# An UML Statechart Diagram-Based MM-Path Generation Approach for Object-Oriented Integration Testing

Ruilian Zhao, and Ling Lin

***Abstract*—**MM-Path, an acronym for Method/Message Path, describes the dynamic interactions between methods in object-oriented systems. This paper discusses the classifications of MM-Path, based on the characteristics of object-oriented software. We categorize it according to the generation reasons, the effect scope and the composition of MM-Path. A formalized representation of MM-Path is also proposed, which has considered the influence of state on response method sequences of messages. .Moreover, an automatic MM-Path generation approach based on UML Statechart diagram has been presented, and the difficulties in identifying and generating MM-Path can be solved. . As a result, it provides a solid foundation for further research on test cases generation based on MM-Path.

***Keywords*—**MM-Path, Message Sequence, Object-Oriented Integration Testing, Response Method Sequence, UML Statechart Diagram.

## I. Introduction

NOWADAYS, object-oriented paradigm has become a popular technology in modern software industry. However, along with the development of object-oriented software, a variety of new challenges for testing appear, compared to testing for procedural software [1, 2]. Although, most unit testing and system testing techniques may also be applicable to object-oriented testing, testing for procedural software and object-oriented software make a great difference in integration testing since objects may interact with each other by unforeseen combinations and invocations. Therefore, it is necessary to explore new and effective object-oriented integration testing technique in theory and practice.

There are two main categories of integration testing strategies. One is based on the functional decomposition tree such as bottom up, top down, sandwich, and "big bang" integration testing strategies. The other is based on the call graph, for instance, pair-wise integration testing and neighborhood integration testing strategies[3]. The call graph-based integration testing is directly applicable to object-oriented software, as Unified Modeling language (UML) provides several dynamic diagrams, such as Statechart diagram, sequence diagram and collaboration diagram to illustrate how class instances are interacted by message invocations [4]. A number of testing approach based on UML diagrams have been developed to detect software faults in recent years [5, 6, 7, 8, 9, 10]. For example, reference [5] identified control flow and data flow by transforming UML Statechart diagram into extended finite state machines (EFSMs), and then generated test cases from flow graphs resulting from EFSMs. Reference [6] first defined the dynamic behavior of components via UML Statecharts, and then test cases were derived from these Statecharts and executed with the help of a test execution tool. Reference [7] introduced an intermediate diagram, called Testing Flow Graph (TFG), which was transformed from UML Statechart diagrams to assist test case generation according to testing coverage criteria of state and transition. In [8], a method to automatically generate test cases based on UML state chart specifications was proposed. There, the conditional predicates on state transitions were transformed, and function minimization technique was applied. Reference [9] used the Category-Partition method to introduce data into a UML model, and described an ongoing research on test case generation based on UML. In [10], authors first analyzed different test elements that were critical to test component-based software. Then they proposed a group of UML-based test elements, test adequacy criteria to test component-based software. Obviously, applying UML concept into object-oriented software testing has become a new trend.

MM-Path, an acronym for Method-Message Path, was proposed by Paul C. Jorgensen in [11] and then a graphical representation approach with respect to MM-Path was introduced in [12]. MM-Path is defined as an interleaved sequence of method execution paths linked by messages, which indicates the interactions between methods in object-oriented systems. Therefore, MM-Path based integration testing technique is more suitable to object-oriented software.

However, traditionally, MM-Path was generated from source code by program instrumentation, which brought a great deal work in order to identify MM-Path in the software under test [3]. This puts obstacles to MM-Path automatic generation, and it seriously restricts the usefulness of MM-Path in integration testing practice although it has many advantages. Hence,

Manuscript received August 31, 2006. This work was supported in part by the China National Natural Science Fund (No.60473032), Science and Technology Emphases Item of China Ministry of Education (NO.105018) and Emphases Laboratory Open Subject of Institute of Software, Chinese Academy of Sciences (ISCAS) Fund (NO.SYSKF0605).

Ruilian Zhao and Ling Lin are with Department of Computer Science, Beijing University of Chemical Technology, Beijing, 100029, China (phone: 86-010-6445-4674, e-mail: rlzhao@mail.buct.edu.cn, xyz_olive@sina.com).

finding a new generation approach for MM-Path is indispensable and meaningful.

