Generating Test Cases from UML Activity Diagram based on Gray-Box Method*

Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuandong and Zheng Guoliang
State Key Laboratory of Novel Software Technology
Department of Computer Science and Technology, Nanjing University
Jiangsu, Nanjing, P.R.China 210093
wanglz@seg.nju.edu.cn

Abstract

Test case generation is the most important part of the testing efforts, the automation of specification based test case generation needs formal or semi-formal specifications. As a semi-formal modelling language, UML is widely used to describe analysis and design specifications by both academia and industry, thus UML models become the sources of test generation naturally. Test cases are usually generated from the requirement or the code while the design is seldom concerned, this paper proposes an approach to generate test cases directly from UML activity diagram using gray-box method, where the design is reused to avoid the cost of test model creation. In this approach, test scenarios are directly derived from the activity diagram modelling an operation. Then all the information for test case generation, i.e. input/output sequence and parameters, the constraint conditions and expected object method sequence, is extracted from each test scenario. At last, the possible values of all the input/output parameters could be generated by applying category-partition method, and test suite could be systematically generated to find the inconsistency between the implementation and the design. A prototype tool named UMLTGF has been developed to support the above process.

Keywords: *gray-box method, UML activity diagram, test scenario, test case*

1. Introduction

Testing is an important part of quality control in the Software life-cycle. As the complexity and size of software grow, the time and effort required to do sufficient testing grow. Manual testing is time-consuming and error-prone. Therefore it is pressing to automate the testing effort. The testing effort can be divided into three parts: test case generation, test execution, and test evaluation. The latter two parts are relatively easy to be automated provided that the criteria for passing the tests are available. However, to determine which tests are required to achieve a certain level of confidence is not trivial. Test case generation in practice is still performed manually most of the time, since automatic test case generation approaches require formal or semi-formal specification to select test case to detect faults in the code implementation.

As the virtual standard of modelling language, UML [1] provides life-cycle support in software development, and is widely used to describe analysis and design specifications of software by both academia and industry. This brings great advantages to software development, but also challenges to study the test generation from UML specification. Using the UML models as inputs for the test case generation process is a natural idea, but how to derive tests from UML analysis and design specification is still a pressing problem to be solved. UML activity diagram[1, 2] describes the sequential or concurrent control flow between activities. Activity diagram can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. Activity diagram emphasizes on the activities of the object, so it is the perfect one to describe the realization of the operation in the design phase, and to describe the sequence of the activities among the involving objects in the control flow during the implementation of an operation, the relationship between the activity and the object in the message flow, the state change of object in the object flow as the executing of activity. What these modelling elements in the activity diagram represent are different aspects of system information, which are essential information of the system and must be preserved from design to implementation of the SUT(System Under Test) [3, 4]. UML becomes more and more pervasively applied in the industry, but there are relatively few practical approaches and tools

* Supported by the National Natural Science Foundation of China (No.60233020 and No.60273036), the National 863 High-Tech Programme of China (No.2002AA116090), and by the National Grand Fundamental Research 973 Program of China (No.2002CB312001).

support deriving test cases from models in analysis and design phases. Research and practice in software testing generate test from the specification based on black-box method, or from the code based on white-box method, but few researches were done on generating test from design. The design models are intermediate artifacts between requirement specification and final code implementations of system under develop, they preserve the essential information from analysis models, and are the basis of code implementation. We hope to work out a method which can generate test directly from UML design models, and can be partly automated so as not to impose too much workload on the user.

The rest of the paper is organized as follows. Section 2 briefly introduces the syntax of UML activity diagrams, and formally defines the semantics of the test-oriented activity diagram. Section 3 overviews the grey-box testing method. The test coverage criteria and test scenario are also defined in this section. Section 4 proposes an approach to generate test cases from UML activity diagram using gray-box method. A prototype tool supporting the presented approach is described in section 5. Finally, the last section discusses the related works and contains some conclusion.

