Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 13, no. 8, December 2017, pp. 1171-1182
DOI: 10.23940/ijpe.17.08.p1.11711182

Testability Metrics for Software Behavioral Models

Pan Liu[a,b,*]

[a]*College of Information and Computer, Shanghai Business School, Shanghai 201400, China*
[b]*Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai 201112, China*

Abstract

Design for testability is one of the important research interests in software engineering. It becomes crucial in model-based testing because software behavioral models can be used to construct test sequences to perform conformance testing. However, testability of models has received less attention in the past. A model with high testability is easy to be used to construct effective test sequences, and conformance testing can also be realized easily. To improve testability of software behavioral models, using the formal method, we present five testability metrics: observability, controllability, test constructibility, performability, and error traceability. Then, a case is studied to evaluate the effectiveness of proposed testability metrics. As a result of the case study, models with higher testability can not only be used to generate executable test sequences, but also the size of the constructed test suite is also smaller. Our research will enrich the modeling theory of model-based testing and can improve the application of this test method in industry.

Keywords: testability metrics; software behavioral model; design for testability; model-based testing

(Submitted on October 29, 2017; Revised on November 12, 2017; Accepted on December 3, 2017)

1. Introduction

Model-based testing [1,2,8,19,25] has been widely studied to generate test sequences from behavior models of the System Under Testing (SUT). Using this test method, we can detect software errors by observing whether the software implementation is consistent with its desired behavior model. Thus, it is also known as conformance testing [17,18,26]. Since software design, software development and software testing are carried out simultaneously in the process of model-based testing. This test method changes the traditional software development process of "now programming, test later", and enables parallel testing in the software development life cycle.

In the past, many researchers and practitioners used a number of models, such as UML diagrams, Finite State Machine (FSM), and Petri nets, to model software behavior and obtained test sequences from behavior models. For instance, state-chart diagram, a UML diagrams, can represent transition effects, state entry/do/exit, composite states, nested state machines, transition guards and conditionals in software [12]. According to these characteristics of the state-chart diagram, Offutt and Abdurazik [29] presented a technique that adopts pre-defined state-based test data generation criteria to generate test cases from UML specifications. However, few studies in the past involved testability metrics of software behavioral models. The more testability a model has, the more effective test sequences generated from the model are. As a result, conformance testing is also easier to be implemented in the software development.

Traditionally, software testability is an important quality characteristic of software [22], which has been described as the ability to test software easily [3,30], as the probability of a program revealing faults [11], or as controllability and observability [6,7,9,14]. However, some researchers thought that it is hard to summarize all influence factors of testability. For example, Fenton et al. [13] defined testability as an external attribute since it cannot be measured simply as size, complexity or coupling. Baudry et al. [4,5] stated that testability measurement is influenced by various factors [33], such as control flow, data flow, complexity and size. Zhao [34] presented that the testability is an elusive concept, thereby hardly getting a clear view on all the potential factors that affect it.

* Corresponding author.
E-mail address: panl008@163.com

To measure the testability of software, many metrics have been proposed in [10,15,28]. In the early days, a few software complexity measures, such as the Cyclomatic number [23] or the Npath metric [24], were proposed as a substitute for testability. Later on, Freedman [14] proposed the domain testability of software components based on controllability and observability notions. More recently, based on inputs and outputs domains of a software component, Voas and Miller [31] proposed a testability metric named as the domain/range ratio. The Propagation, Infection and Execution (PIE) technique was adopted to analyze software testability by Jeffrey and Keith [32]. The formal foundation of testability metrics is also discussed by Sheppard et al. [27]. Jungmayr [16] focused on testability measurement in the context of static dependencies within object-oriented systems.

Most of the existing testability metrics are designed for component validation or unit testing for classes. Therefore, it is difficult to use these testability metrics to evaluate software behavioral models. In this paper, we propose five testability metrics to quantify and analyze testability of software behavioral models. A case is also provided to evaluate the effectiveness of our testability metrics. The study of the case shows that the proposed testability metrics can not only be used to improve the design of behavior models, but also the improved models can be used to produce more effective test cases.

This paper is structured as follows. Section 2 states the significance of testability metrics for software behavioral model to ensure software quality. Section 3 gives some preliminaries for constructing testability metrics. Section 4 presents five testability metrics of behavior models. Section 5 evaluates the validity of these testability metrics through a case study. Section 6 concludes the whole paper.

2. Motivation for Designing Testability Metrics

In this section, we state our motivation for designing testability metrics of software behavioral models with respect to both model-based testing and model-driven development (MDD). Fig.1 (a) shows the traditional process of both model-based testing and MDD. We usually utilize the modeling tool to construct the behavior model of the SUT. To ensure and guarantee the quality of the model, we adopt the technique of model verification to make it satisfy some given properties. Then, test sequences can be generated from the validated model by means of many graphic traversal algorithms. At the same time, the SUT is developed in terms of the method of MDD. Finally, we perform conformance testing by using those test sequences to check the differences between the SUT and the behavior model.

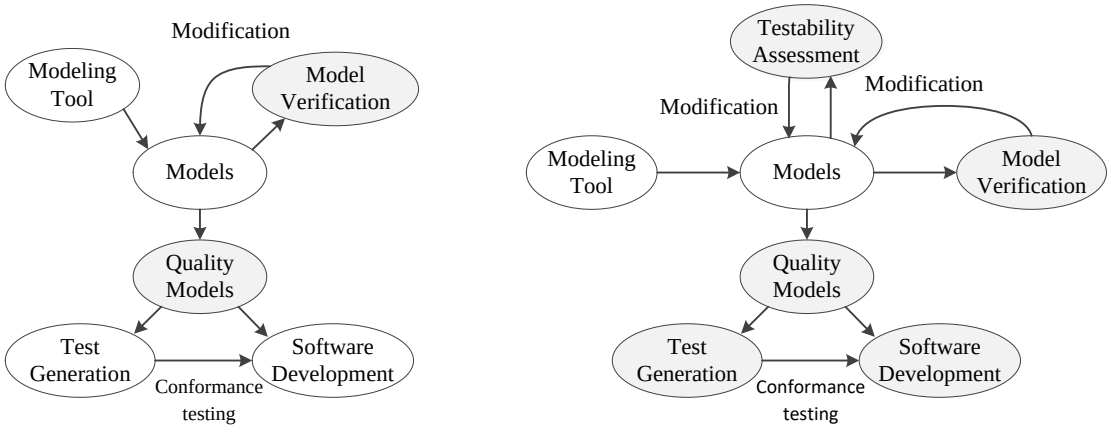

Figure 1. Two models combined model-based testing with model-driven development

However, is it enough to verify the model to generate effective test cases from the model? The answer is no. The model verification technique can't ensure that the designed model is reasonable. If the design of a model is unreasonable, the SUT that is developed in accordance with the model is also unreasonable. As a result, the SUT may contain more bugs. Another problem is the fact that some invalid test cases are often obtained from the verified model, resulting in the increase of test cost of software.

