC_{out} and MPC_{in}, code volume VL_i, number of faults $\mathcal{B}_i$, structural complexity WMC, and lines of code (LOC) were used to describe the testing importance of classes in the software. MPC_{in} represents the total number of times that the methods of class C_i were invoked by methods not belonging to class C_i, MPC_{out} represents the total number of dependencies of class C_i methods on methods not belonging to class C_i, and WMC is the sum of McCabe's cyclomatic complexity metric (McCabe, 1976) values of the methods in class C_i. The results showed that these metric values had a certain correlation with the threats, vulnerabilities, and fault consequences of classes in the OO software.

In the Jmeter software, class 258, org.apache.jmeter.visualizers.RunningSample, was ranked first by the risk index. It had the largest code volume and number of faults in software. Fourteen entities belonging to this class were in use by other classes for a total of 48 times, and this class was dependent on the entities of other classes 46 times. Class 258 was in the hub position of the topological structure, leading to the highest failure consequence. Accordingly, more test efforts should have been focused on class 258.

Class 210, org.apache.xml.security.utils.XMLUtils, in the software Xml-security, contained 47 entities, and its VL_i, $\mathcal{B}_i$, WMC, LOC, K_{in}, and MPC_{in} were all the highest of the software. Nonetheless, it had a lower risk index than class 66, which was passed through by more execution traces in all scenarios. As a result, class 66 ranked first by the risk index in software due to its higher execution probability and failure consequence.

Clearly, class 144, org.joda.time.format.PeriodFormatterBuilder, in the Joda-time software, used more operators and operands, and had relatively high code volume, structural complexity, and code size; hence, it was more error-prone compared with the other classes. Once it failed, on average, 36.7% of information flows would have been lost in 245 function profiles. Although it had a low frequency of execution, the product of the three risk factors had the maximum value in the software. Taking a comprehensive view of this class, it was assigned the greatest risk weight based on the PRA model.

The other top-rank classes were similar to the classes listed in Table S1, and had relatively high K_{out}, K_{in}, MPC_{out}, MPC_{in}, VL_i, $\mathcal{B}_i$, WMC, and LOC. Accordingly, they also had higher failure probabilities and failure consequences. As a result, they should have been assigned higher test priorities. Of the low-rank classes, the K_{in} or K_{out} was close to zero. Thus, they had lower structural complexity and influence on error propagation due to a lack of information transmission relationships. Compared with other classes in the software, low-rank ones were easier, less fault-prone, and affected fewer functions if they failed. In the process of integration testing, the low-rank classes can be paid less attention.

6.2.2 Evaluation of the risk analysis model according to fault detection efficiency

Because whether a test case can detect faults is unknown before it runs on the software, fault detection efficiency can be treated as the ultimate goal of test case prioritization (TCP) (Hao et al., 2016). For a given software program, different test case execution orders yield different importance of codes

covered by the test cases. Consequently, the sequence of severe faults being detected is determined by the evaluation of code criticality. To further respond to research question RQ_1, we used the total risk indices of classes covered by test cases to prioritize their execution, and then compared them with the seven other TCP techniques based on APFD_c measurement.

As with the definition of APFD_r, we assume that $T = \{t_1, t_2, \cdots, t_{|T|}\}$ is the test case suite and that ct'_i is the execution cost of test case $t_i \in T$. Then, we have

$$\text{APFD}_\text{c} = \frac{\sum_i^{|\mathcal{F}|}\left(\text{sf}_i\left(\sum_{j=\text{TF}_i}^{|T|}\text{ct}'_j - \frac{1}{2}\text{ct}_{\text{TF}'_i}\right)\right)}{\sum_{i=1}^{|T|}\text{ct}'_i\sum_{i=1}^{|\mathcal{F}|}\text{sf}_i}. \tag{30}$$

We used the execution time of each test case as its cost. The eight TCP techniques compared are outlined in Table S2. Fig. 4 shows the APFD_c values of the three software programs obtained by all comparable test case prioritization techniques. The results demonstrated that the fault detection ability of the T8 technique was remarkably close to that of the T3 technique and better than those of the six other strategies. In particular, for the Joda-time software, the APFD_c metric obtained by risk-based prioritization reached a maximum of 0.97. This is because 15 errors existed in the high-risk classes and were detected by executing the top 30% of the test cases. In conclusion, the proposed evaluation scheme is effective in detecting severe faults and reducing the total software risk, and can be applied to address the CITO problem.

