

Deriving Software Statistical Testing Model from UML Model

Yan Jiong[†], Wang Ji[‡], and Chen Huowang

National Laboratory for Parallel and Distributed Processing, Changsha, P.R. China

[†] E-mail: yanjiong7172@vip.sina.com

[‡] E-mail: jiwang@mail.edu.cn

Abstract

Software statistical testing is concerned with testing the entire software systems based on their usage models. In the context of UML-based development, it is desired that usage models can be derived from UML analysis artifacts. This paper presents a method that derives the software usage models from reasonably constrained UML artifacts. The method utilizes use case diagrams, sequence diagrams and the execution probability of each sequence diagram in its associated use case. By projecting the messages in sequence diagrams onto the objects under test, the method elicits messages and their occurrence probabilities for generating the usage model of each use case for the objects under test. Then the usage models of use cases are integrated into the system usage model. The integration procedure will utilize the execution sequential relations between use cases.

1. Introduction

Software statistical testing is concerned with testing the entire software systems based on their usage models, and is also called reliability testing, for it generates test cases from the software usage models and thus can be used to estimate software reliability. UML-based software development has been the defacto standard development method [1], so it is desired to integrate UML software development with statistical testing. This paper gives our attempt to derive the software usage models from UML analysis artifacts.

There are still not many research works on statistical testing for UML-based development. Current studies on object-oriented software testing are mainly concerned with class testing, and few are applied to industrial applications. [2] proposes to combine message sequences from sequence diagrams with category-partition testing. [3] adapts the traditional data-flow coverage criteria as test adequacy criteria in the context

of UML collaboration diagrams. [4] considers sequential and dependability relationships between use cases and derives system test requirements from use cases and sequence diagrams. [5] transforms use cases into UML state charts and automatically generates system level test suites with a given coverage level from these state charts by mapping state chart elements to the STRIPS planning language. [6] presents an approach to generating system-level test cases based on usage cases and refined by state diagrams, and transforms these models into usage models to describe the system behavior and usage. Because of UML's flexibility, some variants in the models' content are unavoidable and thus impede the testing automation [2].

It should be noticed that most projects modeled with UML produce use case diagrams, sequence diagrams associated with each use case, and class diagrams. Whereas these artifacts are often not testable for they haven't provided sufficient information to allow automatic generation of test cases [7]. Therefore it is necessary to impose testability constraints on UML artifacts to support statistical testing. On the other hand, Markov chain model, which has been proved to work well in practice [8], is adopted to describe software usage model for statistical test case generation, can also be used for test planning and reliability estimation. Because of the complexity of large systems, large numbers of test cases are needed in statistical testing. Thus those testing methods requiring frequent and complex manual tasks are not likely to be widely used.

In this paper, we present a method to derive the Markov chain usage model from reasonably constrained UML artifacts. The method facilitates statistical testing of the software system whose use case execution sequences are strictly predefined, and can be applied to reactive software statistical testing, whose executions are often state-based.

The paper is organized as follows. In Section 2 we give a case of Satellite Control System, which is used as the running example throughout the whole paper. Section 3 presents the details of the method for deriving the usage model from UML artifacts. In Section 4, we

discuss statistical test case generation and an automation tool, and the conclusions are drawn in Section 5.

2. Satellite Control System Case

The UML model of software system includes use case diagrams, sequence diagrams associated with each use case, and class diagrams. Here we take the example of Satellite Control System to illustrate our method for deriving software statistical usage model from these diagrams.

2.1. System Use Cases

[9] provides a satellite communication system, which consists of four components:

1. The ground control system (GCS) initiates and terminates connections, and monitors the satellite health.
2. The satellite processes commands from GCS and supplies half-duplex communications between two ground sites.
3. The uplink site (UL) transmits data packets to the satellite when connected.
4. The downlink site (DL) receives data packets from UL through the connection supplied by the satellite.

We here address only the software component of the satellite, that is, the Satellite Control System (SCS). The work procedure of SCS can be summarized as follows.