To overcome the above two problems, we suggest adding a testability evaluation process to the model shown in Fig.1 (a). Firstly, we employ a suite of testability metrics to evaluate model's testability. Based on the evaluation results, we can modify the model so that the new model is more suitable for test generation. Meanwhile, the SUT is also developed according to the new model. In this way, the process of conformance testing is successfully performed. Fig.1 (b) shows the improved process of both model-based testing and MDD.

3. Preliminaries

In this section, we give some preliminaries of software behavioral models to help establish some relevant testability indicators.

Definition 1 (action). In the system, actions are defined as a triple $Action=(Tr, \delta, \lambda)$, where

- $Tr = (S, I/O, S)$ is a set of transitions of the system, where S is the set of states of the system, I is the input set and O is the output set of the system,
- $\delta: S \times I \to S$ is the state transition function, and
- $\lambda: S \times I \to O$ is the output function.

Note: In a transition $t=(s_1, x/y, s_2)$, x is called as the transition condition of t, which makes the system reach state s_2 from state s_1 as well as output a result y in state s_2, and s_1 and s_2 are called as pre-state and post-state of t, respectively.

Definition 2 (software behaviors). In the system, software behaviors are a two-dimensional array $B=(Action, R)$, where

- *Action* is a set of actions in the system, and
- R indicates the relationship between different actions, such as concatenation, choice, synchronization, and concurrence.

Definition 3 (software behavioral model). A software behavioral model is a two-dimensional array $M=(S, B)$, where

- S is a nonempty set of states in the system, and
- B is a set of behaviors that are compatible with the laws of the system.

Definition 4 (model-based testing). Model-based testing is to use the model to describe the behaviors of the SUT and then generate test sequences from the model to observe whether the implementation of the SUT is consistent with the descriptions of the model.

Note: According to the above mentioned preliminaries, we can design some evaluation indicators for testability of software behavioral models, regardless of the differences of the definitions for describing some existing models.

4. Testability Metrics

In the past, observability and controllability were two common indicators of testability measurement for the object-oriented system in [7,9,10,22]. Software behavioral models not only needs to satisfy these two properties, but also can be easily used to generate effective test sequences to achieve conformance testing. For this reason, the paper extends these two assessment measures and presents five testability metrics for software behavioral models: observability, controllability, test constructibility, performability, and error traceability.

4.1. Observability

Observability (*OB*) of software behavioral models is the capability of the test system to observe the outputs of behavior models and to determine which input triggers the particular output.

Definition 5 (observable post-state). In a transition t, when we input a condition on the pre-state of t, a result will be outputted on the post-state in t. If we can observe the output in the post-state in t, the post-state is called the observable post-state.

Definition 6 (observable transition). If the pre-state, the next-state, the input, and the output in the transition can be observed in the process of conformance testing, this transition is called the observable transition.

In model-based testing, we can obtain a state sequence taken as a test path by means of the state transition function after we input some transition conditions in the states. At the same time, a set of output results is also shown in those states through the output function. Observability of software behavioral models is a measure of conformance testing by observing whether the execution paths of the SUT are consistent with expected state sequences or whether there exist some expected output results in these post-states. Therefore, it consists of the State Observability (*SO*) and the Output Observability (*OO*). *SO* is defined as

$$SO = \frac{N_{os}}{N_s} \tag{1}$$

where N_{os} denotes the number of all observable post-states in transitions of the behavior model and N_s denotes that of the post-states in transitions of the behavior model.

OO denotes the capability of the model to observe the outputs in all transitions, and it can be defined as

$$OO = \frac{N_{oo}}{N_{ot}} \tag{2}$$

where N_{oo} denotes the number of all observable outputs in transitions of the behavior model and N_{ot} denotes that of all outputs in transitions of the behavior model.

From Eq. (1) and Eq. (2), *OB* of software behavioral models can be defined as

$$OB = SO+OO \tag{3}$$

Example 1: There are two FSMs shown in Fig. 2, where M_I is a SUT, M_S is the behavior model of M_I, and the notation "-" in two FSMs denotes that the output is null, which can't be observed by a tester in conformance testing. We can find out the differences between M_S and M_I by observing the outputs in the transitions or by reaching the unexpected states. In this example, we can discover an error output $\underline{b/1}$ in the transition (s_3, b/1, s_1) in M_I when we input b in the state s_3. Similarly, the unexpected state s_2 can be reached in M_I when we input a in the state s_3.

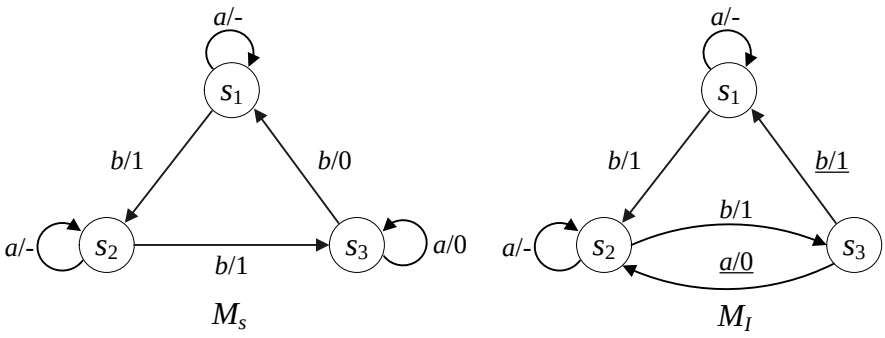

Figure 2. Two FSM models

Now, we discuss observability of the model M_S. From Fig.2, there are transitions (s_1, a/-, s_1), (s_1, b/1, s_2), (s_2, a/-, s_2), (s_2, b/1, s_3), (s_3, a/0, s_3), and (s_3, b/0, s_1) in M_S. Thus, there are N_s = 6, N_{os} = 4, N_{oo} =4, and N_{ot} =6 for M_S. Therefore observability of M_S shown in Fig.2 is *OB*=4/3 according to Eq.(3).