More and more software developers use UML and associated visual modeling tools as a basis to design and implement their applications. Furthermore, UML Statechart is widely used for specifying the dynamic behaviors of classes and contains necessary information about state transition that is more propitious to object-oriented software testing. Therefore, in the research reported in this paper, UML Statecharts are used as a basis to generate MM-Path. Firstly, WE discuss a formal representation with respect to MM-Path based on UML Statechart, and then present an approach to generate MM-Path from UML Statechart diagram.

The remainder of the paper is organized as follows. Section 2 outlines a background related to the research, defining MM-Path and introducing UML Statechart diagram. A description of classification with respect to MM-Path is given in Section 3. Section 4 presents a formal representation to atomic and compound MM-Path. Section 5 proposes an approach to generate MM-Path from UML Statechart diagram, which is illustrated by a case study in Section 6. Conclusive remarks and future work are, finally, indicated in Section 7.

## II. Background

### A. MM-Path (Method-Message Path)

MM-Path, defined by P. C. Jorgensen in [11], is descriptive. In order to conveniently implement the automatic generation of MM-Path, we, referred to the description of MM-Path in [11], define a Method-Message Path in this paper as follows.

Definition: Method-Message Path in object-oriented software, called MM-Path for short, is an execution sequence of methods linked by messages, whose form is "Message1-MethodSeq1-Message2-MethodSeq2-…" (MethodSeqi is the response method sequence of $Message_i$ here).

MM-Path describes that messages trigger the execution of methods in object-oriented software. In fact, an MM-Path is just a series of pairs of messages and its response method sequences. It starts from a message that activates a corresponding method to execute, and ends on a method that does not issue any messages from its own.

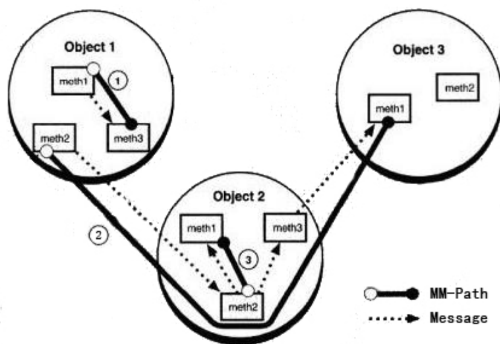


Fig. 1 Example of MM-Path

Fig. 1 is a directed-graph representation of an object network, where dotted line means possible delivering path of messages in this network, and bold line stands for MM-Path. There are three objects and three MM-Paths, denoted by ①, ② and ③, respectively, in Fig. 1. For example, MM-Path② starts when there is a message calling the method *meth2* of the object *Object1*. And then MM-Path② is extended to the method *meth2* of the object *Object2* as there is a message invoking from the *meth2* of *Object1* to the *meth2* of *Object2* during its execution. Similarly, MM-Path② is followed by the method *meth3* of the object *Object2* and the method *meth1* of the object *Object3*. Finally, MM-Path② ends on *meth1* of *Object3* because message quiescence occurs; that is to say, there is no message issue from *meth1* of *Object3*.

Obviously, MM-Path clearly displays the calling relationship between methods. However, due to the interaction complexity in object-oriented software, MM-Path is various. Thus, it is indispensable to discuss the classification, characteristics and formal representation of MM-Path before exploring a MM-Path generation approach.

### B. UML Statechart Diagram

Statechart diagrams in UML are used to describe the dynamic behavior of a class, subsystem or system. The key elements in a UML Statechart diagram are states, transitions, events and actions. States and transitions define all possible states and state changes that an object can achieve during its lifetime. State changes occur as reactions to events received from the object's interfaces. Actions correspond to internal or external method calls.

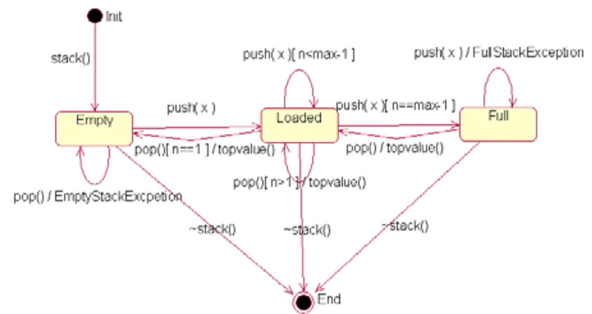


Fig. 2 UML Statechart diagram for a *stack* class

For example, Fig. 2 is the Statechart of a *stack* class. It comprises three states, namely *Empty, Loaded,* and *Full* respectively, as well as a start and an end state, denoted by ● and ◉. There are five operations in *stack* class. They are *stack(), ~stack(), push(), pop() and topvalue().* The transitions from *source* state to *target* state are labeled with "*event [guard condition]/action*", which means that, when *source* state detects the *event* and *guard condition* is also satisfied, a transition will occur by executing *action*.. For instance, the transition "*pop()[n==1]/topvalue()*" from *Loaded* state to *Empty* state, means that, when *pop()* event happens on *Loaded* state, if guard condition "*n==1*" is true, then this transition happens. This is to say, the stack object first executes action