GCS must initialize SCS with command IN, and SCS replies with INA. Then GCS must command SCS to enter maintenance mode with MG command and request the satellite health with command HR. If HA is replied from the satellite, GCS will send FR and BR to adjust the satellite orbit and modify the B/L table. After maintenance, GCS will put SCS into the connection mode. GCS then sends UL and DL information to SCS in a TG message. Then SCS forwards TG to both UL and DL with TGF. The two sites complete the connection by sending UG and DG. When both sites are connected, then SCS sends SDT command to UL, which begins to send data packets (DI). SCS forwards all packets from UL to DL (DO). Once DL detects a corrupted packet, it will send a PB message with packet identifier to SCS, SCS will notify UL to resend the packet. After sending all packets, UL will send a TE message to SCS, TE is forwarded to DL and GCS, and thus the connection is closed. The use case diagram in Figure 1 describes the functionality of SCS.

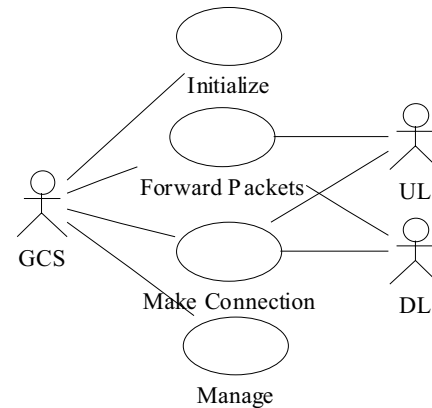


Figure 1. Use case diagram of SCS

2.2. Scenarios in Use Cases

Use case *Initialize* is performed to initialize SCS before its usage and also deals with unrecoverable SCS errors. It is always the first use case performed, which includes only one scenario. In use case *Initialize*, GCS initializes SCS with command IN, and SCS replies with INA.

Use case *Manage* is performed to accomplish maintenance. For the sake of simplicity, we just classify the scenarios of use case *Manage* into two groups. One completes all the required maintenance commands successfully and the satellite is ready for connection, while another group fails and SCS must be reinitialized.

In the basic scenario of use case *Make Connection*, a connection is successfully made among SCS, UL and DL. First, GCS sends SCS the information of involved UL and DL; the two sites reply SCS with UG and DG respectively, indicating a successful connection (Figure 2).

Other scenarios of *Make Connection* are following.

1. UL sends SCS a UB (uplink bad) acknowledgment message indicating that UL is not ready to transmit data to SCS (Figure 3).
2. DL sends SCS a DB (downlink bad) acknowledgment message indicating that DL is not ready to receive data from SCS (Figure 4).
3. Because one or both sites did not have an entry in the B/L table, the connection can't be made between UL and DL, and SCS must be reinitialized (Figure 5).

The use case *Forward Packets* is started after *Make Connection* is successfully completed and the connections among UL, DL and SCS are established. The scenarios of this use case can be classified into the successful transmission and the exception. The former

proceeds until all packets are forwarded and the latter makes SCS must be reinitialized.