4.2. Controllability

Controllability (*CO*) of behavior models describes the ability that the SUT can be executed according to given test sequences generated from the behavior model. It can be defined as

$$CO = \frac{P_o}{P} \tag{4}$$

where P_o denotes the number of controllable paths (test sequences) in the behavior model, which can be directly executed in the SUT, and P is the number of all paths in the behavior model.

Example 2: Fig.3 shows an abstract activity diagram with a synchronous operation, where the notations *a*, *b*, *c*, *d*, *e*, *f* and *g* on arcs denote transitions. *c* and *d* are synchronous, s_0 is the initial state, and ε denotes an empty transition.

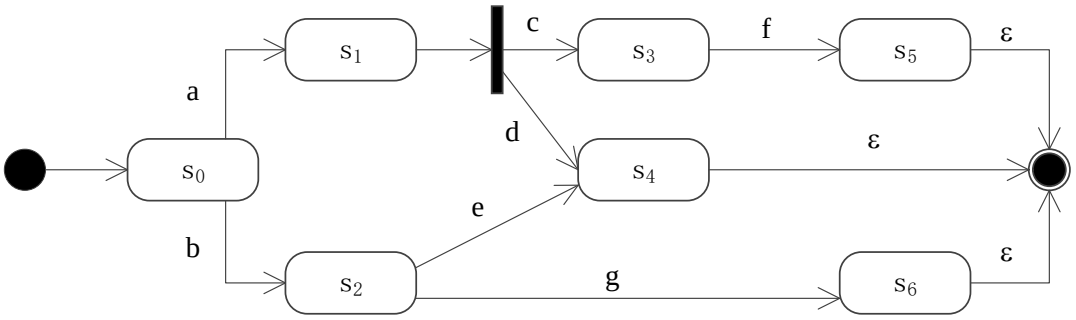

Figure 3. An abstract activity diagram with the synchronous operation

Now, we analyze controllability of this activity diagram as shown in Fig.3 according to Eq. (4). To obtain both P_o and P, we refer to the test generation method in [20] based on regular expressions to construct test sequences from the state diagram. The state diagram in Fig.3 is transformed into an extended regular expression $a.(c.f.\varepsilon \circ d.\varepsilon)|b.(e.\varepsilon|g.\varepsilon)$ according to a set of transformational rules in [20], where the notation . denotes the concatenation operation, the notation | denotes the choice operation and the notation ° denotes the synchronous operation.

Definition 7 (transition sequence). The syntax of the transition sequence *ts* is defined as:

$$ts = \varepsilon \mid a \mid a.ts \mid ts.a,$$

where ε denotes the empty, *a* is any a transition and *ts* is any transition sequence.

To generate test sequences from the extended regular expression $a.(c.f.\varepsilon \circ d.\varepsilon)|b.(e.\varepsilon|g.\varepsilon)$, we give theorem 1.

Theorem 1. In *example* 2, there exists $a.(c.f.\varepsilon \circ d.\varepsilon)|b.(e.\varepsilon|g.\varepsilon) = a.c.f.d \mid a.c.d.f \mid a.d.c.f \mid b.e \mid b.g$.

Theorem 1 can be proved by reference to Theorem 1 in [20], and its proof is not included due to page limit. By theorem 1, we obtain five transition sequences *a.c.f.d*, *a.c.d.f*, *a.d.c.f*, *b.e* and *b.g* from the extended regular expression $a.(c.f.\varepsilon \circ d.\varepsilon)|b.(e.\varepsilon|g.\varepsilon)$. Now we transform those sequences into five state sequences denoting as test paths, including path1: s_0-s_2-s_4, path 2: s_0-s_2-s_6, path 3: s_0-s_1-s_3-s_5…s_1-s_4, path 4: s_0-s_1-s_3…s_1-s_4-s_3-s_5, and path 5: s_0-s_1-s_4…s_1-s_3-s_5. For these five paths, path 1 and path 2 are controllable, while paths 3-5 are uncontrollable because it is hard to get back s_1 from s_5, s_3 and s_4 when we test the system by using five paths. Hence there are $P_o = 2$, $P = 5$ and CO=2/5 according to Eq.(4) in the state diagram shown in Fig.3.

4.3. Test Constructibility

Test Constructibility (*TC*) of software behavioral models is the ability to cover all test requirements by test sequences generated from this model in a given time. It can be defined as

$$TC = \beta \times \frac{TS_e}{TS} \tag{5}$$

where TS_e denotes the number of effective test sequences generated by a traversal algorithm of the model, *TS* denotes the total number of test sequences and β is a coefficient that satisfies the following formula:

$$\beta = \begin{cases} 1 & T \geq T_a \\ \dfrac{T}{T_a} & T < T_a \end{cases} \tag{6}$$

where *T* denotes the given time and T_a denotes the running time of the traversal algorithm for generating test sequences from the model.

Note: Different traversal algorithms have different coverage capability of test requirements and consume different time to generate test sequences from the model.

Example 3: To test a Web application, we not only need to consider those basic functions of the system, but also should take into account the impact of the browser on the system. Generally, the browser contains a basic navigation operation *back*. The user can go back to a page by this operation. Fig. 4 (a) shows a model of a simple Web application. This model contains three pages as three states s1, s2, and s3, five transitions (s1, modify, s2), (s1, link, s3), (s2, view, s3), (s3, *back*, s1), and (s3, *back*, s2).