6.3 Case studies for RQ_2

We re-implemented the approaches (Jorgensen and Erickson, 1994; Tai and Daniels, 1999; Le Traon et al., 2000; Briand et al., 2002, 2003; Jiang et al., 2011; Assunção et al., 2014) used in the comparison, and their main parameter settings and implementations are described briefly as follows:

1. For all baseline approaches, we ignored the number of distinct returns and parameter-type measurements, and considered only the attribute dependency and method invocation coupling metrics to evaluate the complexity of the test stubs. We also clarified this as a thread to determine validity, because this might have affected the results of Abdurazik and Offutt (2006), Jiang et al. (2011), and Assunção et al. (2014)'s strategies, which combine four types of factors to measure test costs.

2. Polymorphism was modeled by the proposed MDEN and TDG (Le Traon et al., 2000). However, the ORD model used in the other algorithms does not contain dynamic binding relationships. In comparison, we analyzed only the results from the point of view of static software structure.

3. The multi-objective optimization algorithms were implemented from the version available at JMetal 3.0. Table S3 lists the parameter settings for the six software programs.

4. For multi-objective optimization algorithms, the risk factor can also be treated as an objective to be minimized. Considering this, we combined the proposed risk analysis model with the traditional GA (Briand et al., 2002) and the improved GA (Assunção et al., 2014), including NSGA-II, SPEA2, and PAES, to generate integration test orders, and then

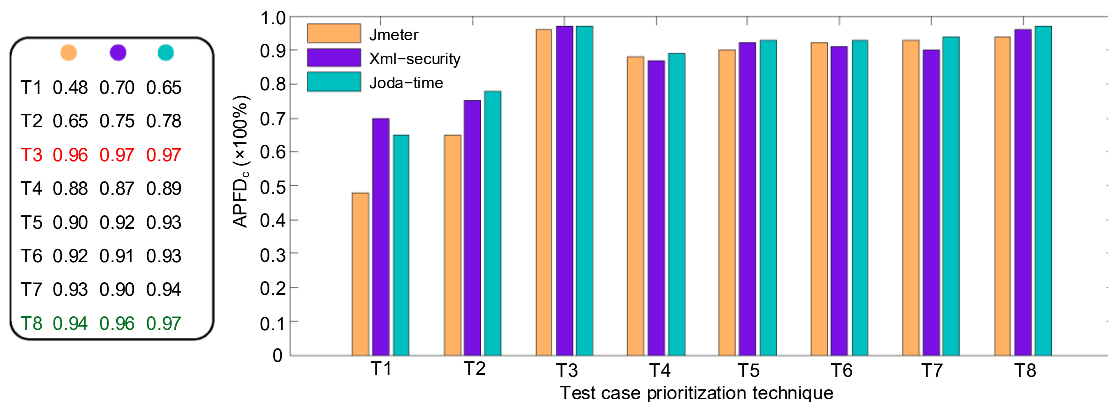

Fig. 4 APFD_c values of the three software programs obtained by all comparable techniques

compared the effectiveness of the proposed graph-based algorithm with those of multi-objective optimization strategies. We used the benefit rate of removing edges quantified by Eq. (14) instead of the fitness function of Briand et al. (2002)'s approach. Moreover, the degree of "high-risk classes being tested first" in the integration test order measured by Eq. (15) was considered as an objective to be optimized in Assunção et al. (2014)'s approach, in addition to stubbing cost. These algorithms are referred to hereinafter as risk-based GA, risk-based NSGA-II, risk-based SPEA2, and risk-based PAES.

Based on the cycle-breaking operations (the statistics of operations for breaking cycles in the three software programs are available at https://github.com/FanyiMeng-NEU/Breaking-cycles-Info), Table S4 describes the integration test orders of six subjects. Existing works (Briand et al., 2002; Vergilio et al., 2012; Assunção et al., 2014) have adopted the above subjects, and provided the experimental results of the integration test orders, to facilitate future validation and comparison with existing CITO algorithms. Tables S5–S10 describe the statistics, where N_s is the number of constructed test stubs, OCplx represents the total complexity of the test stubs, NM and NA denote the numbers of simulation methods and attributes in the stubs, respectively, N_p is the number of tested nodes ranking in the top 30% of classes by the risk index after half of the classes are tested, N_ft represents the number of faults detected in the middle of integration testing, PR denotes the degree of "high-risk classes being integrated first" in the test order, ΣBf is the benefit rate of the obtained test order, which equals PR/OCplx, and $\mathrm{APFD_r}$ reflects the detection efficiency for high-risk faults. The experimental results are analyzed and discussed blow.