2. UML activity diagrams

UML activity diagrams extract the core idea from flow chart, state transition graph, and Petri net[3]. Its modelling elements consist of nodes and edges. The nodes represent processes or process control, including action states, activity states, decisions, swim lanes, forks, joins, objects, signal senders and receivers. The edges represent the occurring sequence of activities, objects involving the activity, including control flows, message flows and signal flows. Activity states and action states are denoted with round cornered boxes. Transitions are shown as arrows. Branches are shown as diamonds with one incoming arrow and multiple exit arrows each labelled with a boolean expression to be satisfied to choose the branch. Forks or joins are shown by multiple arrows entering or leaving a synchronization bar. Activity diagram can be used to model the work flow of business system or the complex behavior of an operation. This paper focuses on UML activity diagrams which model the operations to generate test cases. Figure 1 shows a UML activity diagram for an operation of withdrawing money from an ATM [6]. It consists of the most modelling elements of activity diagram to describe an operation.

In order to automatically analyze the activity diagram to extract essential information to derive test cases, we formalize it as follows.

Definition 1 An activity diagram is a tuple $D = (A, T, F, C, a_I, a_F)$, where

1. $A = \{a_1, a_2, \cdots, a_m\}$ is a finite set of activity states;

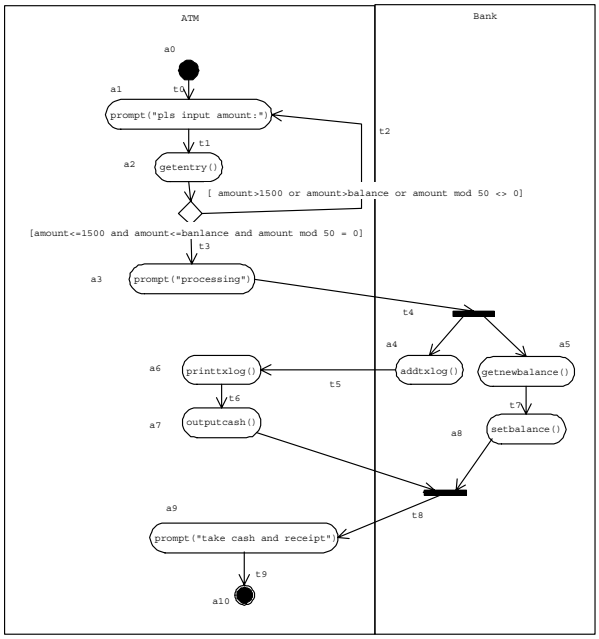

Figure 1. An activity diagram of ATM

2. $T = \{t_1, t_2, \cdots, t_n\}$ is a finite set of completion transitions;
3. $C = \{c_1, c_2, \cdots, c_n\}$ is a finite set of guard conditions, and C_i is in the corresponding transition t_i;
4. $F \subseteq (A \times T \times C) \cup (T \times C \times A)$ is the flow relationship between the activities and transitions;
5. $a_I \in A$ is the initial activity state, $a_F \in A$ is the final activity state, there is only one transition t such that $(a_I, t) \in F$,and $(t', a_I) \notin F$ and $(a_F, t') \notin F$ for any t'.

When an activity diagram runs, at any time its current state (denoted by CS) is represented by a set of activity states.

Definition 2 Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram, CS be the current state of D, for any transition $t \in T$, let

1. ${}^\bullet t, t^\bullet$ denote preset and postset of t respectively, then ${}^\bullet t = \{a \in A \mid (a, t) \in F\}$ and $t^\bullet = \{a \in A \mid (t, a) \in F\}$;
2. $enabled(CS)$ denotes the set of firable transitions from CS, then $enabled(CS) = \{t \mid {}^\bullet t \subseteq CS$ are all completed and $c(t)$ is satisfied$\}$;
3. $fired(CS)$ denotes the only firable transition from CS at certain moment, then $fired(CS) = \{t \mid t \in enabled(CS)$ and $(CS - {}^\bullet t) \cap t^\bullet = \emptyset\}$, and after t

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

was fired, the new current state $CS' = (CS -^{\bullet} t) \cup t^{\bullet}$, if there are more than one transition satisfy the condition, we can randomly choose one which is still unfired.