*topvalue()*, and implement the transition from *Loaded* state to *Empty* state.

As delivering messages completes event-driven processes in object-oriented software, the notations of *event* and *message* in UML Statechart diagram are similar. Therefore, we use *message* in Statechart diagram instead of *event* in this research, and focuses on messages and its corresponding transitions in UML Statechart diagram. On a basis of that, we discuss MM-Path formal representation and generation, taking state transition into consideration.

## III. Classification of MM-Path

In order to formally represent MM-Path, we first discuss the classification of MM-Path, according to the generation reasons, the effect scope and the composition with respect to MM-Path.

### *A. The Classification According to the Generation Reason of MM-Path*

Object-Oriented Software has many new characteristics, such as inheritance, polymorphism and so on. All of these characteristics have an impact on interactions among methods and MM-Path generation. In the terms of the generation reasons of MM-Path, MM-Paths can be classified into three categories

- MM-Path based on calling relationship. This kind of MM-Path exists, because of calling relationship between methods resulting from data flow and control flow. For instance, if method A requires operations or data processed by method B during its execution, there exists an MM-Path based on calling relationship. Most MM-Paths fall into this category since calling relationship between methods is very universal in software.
- MM-Path based on inheritance relationship. This kind of MM-Path results from reuse of methods between subclass and parent class. For example, when initializing a subclass, the constructor of its parent class will be executed before its own is executed. Thus, it appears as MM-Path based on inheritance relationship.
- MM-Path based on polymorphism. This kind of MM-Path is produced since base class's interface is shared by subclasses. The most important characteristics of this kind of MM-Path are diversity and indetermination. A message with different parameters may be responded by different methods from different subclasses. The responded method is determined by dynamic binding.

### *B. The Classification according to the Effect Scope of MM-Path*

Object-oriented software meets the requirement of high aggregation and low coupling. Thus, we can sort MM-Path into two categories, according to the effect scope of MM-Path.

- Intra-class MM-Path. The effect scope of intra-class MM-Path is limited inside a class. This kind of MM-Path describes internal methods interactions of a class. For instance, MM-Path ① in Fig. 1 is an intra-class, which only contains methods in *Object1*.
- Inter-class MM-Path. Inter-class MM-Path identifies the method interactions among classes, whose effect scope is involved in several co-operated classes. For instance, MM-Path ② in Fig. 1 is an inter-class MM-Path, which is related to method interactions among three classes *Object1*, *Object2*, and *Object3*. Compared with intra-class MM-Path, the length of inter-class MM-Path is often longer, and the frequency of inter-class MM-Path is usually lower. Apparently, Intra-class MM-Path is an element of inter-class MM-Path.

### *C. The Classification according to the Composition of MM-Path*

MM-Path is a series of pairs of message and corresponding response method sequence. In addition, state is also an important element in object-oriented software. When taking the effect of state on message's response into consideration, MM-Path can be classified into two categories as follows.

- Atomic MM-Path. Atomic MM-Path is defined as a response method sequence of a message in a certain state, with the form similar to "Message-MethodSeq". Atomic MM-Path is an element pair of a MM-Path.
- Compound MM-Path. Compound MM-Path is defined as ordered sequence of atomic MM-Paths. The order here is related to a certain use case. Thus, a certain use case decides the order of message in a compound MM-Path, and each message and its accepted state determine its own atomic MM-Path. As a result, a compound MM-Path is constructed. A compound MM-Path may be an intra-class or inter-class MM-Path.

Take Fig. 1 as an example. MM-Path① and MM-Path③ are atomic MM-Path and intra-class MM-Path, because they just issue one message within one class, while MM-Path② is a compound MM-Path and inter-class MM-Path.