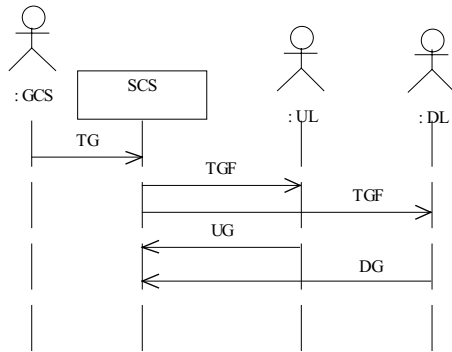


Figure 2. **Scenario 1 of use case Make Connection**

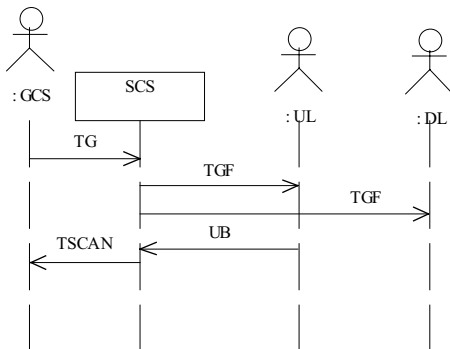


Figure 3. **Scenario 2 of use case Make Connection**

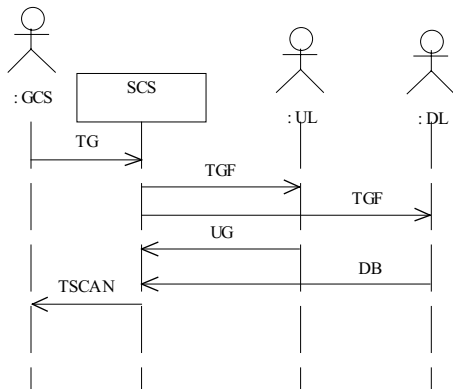


Figure 4. **Scenario 3 of use case Make Connection**

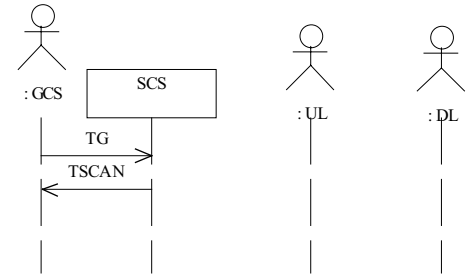


Figure 5. **Scenario 4 of use case Make Connection**

3. Deriving Usage Models from UML Models

Markov chain can be used to describe the usage of software, and thus is used in statistical testing as usage model. A Markov chain is inherently an automaton with transition probability, while execution, the next state is determined by current state and probabilities of each exit transition of current state, without reference to past states. UML-based statistical testing includes the following steps.

1. Acquire the execution probability of each sequence diagram in its associated use case, which is also the execution probability of messages in this sequence diagram. Projecting the messages onto the objects of the system under test, we can elicit the messages for generating usage model.
2. Integrate the usage models of use cases into the system usage model. The integration algorithm will use the execution sequential relations between use cases.
3. Test the system with test cases generated from the software usage model. Having adequately tested the software, testers can measure the software reliability from the test results [9, 10].

Figure 6 summarizes the steps of generating UML-based statistical testing models presented in this paper, thus provides a systematic method to automate the UML-based statistical testing.

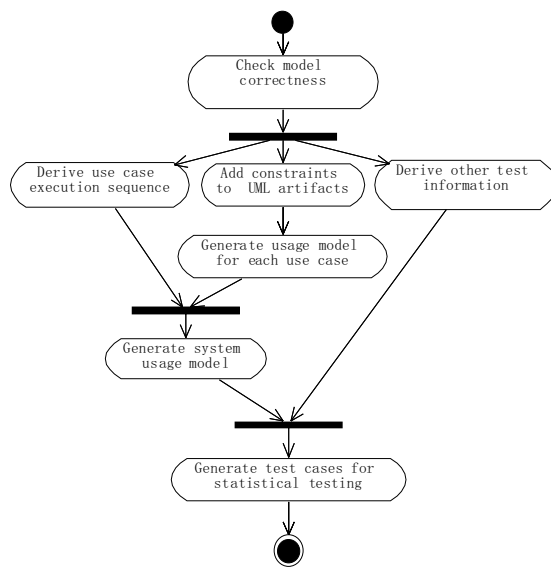


Figure 6. UML-based statistical testing

3.1. Execution Sequential Relations of Use Cases

Use cases describe high-level functionalities of a system and, generally speaking, can be developed independently. Not only use cases have <<extend>> and <<include>> relationships but also the execution sequential relations which reflect the business process the system supports [7, 11]. Usage-based software statistical testing must consider the execution sequential relations because different use case execution sequences may trigger different failures. We represent execution sequential relations between use cases by activity diagrams [4, 11]. In an activity diagram, vertices are use cases and the edges are the execution sequential relations between the use cases. The use cases that have no relations can execute in parallel. The *join* and *fork* synchronization bars are used to present the synchronization among use cases. To execute a use case, some specific preconditions must be satisfied, and some specific postconditions will arise after execution. Consequently, the preconditions and postconditions may determine which use case will be executed next.