Two SCCs existed in the DNS software, i.e., {33, 38, 52} and {58, 48, 32, 25, 11, 8, 21}, whose nodes and edges formed 16 cycles. Six edges, $-21 \rightarrow 11$, $8 \rightarrow 21$, $48 \rightarrow 32$, $32 \rightarrow 58$, $38 \rightarrow 33$, and $52 \rightarrow 33$, were removed by the proposed approach. The total complexity of the test stubs was 1.27, lower than those of the other approaches. The numbers of simulation methods and attributes in the stubs constructed by our approach were smaller those of Le Traon et al. (2000) by 70 and 58, respectively.

The ANT software contained only one SCC {4, 22, 23, 19, 21, 10, 18, 20, 17, 16, 24}, which formed 654 cycles. Briand et al. (2002), Briand et al. (2003), and Jiang et al. (2011) deleted 13, 11, and 10 edges, respectively, to break all cycles in the software, whereas the number of edges removed by our approach was 14 and, as such, the total complexity of the test stubs constructed by the proposed algorithm was lower than those of Tai and Daniels (1999), Le Traon et al. (2000), Briand et al. (2002), Abdurazik and Offutt (2006), and Jiang et al. (2011) by 4.37, 2.66, 0.64, 0.40, and 0.24, respectively. This indicated that in the process of constructing stubs, Tai and Daniels (1999), Le Traon et al. (2000), Briand et al. (2002), Abdurazik and Offutt (2006), and Jiang et al. (2011) needed to simulate 552, 385, 188, 165, and 172 entities, respectively, and the number of simulated entities of stubs constructed by our approach was only 123, thereby reducing the corresponding test expense.

One SCC existed in the BCEL software, which contained 40 nodes. In other words, except for classes 1, 3, 23, 24, and 42, all classes constituted SCCs. There were 416 091 cycles in SCC, although the software contained only 45 classes. To break all cycles, Le Traon et al. (2000), Briand et al. (2002), Briand et al. (2003), and Jiang et al. (2011) on average removed 67, 71, 70, and 73 edges, respectively. However, the number of edges removed by our approach was 75, similar to those removed by the above algorithms, but much lower than those by Tai and Daniels (1999) and Abdurazik and Offutt (2006)'s approaches. Compared with Tai and Daniels (1999), Le Traon et al. (2000), Briand et al. (2002), and Jiang et al. (2011), the total test stub complexity decreased by 7.06, 7.88, 2.47, and 4.85, respectively. Further, the number of simulation methods and attributes decreased by 225, 242, 13, and 120, respectively.

Similar conclusions can be drawn for the Jmeter, Xml-security, and Joda-time software. Ignoring the risk factors, the NSGA-II algorithm used in Assunção et al. (2014)'s strategy always needed a minimum test cost for the small software, whereas the PAES algorithm of Assunção et al. (2014) achieved better results when analyzing the more complex software. Test effort involved in the proposed strategy was very close to that devoted to achieving the optimal results, and even less than that involved in approaches which aim only to minimize the total

complexity of the test stubs.

In these six case studies, the N_{p} values obtained by the proposed approach, risk-based NSGA-II, risk-based PAES, and risk-based SPEA2 were very close and significantly higher than those of all the other algorithms, and this advantage was stable in software of varying sizes. Halfway through the testing, 90% of critical classes on average had been integrated by the proposed approach, which was nearly double the results of Tai and Daniels (1999), Le Traon et al. (2000), Briand et al. (2002), Briand et al. (2003), Abdurazik and Offutt (2006), Jiang et al. (2011), and Assunção et al. (2014), and 1.48 times higher than that of risk-based GA, which also considered both risk factors and test costs.

From Tables S5–S10, we can see that compared with the 13 baseline algorithms, the proposed strategy increased the $\mathrm{APFD_r}$ value to varying extents: on average, it was 49%, 43%, 46%, 42%, 50%, 41%, 27%, 27%, 29%, 31%, 14%, 13%, and 13% higher than the results of Tai and Daniels (1999), Abdurazik and Offutt (2006), Briand et al. (2002), Briand et al. (2003), Le Traon et al. (2000), Jiang et al. (2011), NSGA-II in Assunção et al. (2014), PAES in Assunção et al. (2014), SPEA2 in Assunção et al. (2014), risk-based GA, risk-based NSGA-II, risk-based PAES, and risk-based SPEA2, respectively. In particular, 100% of faults were detected in the middle of integration testing. Based on the above experimental analysis, we can draw the following conclusions:

1. The PR metric defined in Eq. (15) is equal to the degree of "high-risk class being tested first" in test order from the perspective of the global optimum. As a result, all multi-objective optimization algorithms yielded better results when considering the PR metric as an objective.