4. ep denotes an executing path of D in runtime, then $ep = CS_0 \xrightarrow{t_0} CS_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} CS_n$ is a sequence of states and transitions, $CS_0 = \{a_I\}, CS_n = \{a_F\}$, CS_i is current state, and $t_i = fired(CS_i), i \geq 0; CS_i = (CS_{i-1} -^{\bullet} t_{i-1}) \cup t^{\bullet}_{i-1}), i \geq 1$.

Based on above definition, we can parse the activity diagram and represent the essential information in a computer-readable format, so as to be automatically processed.

3. Gray-box method

Black box testing method generates tests from system specifications in the user's viewpoint, it only validates whether the functions specified in the system requirement specifications were implemented or not. It needs no information about how the system was implemented, and does not take into account the developing method and programming language adopted[7]. On the contrary, white box testing method, which is based on the programmer's viewpoint, creates program flow charts from the code implementation by reverse engineering using software comprehension and analysis techniques. Then it analyzes the program structure to generate test cases[8]. These two methods are effective and practical, but they also have shortcomings respectively which can not be overcome by consummating the methods themselves. In the object-oriented context, the design pattern and structure of software differentiate with those of procedure-oriented structural system, but the traditional testing method could not adapt to such changes[9]. Gray-box testing method,which was proposed in recent years[10] in the designer's viewpoint, generates test cases based on high level design models which represent the expected structure and behavior of the SUT. The design specifications are the intermediate artifact between requirement specification and final code. They preserve the essential information from the requirement, and are the basis of the code implementation. Gray box method combines the white box method and the black box method. It extends the logical coverage criteria of white box method and finds all the possible paths from the design model which describes the expected behavior of an operation. Then it generates test cases which can satisfy the path conditions by black box method. It can find problems which used to be ignored by both black and white method.

An activity diagram modelling an operation describes the expected behaviors of that operation. The incorrect implementation of the activity diagram will result in unexpected behaviors of the operation. If there is any defect in the implementations of the operation modelled in the activity diagram, it must be in one ep of the activity diagram defined in section 2. Since we have no idea about in which ep the defect could be, we have to check all the possible eps of the activity diagram. As the result, either we reach the defect and correct the error, or the implementation is proved to be consistent with the design.

To traverse all the eps satisfying the test requirement, general method is to transform the activity diagrams to flow charts, and then traverse the graph to achieve path coverage. In this paper, in order to avoid the cost of complex graph transformation and impossible path traverse, we try to get these paths directly from the activity diagram. An activity diagram which represents the implementation of an operation is just like the flow chart of code implementation of the corresponding operation, an ep of the activity diagram is a possible executing trace of a thread of the program which implements the operation in runtime. Dynamic testing is to run the program in its possible executing scenario against certain inputs, and to evaluate the real outputs with expected results so as to determine its conformance to the specification. The executing trace of thread during the testing is just the implementation of the ep in the corresponding activity diagram,which can be found in the activity diagram.

To do sufficient testing at code level, path coverage is the best choice[6], which means to exercise all possible executing traces of the thread of the operation in runtime. To design the test using gray-box method, we should find all the eps of the activity diagram. However, we could not perform exhaustive testing of all the eps in practice. Because full combination of branches and loops could result in path explosion, and in the extreme circumstance, complex activity diagram could have huge amount of paths. So in this paper, in order to exercise possible and effective paths, the basic path is defined as follows.

Definition 3 When we traverse an activity diagram from the initial activity state to the final activity state by DFS(Depth First Search method), we restrict that the loops be executed at most one time, and all action states and transitions be covered. Thus we can get all **basic paths**.