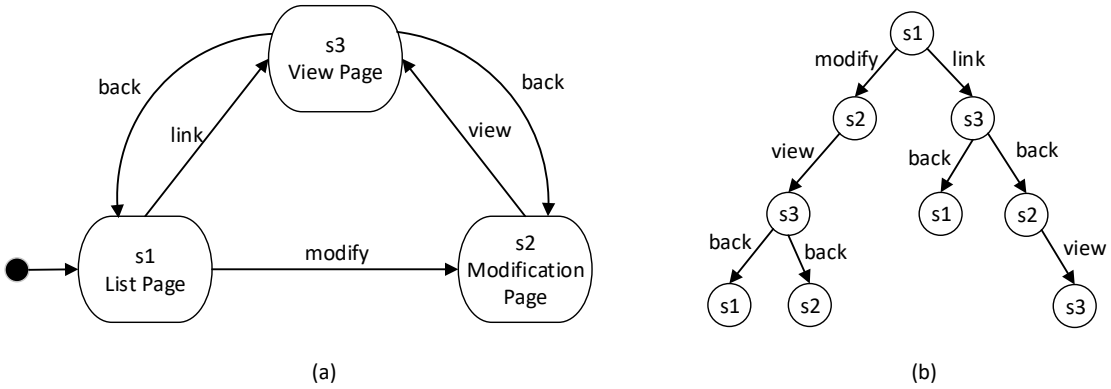

Figure 4. A Web navigation model and its test tree

Now we construct a test tree shown in Fig. 4 (b) of this model. From Fig. 4 (b), we can obtain four test paths from the tree. However, the paths s1-modify-s2-view-s3-back-s2 and s1-link-s3-back-s2-view-s3 are non-executable because the system can't go back to s1 from s3 in the former, and also can't reach to s2 from s3 in the latter by the operation *back* of the browser. Therefore, there are TS_e=2 and TS=4. Assume that there is enough time to generate all test paths. Then, test constructability of the model in Fig. 4 (a) is TC=1/2. Note: The bigger value of TC is, the better testability of the model has.

4.4. Performability

Performability (PE) of software behavioral models describes the ability that test segments in test sequences generated from behavior models can be executed independently to test the SUT. Performability of software behavioral models is defined as

$$PE = RS \times IA \tag{7}$$

where RS denotes the ability that states in the software behavioral model can be directly visited via the initial state, and IA denotes the ability that inputs in transitions of the software behavioral model can be modified so that the expected target state in this model can arrive.

To measure RS and IA, we also define two formulas as follows:

$$RS = \frac{S_r}{S_a} \tag{8}$$

$$IA = \frac{I_t}{I_a} \tag{9}$$

where S_r and S_a denote the number of directly reachable states from the initial state and that of all states in the model, respectively, and I_t and I_a denote the number of alterable inputs and that of all inputs in transitions of the model, respectively.

Note: To increase performability of software behavioral models, some parts similar to stub modules and driver modules in programs need to be designed for the model in advance when we construct the software behavioral model. The advantage of the approach is to economize the testing cost because part of the model does not need to be tested repeatedly in regression testing or in the large-scale integration testing. Generally, the model is transformed into a test tree by means of a graph

searching algorithm. Then, the sequences from the root to the leaves of this tree are taken as test paths to test the SUT. However, since those test sequences may have some same segments from the root to a specific node in the test tree, those identical segments will repeatedly be executed when we test the SUT according to test sequences. Therefore, if the states of the behavior model have related stub parts and driver parts, those same segments in test sequences do not need to repeatedly be executed.

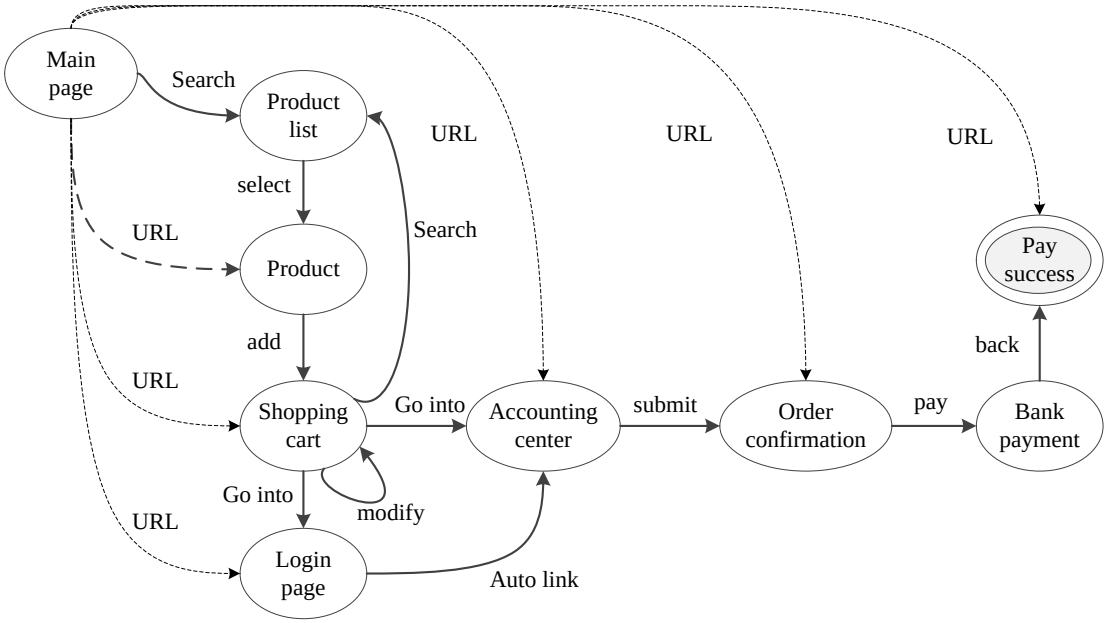

Figure 5. A small shopping website model

Example 4: The state diagram of a small shopping website is shown in Fig.5, where

- *Main page* is the initial state of the website,
- Six dotted lines denote that the tester can go into a specific state (page) from *Main page* by inputting a given hyperlink in the address bar of the browser,
- *Bank payment* is a third-party control for online payment, and
- *Pay success* is the terminal state.

In this shopping website, we consider that customers can leave the website at any stage and go into other pages by changing the hyperlink in the address bar of the browser. Therefore, in this example, there are S_a=9, S_r= 8, I_a= 11 and I_t = 11. Performability of the model shown in Fig.5 is PE = 8/9 according to Eq. (7).