2. A limitation observed in the risk-based GA strategy was that it removed only the dependencies in SCCs to maximize the benefit rates of integration testing without determining the ultimate results. This is why its total test stub complexity was acceptable, whereas the PR and $\mathrm{APFD_r}$ values were smaller than the ideal ones compared with the other risk-based strategies.

3. The evolutionary algorithms used in Assunção et al. (2014)'s approach yielded better solutions than the traditional GA, because they did not need to weigh the objectives when generating test orders. Nevertheless, due to conflicts among the three objectives, the multi-objective optimization algorithms could not guarantee stable performance in balancing benefits and costs. Taking the Jmeter software as an example, although high-risk classes were assigned higher priorities, slightly more simulation methods for stubbing reduced the overall yield. However, our approach broke the cycles according to the edge weights quantified by cost–benefit analysis, and tended toward the principle of "assigning a higher priority to the class with a higher risk index" when there was more than one edge with the greatest weight. Thus, the proposed graph-based strategy performed well across different software.

4. For small-scale software, the evolutionary algorithms can obtain satisfactory results. In particular, for ANT software, PR, $\Sigma\mathrm{Bf}$, and N_{p} measurements give preference to the risk-based NSGA-II algorithm. However, with more classes in total, the benefit rate and fault-detection efficiency of our strategy became more advantageous. In the case of the BCEL software, which contained 45 classes, the value of $\Sigma\mathrm{Bf}$ obtained by our approach was 68%, 48%, 35%, 49%, 70%, 65%, 10%, 13%, 15%, 15%, 0.4%, 9%, and 3% greater than those of Tai and Daniels (1999), Abdurazik and Offutt (2006), Briand et al. (2002), Briand et al. (2003), Le Traon et al. (2000), Jiang et al. (2011), NSGA-II in Assunção et al. (2014), PAES in Assunção et al. (2014), SPEA2 in Assunção et al. (2014), risk-based GA, risk-based NSGA-II, risk-based PAES, and risk-based SPEA2, respectively. For the Joda-time software, which had 156 classes, the above differences were 76%, 57%, 59%, 56%, 78%, 59%, 44%, 43%, 51%, 43%, 27%, 22%, and 38%, respectively. However, for the Jmeter software, which was composed of 285 classes, the differences increased to 86%, 57%, 83%, 76%, 90%, 68%, 63%, 51%, 71%, 45%, 47%, 44%, and 50%, respectively.

5. Almost all high-risk classes identified by the proposed PRA model were "hub" nodes in the function profiles of MDEN. Thus, removing edges with these starting nodes broke more cycles simultaneously, which reduced the total number of established test stubs in our strategy. Compared with Jmeter, Xml-security, and Joda-time, the proposed approach constructed fewer test stubs.

6. The execution time listed in Tables S5–S10 did not include the risk analysis step. For

multi-objective optimization algorithms, we provided the average of runtime and the standard deviation. Clearly, the execution times of all graph-based approaches were very close and significantly lower than those of the multi-objective optimization algorithms. Table 5 shows the execution time of risk analysis in all case studies, and we can see that the costs were acceptable, even for more complex software such as Jmeter and Joda-time.

Table 5 Execution time of risk analysis in all case studies

	Execution time (s)					
	Jmeter	Xml-security	Joda-time	DNS	ANT	BCEL
MDEN	0.84	0.97	1.02	0.38	0.23	0.32
$\mathcal{T}$	6.26	8.93	75.39	1.65	1.36	1.39
$\mathcal{V}$	52.85	20.44	42.68	0.89	0.78	0.81
$\mathcal{C}$	0.38	0.78	3.17	0.21	0.18	0.19
Total	60.33	31.12	122.26	3.13	2.55	2.71

7. Here, we set the coefficients α and β defined in Eq. (13) as equal, i.e., $\alpha = \beta = 0.5$. The ITOsolution tool allows testers to assign coefficients according to preference. If the project being tested has complex attributes or methods to be emulated, testers can adjust coefficient α or β to avoid simulating complicated entities.

Consequently, combining the risk evaluation to prioritize class integration can significantly improve the ability to detect very severe faults without increasing costs.