## IV. The Formal Representation of Atomic and Compound MM-Path

In this section, we discuss how to formally represent MM-Path based on atomic MM-Path and compound MM-Path. Firstly, we use *MM* to stand for the set of MM-Path, *MMa* for the set of atomic MM-Path, *MMc* for the set of compound MM-Path. Obviously, the set MM comprises set *MMa* and *MMc*, namely $MM=MMa \cup MMc$.

Atomic MM-Path can be formed as a triplet, *MMa (Msg, n) = (Msg, {State}, MtSeq),* where *Msg* is a accepted message by class instance, including sender, receiver, message name and necessary parameters (guard condition, parameters passed, etc.); *n* records the number of atomic MM-Path corresponding to the message *Msg; State* represents the current state when the message *Msg* is accepted; *MtSeq* is a response method sequence corresponding to the message *Msg*. This expression

of atomic MM-Path means that, when class instance is in any state of set {*State*} and accepts a message *Msg*, it will execute the method sequence *MtSeq*, and this is No. *n* atomic MM-Path for the message *Msg*.

Some explanations for this formal representation are given as follows.

1) Message *Msg* corresponds to message event in UML Statechart diagram.
2) {*State*} is a set of states, including all states, in any of which class instance executes method sequence *MtSeq* after accepting message *Msg*.
3) Method sequence *MtSeq* may contain more than one method.

For example, in Fig. 1, when class instance is in state *Init* and accepts message *stack*, it executes method *stack()*. The atomic MM-Path here can be expressed as *MMa (stack, 1) = (stack, {Init}, stack)*, which is the first atomic MM-Path of message *stack*.

Compound MM-Path is related to both atomic MM-Path and the order of message determined by use case. So we define three operators to represent the order of message as follows.

- "•" stands for sequence relationship. For instance, *(m1•m2)* indicates that message *m2* will be executed after message *m1* being executed.
- "|" stands for switch relationship. For instance, *(m1|m2)* indicates that message *m1* or *m2* will be executed.
- "*" stands for loop relationship. For instance, *m1** indicates that message *m1* may be executed zero, one, or more than one times.

The precedence of these operator is *, |, •, ranging from high to low.

With these operators, if a use case is given, the order of messages can be settled. Each pair of message and its accepted state corresponds to a determined atomic MM-Path, so we can choose a proper one for every message in the message list, by referring to atomic MM-Path list. Thus, we can get the compound MM-Path corresponding to this use case. The detail will be illustrated in Section 5.

## V. Generation of MM-Path

As the discussion above, atomic MM-Path is the element of MM-Path. Therefore, atomic MM-Path generation is the key to MM-Path. As compound MM-Path is related to certain use cases, its generation need to consider other UML diagrams, such as use case diagram, sequence diagram, and so on. Therefore, generation of compound MM-Path will be discussed in other papers. This paper focuses on the algorithm of atomic MM-Path generation.

In order to solve the generation problem precisely, we make several assumptions as follows.

- UML Statechart diagram discussed here is based on Mealy model. Though there are two kinds of state machine model, i.e. Mealy and Moore, they are same in essential and can be transformed from one to another.
- UML Statechart diagram here is deterministic, consistent, and self-contained. There is no sub state, nested state and concurrent state. Only message event might be taken account of.
- MM-Path here is based on calling relationship. The generation based on inheritance relationship and polymorphism will be studied in consequent research.

We also assume that, *M* is the message set of class instance; *Gm* is the set of guard conditions with respect to message *m* ($m \in M$); *MMa (m, p)* stands for the No. *p* atomic MM-Path with respect to message *m*.

Generating atomic MM-Path from UML Statechart diagram comprises the following two steps:

- Extract information from UML Statechart diagram and create message response table.
- Generate atomic MM-Path from message response table.

The detailed descriptions are given in the following sub-sections.

### A. Creating Message Response Table

Message response table, proposed in this paper, is defined to record corresponding information extracted from UML Statechart diagram. There are three columns in it, i.e. *message*, *guard condition* and *accepted state* columns. The *accepted state* column lists all possible state of a class instance. Information in a row indicates corresponding responses with respect to all possible states. Each item in the accepted state is expressed as a form of "*response method sequence/target state*". It means that, when current *message* with certain *guard condition* is accepted in current *state*, *response method sequence* in the item will be executed, and then current state is transformed to *target state*. For example, TABLE I is a message response table, which is constructed from UML Statechart diagram of *stack* in Fig.2.