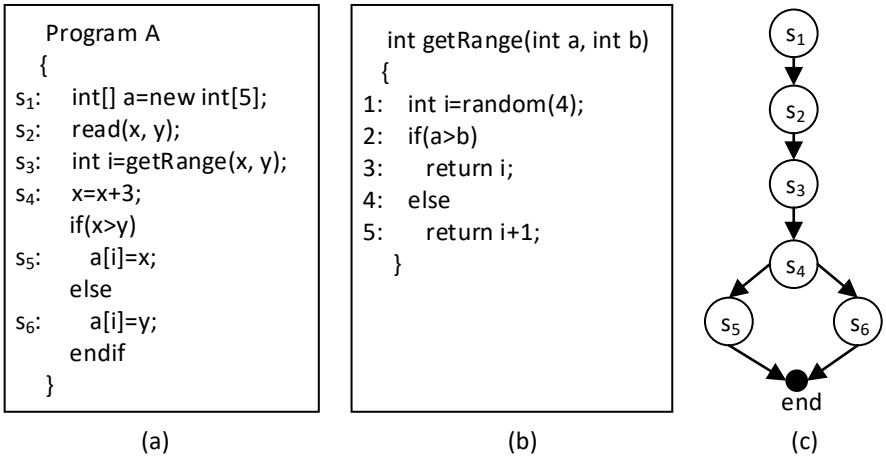

Figure 6. A program and its state diagram

4.5. Error Traceability

Error Traceability (*ET*) of software behavioral models describes the ability that test sequences generated from the model can locate software errors in the SUT. We make use of an example to motivate our ideas for quantitative analysis of error traceability of software behavioral models.

Example 5: Fig.6 (a) shows Program A, Fig.6 (b) is a called function *getRange* by Program A, and (c) is the state diagram of Program A. There is an overflow error in s_6 of Program A when the returned value of the called function *getRange* is 5. However, it is hard to find this overflow error in s_6 of Program A when we test Program A because the returned value of the called function *getRange* is randomly produced from 0 to 5.

Definition 8 (prior state and following state). In a test sequence $s_1, s_2, \ldots, s_n$, the states $s_1, s_2, \ldots$, and s_{i-1} are called as prior states of the state s_i and the states $s_{i+1}, s_{i+2}, \ldots$, and s_n are called as following states of the state s_i.

In this example, the overflow error in s_6 is related to the value of both the integer *i* in s_3 and the integers *x* and *y* in s_2. In fact, for any programs, if we observe a software error in the state, the underlying source of this error is located in the current state or its prior states in the test path. Therefore, to discover and locate the source of this overflow error in s_6 of Program A, we need to modify the values of *x* and *y* in s_2 and manually set the returned value of the called function *getRange* from 0 to 5 in s_3. In the end, test object of Program A is a test segment s_3, s_4, s_6 when the range of *i* in s_3 is from 0 to 5. Based on the above analyses, we give the definition of error traceability as follows:

$$ET = \sum_{i=1}^{n} PE_i \tag{10}$$

where *i* denotes the number of test sequences generated from the software behavioral model and PE_i denotes performability of the path *i*.

According to Eq. (10), we can calculate error traceability of the model shown in Fig.6 (c). From Fig.6 (c), we obtain path 1: s_1-s_2-s_3-s_4-s_5 and path 2: s_1-s_2-s_3-s_4-s_6. For path 1, there are S_a=5, S_r=1, RS=S_r/S_a=1/5, I_t=3, I_a=5, and IA=I_t/I_a=3/5. Thus, there is PE_1=RS×IA=3/25 according to Eq. (7). For path 2, there exist S_a=5, S_r=1, RS=1/5, I_t =3, I_a=5, and IA=3/5. Thus, there is PE_2 = RS×IA=3/25 according to Eq. (7). Therefore, error traceability of the model in Fig. 6 (c) is ET=6/25 according to Eq. (10).

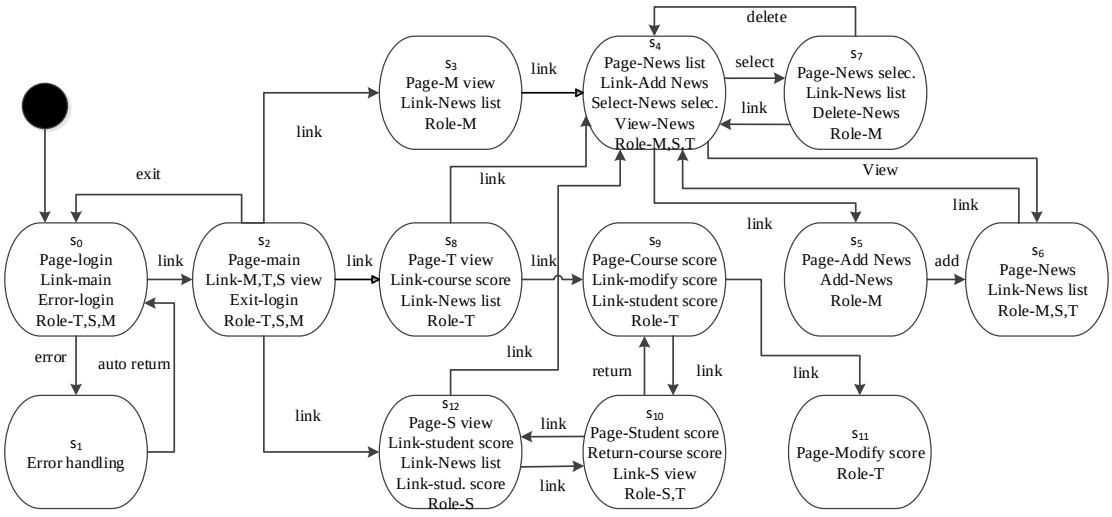

Figure 7. A navigation model of the student course management system

5. Case Study

In this section, we study the implementation of five testability metrics using a case. There is a navigation model of the student course management system [21] shown in Fig. 7. In Fig. 7, some atomic propositions are predefined as follows:

- "Page-<page name>" denotes the page,
- "Role-<role name>" denotes the roles that include Teacher role (T), Student role (S), and Manager role (M),
- "Link-<page name>" denotes the link,
- "Error-<page name>" denotes the login error,
- "Return-<page name>" denotes a returning page, and
- "Error handling" denotes that the system deals with an error input.