6.4 Case studies for RQ$_3$

Assume that the invested cost is given by Eq. (31). If $\theta \in [0, 1)$, we say that the test cost is limited.

$$B_0 = \theta \left(\sum_{k=1}^{|V^{\mathrm{c}}|} \mathrm{DA}_k^{\max} + \sum_{t=1}^{|N_{\mathrm{s}}|} \mathrm{DA}_t^{\mathrm{stub}} \right). \tag{31}$$

With the test cost limitation, we compared the changes in residual software risk affected by different integration test orders in three cases, $\theta = 20\%$, 50%, 80%, i.e., considering situations where the invested test cost accounted for 20%, 50%, and 80%, respectively, of the total cost of complete software risk elimination. Suppose that our approach assigned costs using the cost–benefit strategy proposed in Section 4.1, and that the other approaches adopted the average distribution scheme. Fig. S6 shows a comparison of the reduction in the total risk index by various integration strategies in all cases. Each curve corresponds to the effectiveness of the test order obtained by a baseline approach, plotted by integration steps along the horizontal axis and the percentage of residual software risk resulting from the integrated classes along the vertical axis. Table 6 shows the statistics of the results corresponding to Fig. S6. Note that the experimental data were simulation results based on the test orders of all comparable algorithms obtained in Section 5.3. For the multi-objective optimization strategies in Assunção et al. (2014), we used their optimal results.

6.5 Threats to validity

1. MDEN model

Because the MDEN model simulates the dynamic execution status of the software, it contains the relationships of dynamic binding between classes. However, the approaches of Tai and Daniels (1999) and Briand et al. (2002) broke the cycles of the software from the point of view of static software structure. To be fair, we ignored dynamic binding relationships in the experiments, which might have affected the results.

2. PRA model

Because different methods have different depths in the call tree, even if their execution probabilities and complexities are the same, their failures have varying influences on the overall software. Thus, the proposed approach can always identify classes with relatively high-risk factors. In the extreme case in which the risk indices of all classes were equal, our strategy can still generate CITOs by considering only the complexity of the established test stubs.

3. Experimental design

With regard to our experimental subjects, the versions of BCEL, ANT, and DNS may be different from the software used in Briand et al. (2002, 2003), da Veiga Cabral et al. (2010), and Jiang et al. (2011). For comparison, we maintained those versions provided in Briand et al. (2002). In addition, as the three software programs did not contain a test case suite, we considered that all the scenarios had an equal probability of being executed in the experiments of Section 5. Another threat to validity is that we used the numbers of simulation methods and attributes to assess the complexity of the stubs.

Table 6 Residual software risks under different invested costs

Method	ϖ								
	DNS			ANT			BCEL		
	θ = 80%	50%	20%	80%	50%	20%	80%	50%	20%
Tai and Daniels (1999)'s	0.499	0.636	0.873	0.581	0.669	0.884	0.540	0.645	0.862
Abdurazik and Offutt (2006)'s	0.465	0.611	0.864	0.367	0.500	0.825	0.430	0.560	0.829
Briand et al. (2002)'s	0.467	0.612	0.864	0.371	0.503	0.826	0.420	0.553	0.826
Briand et al. (2003)'s	0.469	0.614	0.865	0.405	0.530	0.836	0.438	0.566	0.831
Le Traon et al. (2000)'s	0.526	0.655	0.880	0.515	0.617	0.866	0.546	0.650	0.864
Jiang et al. (2011)'s	0.469	0.613	0.865	0.393	0.521	0.832	0.494	0.610	0.848
NSGA-II in Assunção et al. (2014)	0.464	0.610	0.865	0.353	0.489	0.821	0.421	0.553	0.826
PAES in Assunção et al. (2014)	0.470	0.615	0.863	0.353	0.489	0.821	0.417	0.550	0.825
SPEA2 in Assunção et al. (2014)	0.469	0.613	0.865	0.353	0.489	0.821	0.422	0.554	0.827
Risk-based GA	0.468	0.613	0.865	0.379	0.510	0.828	0.433	0.563	0.830
Risk-based NSGA-II	0.473	0.617	0.866	0.360	0.494	0.823	0.436	0.565	0.831
Risk-based PAES	0.470	0.614	0.865	0.383	0.513	0.830	0.433	0.563	0.830
Risk-based SPEA2	0.473	0.617	0.866	0.388	0.517	0.831	0.436	0.565	0.831
Ours	**0.220**	**0.316**	**0.520**	**0.207**	**0.307**	**0.568**	**0.205**	**0.310**	**0.524**