The number of the basic paths is usually acceptable in practice, so we define the basic path based coverage criteria of gray box method as the test completion criteria .

Definition 4 Coverage criteria: Let BP be the basic path set of an activity diagram, tc be the set of test cases, for any $p_1 \in BP$, there must be at least one test case $t \in tc$ such that when the software is executed using t, the basic path p_1 of the activity diagram is executed.

We can get the maximum number of basic paths of the activity diagram by cycle complexity [8]. That is the mini-

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

mum number of test cases to satisfy the gray box test criteria, and the BP can be the guidance to generate test cases. Any ep in an activity diagram is a complete executing trace of a thread implementing the operation, and also is a trace of the interactions among the involving objects. BP is the subset of the set of all the eps. Every ep represents a run scenario of SUT, in the view of testing in this paper, we name it as a testing scenario, which is made up of activity states and transitions including the guard condition in the executing sequence of the control flow, from the initial activity state to the final activity state. In this paper, we formalize it as follow.

Definition 5 Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram, TS the set of test scenarios of D, $ts \in TS$ is a sequence of activity states and conditioned transitions, then $ts = CS_0 \stackrel{[c_0]t_0}{\longrightarrow} CS_1 \stackrel{[c_1]t_1}{\longrightarrow} \cdots \stackrel{[c_{n-1}]t_{n-1}}{\longrightarrow} CS_n$, where,$CS_0 = \{a_I\}, CS_n = \{a_F\}$, CS_i is current state, and $t_i = fired(CS_i)$, C_i is the guard condition on t_i, $i \geq 0; CS_i = (CS_{i-1} -^{\bullet} t_{i-1}) \cup t_{i-1}^{\bullet}), i \geq 1;$

In the above definition, if all the conditions of a test scenario, i.e. $c_0, c_1, \cdots, c_{n-1}$, are set,we can get an ep. An ep is an implementation of a test scenario in runtime. If one ep is set for each test scenario of TS, BP can be derived from TS. The number of the test scenarios of an activity diagram must be finite, because it is equal to the number of the elements of the BP of the activity diagram. So we can find all the test scenarios more easily than all the eps. The purpose of testing is to validate whether the implementation preserve the semantics of design, that is to verify the consistency between the code implementation and the behavior specified in the activity diagram. A test scenario is an expected execute scenario of the operation, also is a conditioned run of the activity diagram which models the operation. When all the guard condition on transitions of the test scenario sequence are satisfied, an ep implements a test scenario of the activity diagram. The purpose of test case generation is to find the proper inputs which satisfy the path conditions and can run the SUT in the test scenario so as to reach the potential error in corresponding path.

4. Generating test cases from activity diagrams using gray-box method

The verification of models themselves is done by informal review and formal model checking[5], and it is out of our concern in this paper. In order to directly reuse the activity diagrams modelling an operation as the test model to generate test cases, it is necessary to follow the testability requirement. An activity diagram only has one initial activity state, pair of branches and merges, pair of forks and joins. The owner object of each activity state should be labelled by swim lanes, or be labelled in the name of activity state. Every node other than the initial node and final node has at least one outgoing edge and one incoming edge, which means all nodes are reachable. Any fork node only has two exit edges. Concurrent activity states will not access the same object and only execute asynchronously. In order to focus on the test case generation, we suppose that the operation represented in the activity diagram be consistent with the specification of use case.

Based on these assumptions, gray-box method could systematically generate test cases directly from the activity diagrams which can be used to test the system at code level. Firstly, it parses the activity diagram and derives the set of test scenarios to satisfy the basic path coverage criteria. Then, each test scenario is processed. The input and output parameters are extracted from the action sequence. The constraint conditions are extracted from the guard conditions in each transition of the test scenario sequence. The object method sequence which represents the internal behavior of the software in runtime is extracted from activity states and corresponding objects. At last we can use category partition method [11] to generate proper combination of values of input and output parameters to satisfy the condition constraints. So the input sequence, expected object method call sequence and expected output form a test case. And we could also generate all test cases to form the test suite for the activity diagrams.