Figure 7 describes the execution sequential relations between use cases of SCS. SCS must perform use case *Initialize* before working. The successful initialization makes system state changed into *Initialized* and SCS comes to execute use case *Manage*. Successful management changes the system state into *Managed*, which enables use case *Make Connection*, while the unsuccessful management into *ManageFail* and SCS must be reinitialized. Successful connection makes SCS

enter state *Connected* and data packets can be transmitted, while the unsuccessful connection *ConnectFail*, and SCS must be reinitialized. During transmitting, SCS forwards every packet from UL to DL until all packets are forwarded successfully or something inextricable happens. The former ends use case *Forward Packets* and leads SCS to state *Idle* while the latter makes the SCS enter state *TransmitFail* and reinitialized.

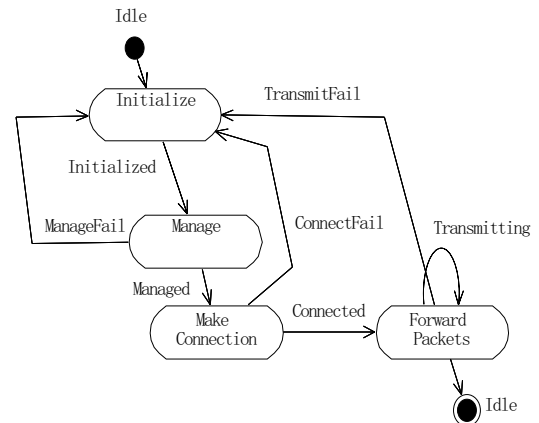


Figure 7. Sequential dependencies of SCS use cases

3.2. Adding Constraints to UML Model

To make UML models testable, we must add constraints to UML artifacts. These constraints are indispensable for deriving usage models. In this paper, the following constraints are added.

1. The conditions that must be satisfied before a use case executes, expressed in OCL [12].
2. The states that the system enters after one use case completed execution. Because different scenarios may lead to different states, and states expressed in OCL are added for each sequence diagram of each use case.
3. The execution probability of each scenario in its associated use case.

Those constraints can be added during system analysis and design, and will greatly increase testability of the system, facilitate understanding of the system without too much modification to UML artifacts, thus can be easily adopted by system analysts and designers.

Take the example of use case *Make Connection*, the OCL constraints are described as follows.

context SCS --in SCS context

-- Conditions before a use case execution

Self. Make Connection. Pre = Managed

-- States entered after scenario execution finishes

Self. Make Connection. $Scn_1.Post = Connected$
 Self. Make Connection. $Scn_2.Post = ConnectFail$
 Self. Make Connection. $Scn_3.Post = ConnectFail$
 Self. Make Connection. $Scn_4.Post = ConnectFail$

The states of SCS are defined with OCL as follows.

State: **enum** {Idle, Initialized, Managed, ManageFail, Connected, ConnectFail, Transmitting, TransmitFail}

3.3. Deriving Single Use Case's Usage Model

During statistical testing, the execution of a use case can be described with a Markov chain because the execution can be considered as a system state transition process guided by the transition probabilities of exit arcs from each state.

3.3.1. Constrained use cases. Analysis of UML-based development usually produces use cases and sequence diagrams associated with each use case. Each scenario is described with a sequence diagram. Adding constraints to use cases and scenarios, we will get the following constrained use cases.

A use case \mathcal{U} is defined as $\langle Pr, Scns \rangle$, where

- Pr is the condition set that must be satisfied before \mathcal{U} executes.
- $Scns$ is the scenario set of \mathcal{U} , and each scenario is described with a sequence diagram.

Each scenario is define as $\langle S_{num}, Msgs, pf_{num}, Po \rangle$, where

- S_{num} is the scenario number.
- $Msgs$ is the message set of the scenario.
- pf_{num} is the execution probability of this scenario, which is also the execution probability of each message in this scenario.
- Po is the state that the system will enter after the scenario completes execution.