Method	R_{ϖ}								
	DNS			ANT			BCEL		
	θ = 80%	50%	20%	80%	50%	20%	80%	50%	20%
Tai and Daniels (1999)'s	−0.009	−0.006	−0.002	−0.023	−0.018	−0.006	−0.011	−0.009	−0.003
Abdurazik and Offutt (2006)'s	−0.008	−0.006	−0.002	−0.023	−0.019	−0.007	−0.009	−0.007	−0.003
Briand et al. (2002)'s	−0.008	−0.006	−0.002	−0.02	−0.016	−0.006	−0.010	−0.008	−0.003
Briand et al. (2003)'s	−0.008	−0.005	−0.002	−0.021	−0.020	−0.007	−0.011	−0.008	−0.003
Le Traon et al. (2000)'s	−0.009	−0.007	−0.002	0.0191	−0.015	−0.005	−0.007	−0.006	−0.002
Jiang et al. (2011)'s	−0.009	−0.007	−0.002	−0.022	−0.017	−0.006	−0.008	−0.006	−0.002
NSGA-II in Assunção et al. (2014)	−0.009	−0.007	−0.002	−0.025	−0.024	−0.011	−0.014	−0.011	−0.004
PAES in Assunção et al. (2014)	−0.009	−0.007	−0.002	−0.027	−0.025	−0.010	−0.013	−0.010	−0.005
SPEA2 in Assunção et al. (2014)	−0.009	−0.007	−0.002	−0.030	−0.023	−0.010	−0.014	−0.011	−0.004
Risk-based GA	−0.010	−0.008	−0.002	−0.029	−0.023	−0.008	−0.012	−0.009	−0.004
Risk-based NSGA-II	−0.013	−0.009	−0.003	−0.046	−0.036	−0.013	−0.016	−0.013	−0.006
Risk-based PAES	−0.012	−0.009	−0.003	−0.044	−0.034	−0.013	−0.015	−0.012	−0.005
Risk-based SPEA2	−0.012	−0.008	−0.003	−0.046	−0.035	−0.013	−0.014	−0.012	−0.005
Ours	**−0.020**	**−0.017**	**−0.013**	**−0.067**	**−0.056**	**−0.027**	**−0.025**	**−0.026**	**−0.019**

However, the approaches proposed by Abdurazik and Offutt (2006), Jiang et al. (2011), and Assunção et al. (2014) considered other stub characteristics. This might have affected the evaluation of test costs.

7 Conclusions

Devising optimal integration test orders impacts the development and evolution of software. Prior studies have been dedicated to reducing test costs in integration test order generation but ignore test efficiency. Test efficiency affects the sequence in which classes are developed and inter-class faults are detected, and affects the design of test cases and construction of test stubs. Thus, we have proposed a multi-layer dynamic execution network model to analyze the dynamic topology of software from both temporal and spatial perspectives. This model maps the function profiles of the software in various scenarios triggered by users into network layers consisting of execution paths. On this basis, the dynamic execution probabilities, complexities, and fault consequences of classes in software were quantified. Furthermore, the testing importance of classes was evaluated by risk analysis. In this manner, the class with a high failure rate and error propagation influence was assigned a high priority. However, if we directly generate integration test orders based on risk indices, the total complexity of the constructed test stubs increases drastically. To solve this problem, we weighed each edge in the class-level network with a trade-off between test benefits and costs. Then,

cycle-breaking operations were performed by removing edges based on their weights. We proposed an evaluation scheme that assesses the effectiveness of integration test orders based on the risk reduction rate of the overall software. Moreover, the $APFD_r$ metric was proposed to measure the average percentage of faults detected per cost of an integration test order. Finally, a strategy for optimizing the allocation of testing resources was presented which balances the expected risk reductions of each class against the required test efforts. Compared to existing algorithms, we concluded that the proposed approach guarantees that higher-risk classes are tested earlier and, as such, minimizes the total complexity of the test stubs. With test cost limitations, our integration strategy can obtain a higher benefit rate in software risk reduction. Perhaps of greatest practical significance is the fact that the functions mentioned above are encapsulated in an executable tool ITOsolution, which allows users to assess the risk factors of classes and generate integration test orders automatically.

In future work, we plan to apply the ITOsolution tool on larger datasets and further evaluate the effectiveness of the proposed approach.

Contributors

Fanyi MENG and Ying WANG designed the research and processed the data. Fanyi MENG drafted the paper. Ying WANG helped organize the paper. Hai YU and Zhiliang ZHU revised and finalized the paper.