Pages and requests in the navigation model of a Web application are alternated. This system includes three user roles: the teacher's role denoted as "Role-T", the student's role denoted as "Role-S", and the manager's role denoted as "Role-M". The teacher's role can record and modify the scores of student course, and check the score of a student. The student's role can only browse their own course scores. The manager's role can browse, add and delete news. In this model, "Role-M,T,S" denotes that the page can be visited by three roles, and "Role-T,S" denotes that the page (state) can be visited by the teacher and student roles. For any role, the state s1 is invisible, and two transitions (s0, error, s1) and (s1, auto return, s0) are also invisible.

To obtain test sequences from the navigation model shown in Fig.7, we construct a test tree shown in Fig. 8 by using the breadth-first search approach. Then, twelve root-to-leafs state sequences regarded as test paths are obtained from this test tree. Assume that the test requirement is branch coverage in this example.

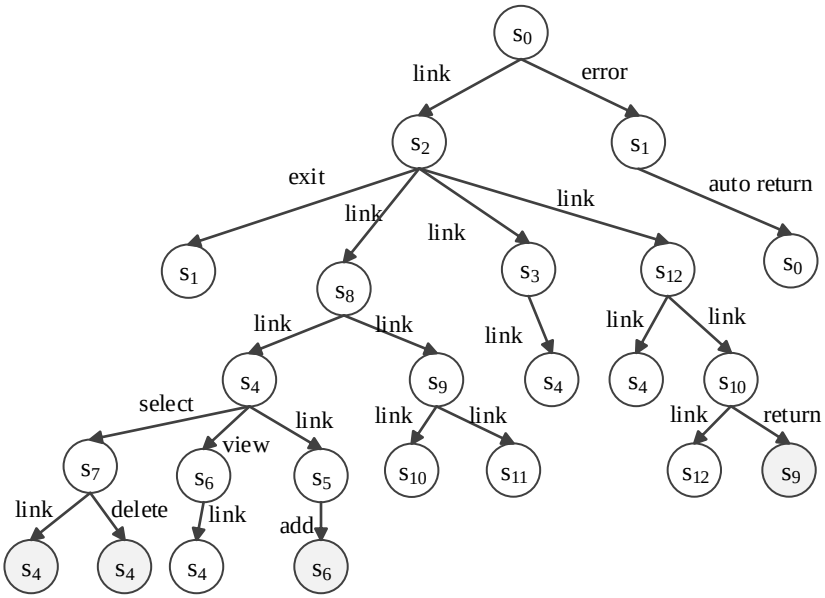

Figure 8. A test tree of the navigation model shown in Fig. 7

Discussion for observability. From Fig. 7, the model of the student course management system contains 13 states where the state s_1 is invisible, and 23 transitions where two transitions (s_0, error, s_1 and s_1, auto return, s_0) cannot be observed. So, observability of the model is *OB*=(12/13+21/23). To improve observability of the model, we recommend changing the state s1 to a visible error page.

Discussion for controllability and test constructibility. From Fig. 8, in the path s_1-s_2-s_{12}-s_{10}-s_9, the role in s_{12} is "Role-S", hence it is impossible to reach s_9 with "Role-M" from s_{10} by the operation *return*. Thus the path s_1-s_2-s_{12}-s_{10}-s_9 is ineffective. For the path s_1-s_2-s_8-s_4-s_5-s_6, since the role of s_8 is "Role-T", it has no authority to access to state s_5 with "Role-M". Therefore the path s_1-s_2-s_8-s_4-s_5-s_6 is infeasible. Similarly, s_8 with "Role-T" also can't go to s_7 with "Role-M" from s_4, hence the path s_1-s_2-s_8-s_4-s_7 is unreachable. Thus, controllability of the navigation model is *CO*=2/3. Suppose that there is adequate time to generate all test sequences. Then β=1 and test constructibility of this navigation model is *TC*=2/3. To improve controllability and test constructability of the navigation model in Fig. 7, we suggest that the model needs to be decomposed based on the user role.

Discussion for performability and error traceability. From Fig. 7, the student course management system must start from the state s_0. Thus, performability of the navigation model is *PE*=1/13×1/23, and error traceability of the navigation model is *ET*=(PE_1+…+ PE_{12})=5/6. If we can add some states that can be directly visited via the initial state, and some transitions that

can be modified so that the expected target state in this model can arrive, both *PE* and *ET* can be increased. So, we recommend separating the login part from the navigation model so that more states can be reached from the initial state of the model.

Based on the above discussions, we decompose the navigation model in Fig. 7 into four models shown in Fig. 9, where Fig. 9 (a) is the login model of the system, Fig. 9 (b) is the model of the manager role, Fig. 9 (c) is the model of the teacher role, and Fig. 9 (d) is the model of the student role. Then, four test trees in Fig. 10 (a), (b), (c), and (d) can be respectively constructed from four models in Fig. 9 (a), (b), (c), and (d).