Each message is defined as $\langle Msg, Snd, Rcv \rangle$, where

- Msg is the message name.
- Snd is the message sender.
- Rcv is the message receiver.

Take the example of use case *Make Connection*.

Assuming that the execution probabilities of the four scenarios are pf_1 , pf_2 , pf_3 and pf_4 , notice that $pf_1 + pf_2 + pf_3 + pf_4 = 1$. We thus can describe this use case as (We have replaced the OCL constraints' variables with corresponding values):

$\langle Managed,$
 $\langle Scn_1, \langle \langle TG, GCS, SCS \rangle, \langle TGF, SCS, UL \rangle,$
 $\langle TGF, SCS, DL \rangle, \langle UG, UL, SCS \rangle,$
 $\langle DG, DL, SCS \rangle \rangle,$
 $pf_1, Connected \rangle,$
 $\langle Scn_2, \langle \langle TG, GCS, SCS \rangle, \langle TGF, SCS, UL \rangle,$

$\langle TGF, SCS, DL \rangle, \langle UB, UL, SCS \rangle,$
 $\langle TSCAN, SCS, GCS \rangle \rangle,$
 $pf_2, ConnectFail \rangle,$
 $\langle Scn_3, \langle \langle TG, GCS, SCS \rangle, \langle TGF, SCS, UL \rangle,$
 $\langle TGF, SCS, DL \rangle, \langle UG, UL, SCS \rangle,$
 $\langle DB, DL, SCS \rangle, \langle TSCAN, SCS, GCS \rangle \rangle,$
 $pf_3, ConnectFail \rangle,$
 $\langle Scn_4, \langle \langle TG, GCS, SCS \rangle, \langle TSCAN, SCS, GCS \rangle \rangle,$
 $pf_4, ConnectFail \rangle$

>

3.3.2. Deriving usage model for single use case.

During statistical testing, the usage model of each use case can be described as a Markov chain, whereas in UML each use case is modeled with a collection of sequence diagrams. The usage model of a use case \mathcal{U} can be defined as $\langle S, \Sigma, \delta, q_0, F \rangle$, where

- S is a set of states of execution of \mathcal{U} .
- Σ is a set of transition labels, where each label corresponds to a message.
- δ is the transition relation: $S \times \Sigma \times [0,1] \rightarrow S$.
- F is a set of final states. The final states of use case \mathcal{U} are the states that the system enters after the completion of its execution.
- q_0 is the initial state that the system must satisfy before executing \mathcal{U} .

Each state is defined as $\langle State_{num}, Notation \rangle$, where

- $State_{num}$ is the state number.
- $Notation$ is the information attached to each state.

Figure 8 shows the Markov chain of use case *Make Connection*, which is derived by the algorithm presented in Figure 9. It is worth noting that different constraints added to UML artifacts may call for a different algorithm for use case usage model derivation. For example, if each message in sequence diagrams is attached with a precondition and a postcondition, a more elaborate usage model will be derived [13]. Whereas too much testability constraints may prevent the method from industrial acceptance, we add only relatively weak constraints to the artifacts but it is enough for usage model derivation.

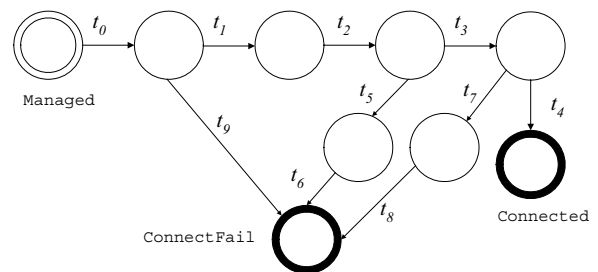


Figure 8. Markov chain of use case *Make Connection*

Input: Constrained use cases and sequence diagrams.