Compliance with ethics guidelines

Fanyi MENG, Ying WANG, Hai YU, and Zhiliang ZHU declare that they have no conflict of interest.

References

Abdurazik A, Offutt J, 2006. Coupling-based class integration and test order. Proc Int Workshop on Automation of Software Test, p.50-56. https://doi.org/10.1145/1138929.1138940

Al Mannai WI, Lewis TG, 2008. A general defender-attacker risk model for networks. *J Risk Finance*, 9(3):244-261. https://doi.org/10.1108/15265940810875577

Amland S, 2000. Risk-based testing: risk analysis fundamentals and metrics for software testing including a financial application case study. *J Syst Softw*, 53(3):287-295. https://doi.org/10.1016/S0164-1212(00)00019-4

Assunção WKG, Colanzi TE, Vergilio SR, et al., 2014. A multi-objective optimization approach for the integration and test order problem. *Inform Sci*, 267:119-139. https://doi.org/10.1016/j.ins.2013.12.040

Bang L, Aydin A, Bultan T, 2015. Automatically computing path complexity of programs. Proc 10th Joint Meeting on Foundations of Software Engineering, p.61-72. https://doi.org/10.1145/2786805.2786863

Bansal P, Sabharwal S, Sidhu P, 2009. An investigation of strategies for finding test order during integration testing of object oriented applications. Proc Int Conf on Methods and Models in Computer Science, p.1-8. https://doi.org/10.1109/ICM2CS.2009.5397936

Binder RV, 1996. Testing object-oriented software: a survey. *Softw Test Verif Reliab*, 6(3-4):125-252. https://doi.org/10.1002/(SICI)1099-1689(199609/12)6:3/4<125::AID-STVR121>3.0.CO;2-X

Bowring JF, Rehg JM, Harrold MJ, 2004. Active learning for automatic classification of software behavior. *ACM SIGSOFT Softw Eng Notes*, 29(4):195-205. https://doi.org/10.1145/1013886.1007539

Briand LC, Jie F, Labiche Y, 2002. Experimenting with genetic algorithms to devise optimal integration test orders. Technical Report, No. SCE-02-03, Carleton University, Ottawa, Canada.

Briand LC, Labiche Y, Wang YH, 2003. An investigation of graph-based class integration test order strategies. *IEEE Trans Softw Eng*, 29(7):594-607. https://doi.org/10.1109/TSE.2003.1214324

Cai KY, Yin BB, 2009. Software execution processes as an evolving complex network. *Inform Sci*, 179(12):1903-1928. https://doi.org/10.1016/j.ins.2009.01.011

da Veiga Cabral R, Pozo A, Vergilio SR, 2010. A Pareto ant colony algorithm applied to the class integration and test order problem. Proc 22nd IFIP WG 6.1 Int Conf on Testing Software and Systems, p.16-29. https://doi.org/10.1007/978-3-642-16573-3_3

Deb K, Pratap A, Agarwal S, et al., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput*, 6(2):182-197. https://doi.org/10.1109/4235.996017

Floyd RW, 1962. Algorithm 97: shortest path. *Commun ACM*, 5(6):345. https://doi.org/10.1145/367766.368168

Frankl PG, Weyuker EJ, 2000. Testing software to detect and reduce risk. *J Syst Softw*, 53(3):275-286. https://doi.org/10.1016/S0164-1212(00)00018-2

Goseva-Popstojanova K, Hassan A, Guedem A, et al., 2003. Architectural-level risk analysis using UML. *IEEE Trans Softw Eng*, 29(10):946-960. https://doi.org/10.1109/TSE.2003.1237174

Hao D, Zhang L, Zang L, et al., 2016. To be optimal or not in test-case prioritization. *IEEE Trans Softw Eng*, 42(5):490-505. https://doi.org/10.1109/TSE.2015.2496939

Henry S, Kafura D, 1981. Software structure metrics based on information flow. *IEEE Trans Softw Eng*, 7(5):510-518. https://doi.org/10.1109/TSE.1981.231113

Huang CY, Lyu MR, 2005. Optimal testing resource allocation, and sensitivity analysis in software development. *IEEE Trans Reliab*, 54(4):592-603. https://doi.org/10.1109/TR.2005.858099

Jiang SJ, Zhang YM, Li HY, et al., 2011. An approach for inter-class integration test order determination based on coupling measures. *Chin J Comput*, 34(6):1062-1074. https://doi.org/10.3724/SP.J.1016.2011.01062