In order to create message response table, we first find all possible states, pairs of message and guard condition, and record them in the proper place of the message response table. If a message has no guard condition, DC is placed in corresponding position, meaning of "don't consider". After that, check the response information of each pair of message and guard condition. If there is a transition line in UML Statechart diagram, then we record corresponding information in corresponding item. If current message isn't accepted by current state, we set a "×" in corresponding item. More details about the creation of message response table will be illustrated in section 6.

### B. Atomic MM-Path Generation Algorithm

The algorithm of generating atomic MM-Path is given as follows. The input is message response table, and the output is the set of atomic MM-Path identified in the form presented by this paper.

(1) Get message set *M* and guard condition set *Gm,* which are, respectively, corresponding to *message* column and *guard condition* column in message response table.

Similarly, get all possible state set $S$, which can be attained from *accepted state* column in message response table. Let i=j=k=1and state, $s_i(s_i \in S \mid 1 \le i \le |S|)$, guard condition $g_j(g_j \in Gm \mid 1 \le j \le |Gm|)$ , message $m_k(m_k \in M \mid 1 \le k \le |M|)$ . In addition, set all counters $p_k = 0(k = 1,2,...|M|)$ for MMa of message $m_k$

(2) When message $m_k$ with guard condition $g_j$ is accepted in state $s_i$, if MMa of message $m_k$ is none, create an atomic MM-Path, add it into MMa of, $m_k$ and $p_k = p_k+1$. Then, go to (4).

(3) If MMa of message $m_k$ is not empty, search MMa of $m_k$ for method sequence, which is same as current method sequence. If such an atomic MM-Path exists, then add current state $s_i$ into {state} set of this atomic MM-Path. If there not exists such an atomic MM-Path, create an atomic MM-Path, and add it into MMa of $m_k$, and $p_k = p_k+1$.

(4) Let i=i+1, repeat (2), (3), until i=|S|.

(5) Let j=j+1, repeat (2), (3), (4), until j=|Gm|.

(6) Let k=k+1,repeat (2), (3), (4), (5), until k=|M|

## VI. CASE STUDY

A case study has been carried out with the aim of illustrating the representation and algorithm discussed above. We take *stack* class shown in Fig. 2, as an example. Following the rules in Section *V.A*, we can get the message response table as Table I.

TABLE I
MESSAGE RESPONSE TABLE OF STACK CLASS

| Message | Guard Condition | Accepted State | | | | |
|---|---|---|---|---|---|---|
| | | Init | Empty | Loaded | Full | End |
| Stack | DC | /Empty | × | × | × | × |
| ~Stack | DC | × | /End | /End | /End | × |
| Push | DC | × | /Loaded | × | FullStackException()/Full | × |
| Push | n<max-1 | × | × | /Loaded | × | × |
| Push | n==max-1 | × | × | /Full | × | × |
| Pop | DC | × | EmptyStackException()/Empty | × | Topvalue()/Loaded | × |
| Pop | n==1 | × | × | Topvalue()/Empty | × | × |
| Pop | n>1 | × | × | Topvalue()/Loaded | × | × |

Five possible states of *stack* are listed as sub-columns in the column of *accepted state*. All possible pairs of message and guard condition are listed in the left of the table. As we see, message *push* in Fig. 1 has three guard conditions. So there are three pairs of message *push* with different guard conditions in Table I. Similarly, message *pop* also has three pairs corresponding three different guard conditions. A transition line in Fig. 2 corresponds to a non-empty item in Table I. For example, the transition from state *Loaded* to state *Empty* with message *pop*, has a guard condition "*n==1*" and a response method sequence of *topvalue()*. We record such information as "*topvalue()/Empty*", in the item corresponding to state *Loaded* and message *pop* with guard condition "*n==1*".

After applying the algorithm of generating atomic MM-Path in Section *V.B* to Table I, we can get the set of atomic MM-Path of stack class as follows:

MMa (stack,1)=(stack,{Init}, stack)
MMa (~stack,1)=(~stack,{Empty, Loaded, Full}, ~stack)
MMa (push,1)=(push,{Empty, Loaded}, push)
MMa (push,2)=(push,{Full},push•FullStackException)
MMa (pop,1)=(pop,{Empty},pop•EmptyStackException)
MMa (pop,2)=(pop,{Loaded, Full}, pop•topvalue)

From the results above, we can obtain six atomic MM-Paths of *stack* class, one for message *stack*, one for message *~stack*, two for message *push*, and two for message *pop*. For example, *MMa(push,1)* means that when stack instance accepts message *push* in state *Empty* or *Loaded*, operation *push* will be executed.