Step 1: Create Markov chain for the first sequence diagram

Create a Markov chain with initial state S_0 ;

$pf \leftarrow$ Scenario execution profile; $CurrentState \leftarrow S_0$;

FOR each message m in current scenario

Insert a new state S ;

IF m is message that the objects under test received THEN

Add new transition $t \leftarrow \langle \text{event}, m, pf \rangle$;

ELSE

Add new transition $t \leftarrow \langle \text{action}, m, pf \rangle$;

ENDIF

LinkStates ($CurrentState, S, t$); // link state $CurrentState$ and S with transition t

$CurrentState \leftarrow S$;

ENDFOR

Let the notation of S be the current scenario's postcondition;

Step 2: Integrate other sequence diagrams to derive Markov chain for a use case

FOR each of the other scenarios

$CurrentState \leftarrow S_0$; $pf \leftarrow$ current scenario execution probability;

FOR each message m in current scenario

IF m is message that the objects under test received THEN

Add new transition $t \leftarrow \langle \text{event}, m, pf \rangle$;

ELSE

Add new transition $t \leftarrow \langle \text{action}, m, pf \rangle$;

ENDIF

IF (there is a transition tr exists in the chain is same as t with exception of pf)

AND (SourceStateOf (tr) = $CurrentState$) THEN

Accumulate pf to the transition probability of tr ;

$CurrentState \leftarrow$ target state of t ;

ELSE

Insert a new state S ;

LinkStates ($CurrentState, S, t$);

$CurrentState \leftarrow S$;

ENDIF

ENDFOR

IF current scenario's postcondition exists in the notations of the final states of the chain THEN

Let S_{final} be the final state in the chain whose notation is the same as current scenario postcondition;

Unify S and S_{final} ;

ELSE

Let the notation of S be the current scenario's postcondition;

ENDIF

ENDFOR

Step 3: Normalize transition probabilities for each transition attached to each state.

Figure 9. Algorithm for translating a use case into a Markov chain

Transitions in Figure 8 are described as follows.

$t_0 = \langle \text{event}, TG, 1 \rangle$

$t_1 = \langle \text{action}, TGF, pf_1 + pf_2 + pf_3 \rangle$

$t_2 = \langle \text{action}, TGF, 1 \rangle$

$t_3 = \langle \text{event}, UG, (pf_1 + pf_3) / (pf_1 + pf_2 + pf_3) \rangle$

$t_4 = \langle \text{event}, DG, pf_1 / (pf_1 + pf_3) \rangle$

$t_5 = \langle \text{event}, UB, pf_2 / (pf_1 + pf_2 + pf_3) \rangle$

$t_6 = \langle \text{action}, TSCAN, 1 \rangle$

$t_7 = \langle \text{event}, DB, pf_3 / (pf_1 + pf_3) \rangle$

$t_8 = \langle \text{action}, TSCAN, 1 \rangle$

$t_9 = \langle \text{action}, TSCAN, pf_4 \rangle$

It is well known that the transition probabilities of a usage model are hard to acquire because the granularity of transition maybe too small to sample and record. That is why we utilize the execution probability of each scenario in its associated use case, which is relatively

easy to acquire. The execution probability of the scenario in its associated use case is the execution probability of each message in the sequence diagram of the scenario. The probability of each message is mapped and accumulated to each transition, thus we get the transition probability of each different transition.

3.4. Generating System Usage Model

Having obtained the Markov chain of each use case, we can now combine the chains by unifying the preconditions of the use cases with the states that the system enters after the scenarios execution. The algorithm in Figure 10 shows how to integrate the Markov chains of use cases to generate the entire system usage model.

```

FOR each final state  $S_{out}$  of each use case's Markov chain
  FOR each use case's precondition  $S_{enter}$ 
    IF  $S_{out} = S_{enter}$  THEN
      Unify  $S_{out}$  and  $S_{enter}$  ;
    ENDIF
  ENDFOR
ENDFOR

```

Figure 10. Algorithm for generating the entire system usage model

4. Test Case Generation

Markov chain usage model-based statistical testing supports not only automatic generation of test cases but also the measurement of software quality and test adequacy.