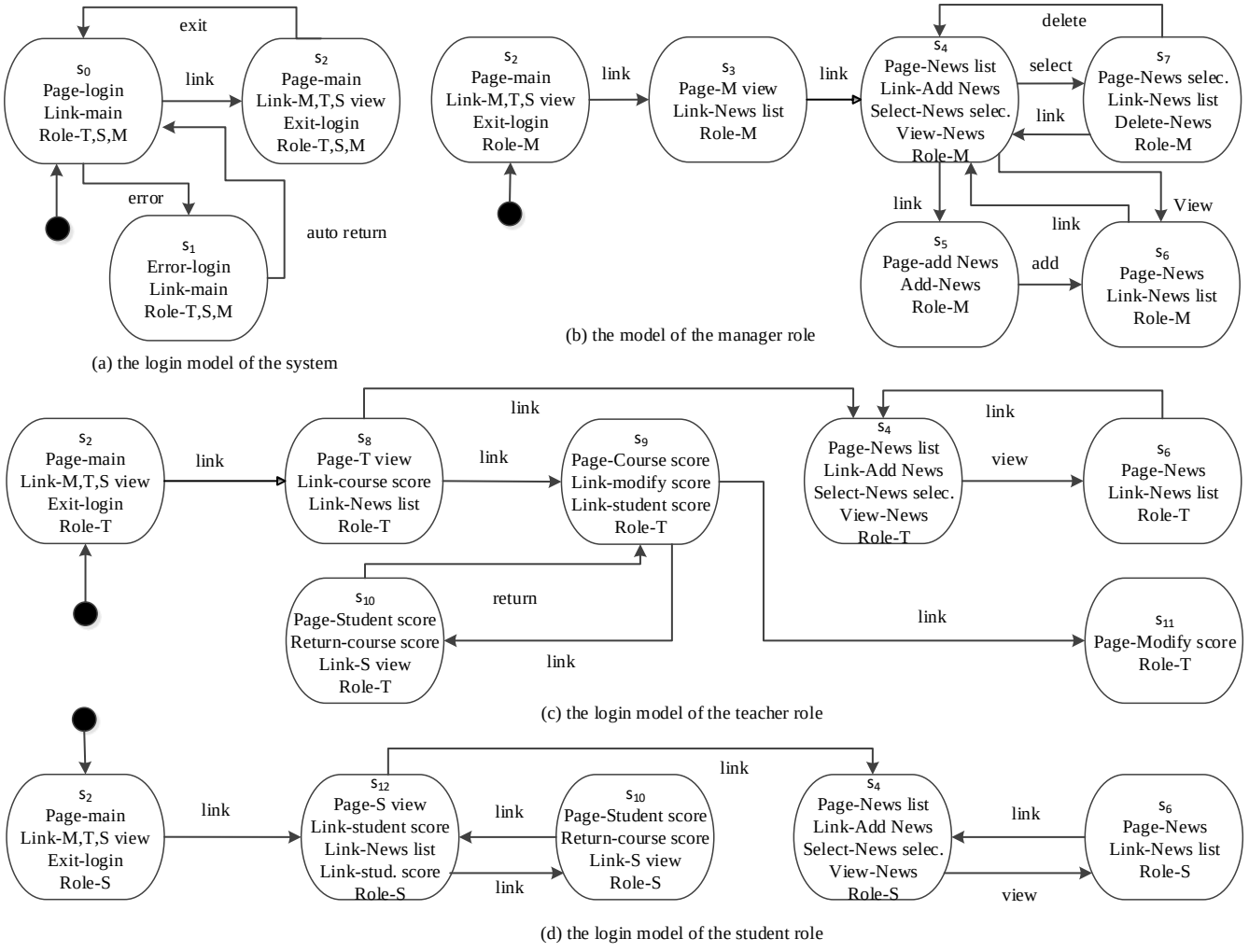

Figure 9. Four navigation sub-models

Evaluation for four new models: All states and transitions in four models are observable; hence, observability of four models is *OB*=1. In addition, three models except for the login model are built based on three different roles, so test paths in each model are executable. Then, both controllability and test constructability of four models are 1. Performability of four models is 1/6, 1/90, 1/72, and 1/35, respectively. Error traceability of four models is 1/3, 1/5, 11/60, and 4/30, respectively. From Fig. 10, we can obtain eleven test paths from four test trees, while we can get twelve test paths from the test tree in Fig. 8. The total length of eleven test paths in Fig. 10 is 38, while that of twelve test paths in Fig. 8 is 46. Moreover, these eleven test paths are executable generated from four test trees. Therefore, compared to the model in Fig. 7, these new models in Fig. 9 can produce fewer test cases with less total length. This indicates that testability metrics for software behavioral model is valid in this study.

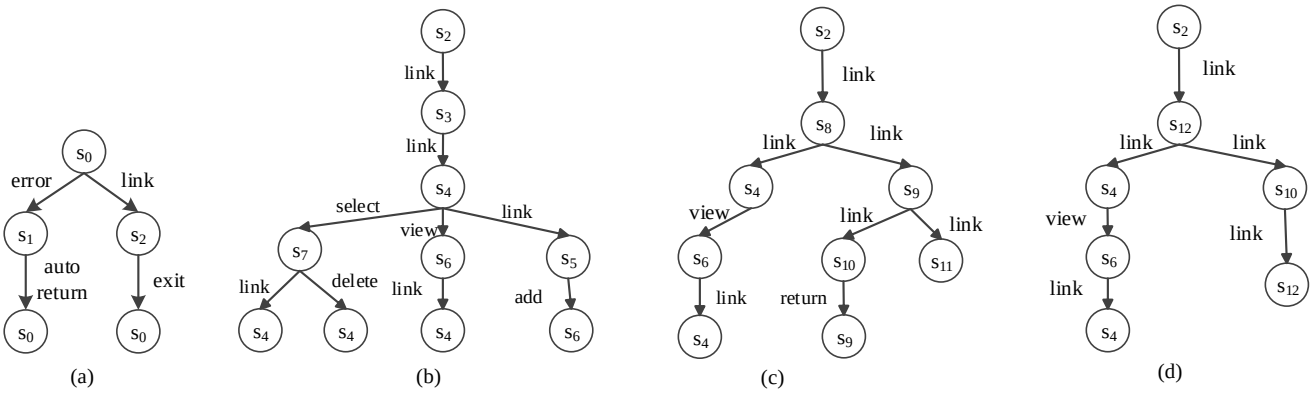

Figure 10. Four test trees for four navigation sub-models, respectively

6. **Conclusions**

Traditionally, software testability has come from the experience of the people, which cannot be assessed by using the quantitative way. The main measures for evaluating software testability only focus on two properties of both observability and controllability. The related assessment is for object-oriented programs and not for behavior models of software. As an effective and efficient test method, in the past decades, more and more researchers have focused on model-based testing. However, few studies address the issues related to testability of models. In this vein, it is difficult to ensure the effectiveness of test sequences generated from the model, resulting in the failure of conformance testing. To improve testability of models, we have presented five testability metrics of software behavioral models so as to generate high-effective test sequences. Utilizing those metrics, we have evaluated testability of a software behavioral model in a case study. The result of the study shows that we can modify models to improve their testability. The main contribution of the paper is to present five testability metrics for software behavioral models, which enriches the modeling theory of model-based testing.