With the atomic MM-Paths above, we can give a formal representation of compound MM-Path related to certain use case. For example, if there is a use case of stack in Fig. 2, whose message order can be expressed as *stack•(push*|pop*)*•~stack*, the corresponding compound MM-Path is *MMa(stack,1)•((MMa(push,1)|MMa(push,2)) *|(MMa(pop,1)|MMa(pop,2)) *)*•MMa(~stack,1).*

A conclusion can be drawn from above case study that atomic MM-Path generation from UML Statechart diagrams has extremely simplified the identifition and generation of atomic MM-Path. During the process of generating MM-Path, we needn't care software code, while just concentrate on the UML Statechart Diagrams. Combined to certain use cases, test paths and test cases can be developed on a basis of the results above to test object-oriented software.

## VII. CONCLUSION AND FUTURE WORK

In this paper, such problems as classification of MM-Path, formal representation of MM-Path and generation of MM-Path from UML Statechart diagram, have been solved. We also give a case study to illustrate the representation and generation algorithm presented above. In all, the work above makes a foundation for object-oriented integration testing based on MM-Path.

In our further work, following aspects will be studied: (1) MM-Path generation based on inheritance relationship and polymorphism; (2) realization of automatic generation tool of MM-Path; (3) test case generation method based on MM-Path, combining to UML diagrams; (4) coverage analysis of testing based on MM-Path.

## REFERENCES


[1] W. K. Chan, T. Y. Chen, T. H. Tse, "An Overview of Integration Testing Techniques for Object-Oriented Programs," Proceeding of the 2nd ACIS Annual International Conference on Computer and Information Science (ICIS 2002), 2002

[2] R.V. Binder, *Testing Object-Oriented Systems-Models, Patterns, and Tools*, Addison-Wesley, pp.63-107, 1999

[3] Paul C. Jorgensen, *Software Testing: A Craftsman's Approach (Second Edition)*, Simplified Chinese language edition, China Machine Press,pp. 187-210, 2000

[4] James Rumbaugh, Ivar Jacobson, Grady Booch, *The Unified Modeling Language Reference Manual,* Addison-Wesley, pp. 85-110, 1999

[5] Y. K. Kim, H. S. Hong, D. H. Bae, "Test Cases Generation From UML State Diagrams," IEEE Proc. Software, Vol. 146, No. 4, 1999

[6] Jean Hartmann, Claudio Imoberdorf, Michael Meisinger, "UML-Based integration testing," Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis, pp. 60-70, 2000

[7] Supaporn Kansomkeat, Wanchai Rivepiboon, "Automated-generating Test Case Using UML Statechart Diagrams," Proceedings of the 2003

annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, pp.296-300, Sep. 2003

[8] Philip Samuel, Rajib Mall,"Boundary Value Testing based on UML Models," Proceedings of the 14th Asian Test Symposium (ATS'05), pp.94-99,Dec. 2005

[9] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, Juergen Kazmeier,"Automation of GUI testing using a model-driven approach," Proceedings of the 2006 international workshop on Automation of software test, pp. 9-14, May. 2006

[10] Y. Wu, M. Chen, J. Offutt, "UML-based Integration Testing for Component-Based Software," 2nd International Conference on COTS-Based Software Systems (ICCBSS), Ottawa, pp.251-260, 2003

[11] Paul C. Jorgensen, Carl Erickson, Object-Oriented Integration Testing, Communications of ACM, Vol.37,No. 9, pp. 30-38, 1994

[12] Di Lucca G A, Fasolino A R, Carlini U D. “Recovering Use Case Models from Object-oriented Code: A Thread-based Approach,” In Proc. of 7th Working Conf. on Reverse Engineering (WCRE'00), IEEE Computer SocietyPress,pp.108-177,2000

**ZHAO RuiLian**, received her B.S. and M.S. degree in computer science from North China Industry University in 1985 and 1990, and Ph.D. degree in computer science from Institute of Computing Technology, Chinese Academy of Sciences in 2001. She is now a professor at Department of Computer Science, Beijing University of Chemical Technology. Her current research interests include software testing and fault-tolerant computing.