4.1. Test Case Generation and Testing

In statistical testing, test cases are generated automatically by traversing the states of usage model, guided by the transition probabilities associated with the exit arcs from each state [9]. Each arc of the usage model is attached with a particular stimulus (or response) of the system. While traversing the usage model derived with the algorithms presented in this paper, only the arcs labeled with 'event' are accumulated to the traverse path because they represent the stimuli that the system receives. Each path from the initial state to a final state is a specific test case. Testing continues until statistical testing adequacy criteria are satisfied. If testing cost and project schedule are main concerns, testing continues until test resources available are used up.

4.2. Automation

In order to automate usage models derivation from UML models, we have developed the tool UMGGen (Usage Model Generator). The tool elicits UML model information by API provided by UML modeling tool or XMI representation of the model. OCL parser is used to parse the constraints imposed on use cases and sequence diagrams. With the model information and the OCL constraints, UMGGen uses the algorithms presented in this paper to generate software system usage model.

5. Conclusion and Future Work

This paper has discussed UML-based usage model derivation, and presented a method to derive the usage model from early artifacts of software development. The early use of analysis artifacts is very important as it helps devise statistical test plan and plan resources early in the life cycle. The main interesting features of our approach can be summarized in three points. The first concerns adding the constraints of preconditions to use cases and postconditions to scenarios, thus makes the UML model testable. The second emphasizes the execution sequential relations between use cases, which reflect the usage of reactive software and affect the derivation of the usage model. The third is to map the execution probabilities of scenarios to the transition probabilities of the usage model, and thus gives a new approach to the operation profile.

The ongoing and future work includes,

1. How to incorporate time constraints of sequence diagrams into usage models. This is the key problem of real-time software statistical testing.
2. How to deal with the systems whose execution between use cases are non-deterministic. These systems often include use cases with same preconditions. In order to derive system usage models, we must solve the problem of transition determination.
3. How to incorporate other information of UML artifacts such as class diagrams and deployment diagrams into usage models to produce complete statistical test cases for distributed software systems.

Acknowledgement

This work is supported by National Natural Science Foundation of China Grants No. 90104007 and No. 60233020, 863 Hi-Tech Program of China Grants No. 2001AA113202 and No. 2001AA113190, and Huo Ying Dong Education Foundation Grant No. 71064.

References

- [1] Fowler, M. and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Massachusetts, Addison Wesley, 1997.
- [2] F. Basanieri and A. Bertolino, "A Practical Approach to UML-Based Derivation of Integration Tests", *Proc. 4th International Software Quality Week Europe*, Brussels, Nov. 2000, pp. 20-24.
- [3] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", *UML'00, LNCS 1939*, Springer, 2000, pp. 383-395.
- [4] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing", Carleton University TR SCR-01-01-Version 2, 2002.
- [5] Peter Frohlich and Johannes Link, "Automated Test Case Generation from Dynamic Models", *LNCS 1850*, Springer, 2000, pp. 472-491.
- [6] Matthias Riebisch, Ilka Philippow and Marco Götze, "UML-Based Statistical Test Case Generation", *LNCS 2591*, Springer, 2003, pp. 394-411.
- [7] Binder, R., *Testing Object-Oriented Systems*, Addison-Wesley, 1999.
- [8] J.H. Poore, "Introduction to the Special Issue on: Model-Based Statistical Testing of Software Intensive Systems", *Information and Software Technology*, Vol.42, 2000, pp. 797-799.
- [9] Prowell, S.J., C.J. Trammell, R.C. Linger, and J.H. Poore, *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999.
- [10] Lyu, M.R., *Handbook of Software Reliability Engineering*, McGraw-Hill Companies, 1996.
- [11] Bruegge, B. and A.H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall, 2000.
- [12] Warmer, J. and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.
- [13] Jon Whittle and Johann Schumann, "Generating Statechart Designs From Scenarios", *Proceedings of the 22nd International Conference on Software Engineering*, 2000, pp.314-323.