Jiang SJ, Zhang M, Zhang YM, et al., 2021. An integration test order strategy to consider control coupling. *IEEE Trans Softw Eng*, 47(7):1350-1367. https://doi.org/10.1109/TSE.2019.2921965

Jorgensen PC, Erickson C, 1994. Object-oriented integration testing. *Commun ACM*, 37(9):30-38. https://doi.org/10.1145/182987.182989

Knowles JD, Corne DW, 2000. Approximating the nondominated front using the Pareto archived evolution strategy. *Evol Comput*, 8(2):149-172. https://doi.org/10.1162/106365600568167

Kung D, Gao J, Hsia P, et al., 1995. A test strategy for object-oriented programs. Proc 19th Annual Int Computer Software and Applications Conf, p.239-244. https://doi.org/10.1109/CMPSAC.1995.524786

Le Traon Y, Jeron T, Jezequel JM, et al., 2000. Efficient object-oriented integration and regression testing. *IEEE Trans Reliab*, 49(1):12-25. https://doi.org/10.1109/24.855533

Lipow M, 1982. Number of faults per line of code. *IEEE Trans Softw Eng*, 8(4):437-439. https://doi.org/10.1109/TSE.1982.235579

McCabe TJ, 1976. A complexity measure. *IEEE Trans Softw Eng*, 2(4):308-320. https://doi.org/10.1109/TSE.1976.233837

Monden A, Hayashi T, Shinoda S, et al., 2013. Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Trans Reliab*, 39(10):1345-1357. https://doi.org/10.1109/TSE.2013.21

Myers CR, 2003. Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Phys Rev E*, 68(2):046116. https://doi.org/10.1103/PhysRevE.68.046116

NASA, 1999. Pyroshock Test Crite. Technical Report, No. NASA-STD-7003A. NASA, Washington, USA.

Sharma C, Sibal R, 2013. Application of different metaheuristic techniques for finding optimal test order during integration testing of object oriented systems and their comparative study. *Int J Soft Comput Eng*, 3(12):1-19.

Tai KC, Daniels JF, 1999. Interclass test order for object-oriented software. *J Obj-Orient Progr*, 12(4):18-25.

Vergilio SR, Pozo A, Árias JCG, et al., 2012. Multi-objective optimization algorithms applied to the class integration and test order problem. *Int J Softw Tools Technol Transf*, 14(4):461-475. https://doi.org/10.1007/s10009-012-0226-1

Walters C, Ludwig D, 1994. Calculation of Bayes posterior probability distributions for key population parameters. *Can J Fish Aquat Sci*, 51(3):713-722. https://doi.org/10.1139/f94-071

Wang Y, Zhu ZL, Yang B, et al., 2018a. Using reliability risk analysis to prioritize test cases. *J Syst Softw*, 139:14-31. https://doi.org/10.1016/j.jss.2018.01.033

Wang Y, Zhu ZL, Yu H, et al., 2018b. Risk analysis on multi-granular flow network for software integration testing. *IEEE Trans Circ Syst II Expr Briefs*, 65(8):1059-1063. https://doi.org/10.1109/TCSII.2017.2775442

Wang ZS, Li BX, Wang LL, et al., 2011. A brief survey on automatic integration test order generation. Proc 23rd Int Conf on Software Engineering & Knowledge Engineering, p.254-257.

Weyuker EJ, 1988. Evaluating software complexity measures. *IEEE Trans Softw Eng*, 14(9):1357-1365. https://doi.org/10.1109/32.6178

Xu C, Qin Y, Yu P, et al., 2020. Theories and techniques for growing software: paradigm and beyond. *Sci Sin Inform*, 50(11):1595-1611. https://doi.org/10.1360/SSI-2020-0079

List of supplementary materials

Fig. S1 Sample code illustrating the proposed MDEN model

Fig. S2 Generating an integration test order process

Fig. S3 Main features of our tool ITOsolution

Fig. S4 Class-level dependency networks of the subjects

Fig. S5 Class risk probability density distribution

Fig. S6 Comparison of the reduction in total risk indices

Table S1 Statistics of the top and bottom five nodes

Table S2 Compared test case prioritization techniques

Table S3 Parameter settings

Table S4 Test orders obtained by the proposed algorithm

Table S5 Comparison of experimental results for Jmeter

Table S6 Comparison of experimental results for Xml-security

Table S7 Comparison of experimental results for Joda-time

Table S8 Comparison of experiment results for DNS

Table S9 Comparison of experimental results for ANT

Table S10 Comparison of experimental results for BCEL