Acknowledgements

This work is supported by National Natural Science Foundation of China (NSFC) under grant (No. 61502299), Science and technology key project of Jiangxi Province (No. 20142BBE50015).

References

1. D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," *IEEE Software*, vol. 32, pp. 53-59, 2015
2. P. Arcaini, A. Gargantini, and E. Riccobene, "Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism," *in proceedings of IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pp. 178-187, Luxembourg, Luxembourg, March 2013
3. B. Baudry and Y. L. Traon, "Measuring design testability of a UML class diagram," *Information and Software Technology*, vol. 47, pp. 859-879, 2005
4. B. Baudry, Y. Le Traon, and G. Sunyé, "Testability analysis of a UML class diagram," *in Proceedings of Eighth IEEE Symposium on Software Metrics*, pp. 54-63, Ottawa, Canada, June 2002
5. B. Baudry, Y. Le Traon, G. Sunyé, and J. Jezequel, "Measuring and improving design patterns testability," *in Proceedings of Ninth International on Software Metrics Symposium*, pp. 50-59, Washington, DC, USA, September 2003
6. A. Bertolino and L. Strigini, "On the use of testability measures for dependability assessment," *Software Engineering, IEEE Transactions on*, vol. 22, pp. 97-108, 1996
7. R. V. Binder, "Design for testability in object-oriented systems," *Communications of the ACM*, vol. 37, pp. 87-101, 1994
8. R. V. Binder, B. Legeard, and A. Kramer, "Model-based testing: where does it stand?," *Communications of the ACM*, vol. 58, 2015
9. M. Bruntink and A. van Deursen, "An empirical study into class testability," *Journal of Systems and Software*, vol. 79, pp. 1219-1232, 2006
10. S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, pp. 476-493, 1994
11. P. Clements, M. J. Escalona, P. Inverardi, et al. "Exploiting software architecture to support requirements satisfaction testing," *in Proceedings of the 19th ACM Sigsoft Symposium and the 13th European Conference on Foundations of Software Engineering*, pp:484-487, Szeged, Hungary, September 2011
12. H.-E. Eriksson and M. Penker, *Business modeling with UML*: Wiley Chichester, 2000
13. N. E. Fenton and S. L. Pfleeger, *Software metrics: a rigorous and practical approach*: PWS Publishing Co., 1998
14. R. S. Freedman, "Testability of software components," *Software Engineering, IEEE Transactions on*, vol. 17, pp. 553-564, 1991

15. A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. van Gemund, "Minimising the preparation cost of runtime testing based on testability metrics," *in Proceedings of IEEE 34th Annual in Computer Software and Applications Conference*, 2010, pp. 419-424, Seoul, Korea, July 2010
16. S. Jungmayr, "Testability measurement and software dependencies," *Shaker*, 2002
17. R. Lai, "A survey of communication protocol testing," *Journal of Systems and Software,* vol. 62, pp. 21-46, 2002
18. D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE,* vol. 84, pp. 1090-1123, 1996
19. P. Liu, H.-K. Miao, H.-W. Zeng, and Y. Liu, "FSM-based testing: Theory, method and evaluation," *Jisuanji Xuebao(Chinese Journal of Computers),* vol. 34, pp. 965-984, 2011
20. P. Liu, H. Miao, "Theory of Test Modeling Based on Regular Expressions," *Structured Object-Oriented Formal Language and Method*. Springer International Publishing, pp:17-31, 2013
21. P. Liu, H. Miao, H. Zeng, and L. Cai, "An Approach to Test Generation for Web Applications," *International Journal of u- and e- Service, Science and Technology,* vol. 6, p. 16, 2013
22. A. Kout, F. Toure, and M. Badri, "An empirical analysis of a testability model for object-oriented programs," *ACM SIGSOFT Software Engineering Notes,* vol. 36, pp. 1-5, 2011
23. T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on,* pp. 308-320, 1976
24. B. A. Nejmeh, "NPATH: A measure of execution path complexity and its applications," *Communications of the ACM,* vol. 31, pp. 188-200, 1988
25. C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: an empirical study conducted in industry," *in Companion Proceedings of the 36th International Conference on Software Engineering*, pp. 135-144, Hyderabad, India, May 2014
26. Y.-N. Shen, F. Lombardi, and A. T. Dahbura, "Protocol conformance testing using multiple UIO sequences," *Communications, IEEE Transactions on,* vol. 40, pp. 1282-1287, 1992
27. J. W. Sheppard and M. Kaufman, "Formal specification of testability metrics in IEEE P1522," *in IEEE Autotestcon Proceedings of IEEE Systems Readiness Technology Conference* , pp. 71-82, Valley Forge, PA, USA, August 2001
28. Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *Software Engineering, IEEE Transactions on,* vol. 37, pp. 772-787, 2011
29. J. Offutt and A. Abdurazik, *Generating tests from UML specifications*: Springer, 1999
30. Y. Le Traon, F. Ouabdesselam, and C. Robach, "Analyzing testability on data flow designs," *in Proceedings of the 11th International Symposium on Software Reliability Engineering,* pp. 162-173, San Jose, CA, USA, Oct. 2000.
31. J. M. Voas and K. W. Miller, "Semantic metrics for software testability," *Journal of Systems and Software,* vol. 20, pp. 207-216, 1993
32. J. M. Voas and K. W. Miller, "Software testability: The new verification," *IEEE software,* vol. 12, pp. 17-28, 1995
33. P.-L. Yeh and J.-C. Lin, "Software testability measurements derived from data flow analysis," *in Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pp. 96-102, Washington, DC, USA, March 1998
34. L. Zhao, "A new approach for software testability analysis," *in Proceedings of the 28th international conference on Software engineering*, pp. 985-988, Shanghai, China, May 2006.

Pan Liu received the MSc degree in computer software and theory from Nanchang University in 2006 and the PhD degree in computer application from Shanghai University in 2011. Now he is an associate professor at College of Information and Computer, Shanghai Business School, Shanghai, China. He is also an associate researcher in Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai, China. His papers have been published in some well-known international Journals and IEEE conferences. His main interests include software testing, model-based testing, formal method, and algorithm design.