4.1. Deriving test scenarios from activity diagram

To traverse all the possible basic paths of an activity diagram, we comply with the criteria described in section 3, fully combine the conditional branches and forks, and execute the loops at most one time. Based on the definition in section 2, we can represent the essential information of the activity diagram in intermediate data structures. As an example, table 1 shows guard condition, preset and post set of each transition of the activity diagram in Figure 1.

Based on these program readable representations of the activity diagram, we design an algorithm **TsGenerator** to automatically traverse all the activity states and transitions of each basic path with the retrospective DFS method, so as to get all the test scenarios from the activity diagram. From the initial activity state($CS[0] = \{a_I\}$) to the final activity state($CS[n] = \{a_F\}$), **TsGenerator** visits the current state $CS[i]$ and transitions fired from it in turn. A stack s is employed to sequentially record the visiting trace of CSs and transitions, which is also the trace of a run of the activity diagram. After the algorithm is initialized, $CS[i]$ is pushed into the top of s, and its occurring time in the stack is set in the flag array. When the $enabled(CS[i])$ is not empty, one firable transition t is chosen and fired from $CS[i]$, and its occurring time in the stack is set in the flag array. After that t is deleted from $enabled(CS[i])$. t and the cor-

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

t	${}^\bullet t$	$c(t)$	$t^\bullet$
t_0	a_0		a_1
t_1	a_1		a_2
t_2	a_2	$amount > 1500$ or $amount > balance$ or $amount\ mod\ 50 \neq 0$	a_1
t_3	a_2	$amount \leq 1500$ and $amount \leq balance$ and $amount\ mod\ 50 = 0$	a_3
t_4	a_3		a_4, a_5
t_5	a_4		a_6
t_6	a_6		a_7
t_7	a_5		a_8
t_8	a_7, a_8		a_9
t_9	a_9		a_{10}

Table 1. The transition transformation relationships

responding guard conditions are pushed into top of s. The new state set $CS[i+1]$ could then be calculated by deleting the prestate of t from $CS[i]$ and merging the poststate of t into $CS[i]$. If a loop has been executed once, i.e. the occurring times of $CS[i]$ in s equal to two, it is bypassed in the sequence. If $CS[i]$ in the top of the stack s is empty, a full path is completed. We can read out a test scenario from the bottom to the top of the stack into a test scenario $ts[j$. Then the algorithm backtracks to the last visited $CS[i]$ with an unvisited firable transition in the $enabled(CS[i])$ and continues the traverse. This progress continues until all the current states set and transitions of the activity diagram were found at least one time in the set of test scenarios. The pseudocode algorithm is presented in figure 2.

By applying the above algorithm, we could get a set of test scenarios. In fact we have applied it to the activity diagram in figure 1. Table 2 shows the relations between the current states and fired transitions. One of six derived test scenarios from the above activity diagram is as follows.
ts1:$\{a_0\}t_0\{a_1\}t_1\{a_2\}[amount > 1500$ or $amount > balance$ or $amount\ mod\ 50 \neq 0]t_2\{a_1\}t_1\{a_2\}$ $[amount \leq 1500$ and $amount \leq balance$ and $amount\ mod\ 50 = 0]t_3\{a_3\}t_4\{a_4, a_5\}t_5\{a_5, a_6\}t_6\{a_5, a_7\}$ $t_7\{a_7, a_8\}t_8\{a_9\}t_9\{a_{10}\}$;

After all the test scenarios have been found from the activity diagram,we can now work on generating test cases from the set of test scenarios.

4.2. Generating test cases for each test scenario

A test case of gray-box testing is made up of the possible input sequence, object method executing sequence and out-

```
TsGenerator(D)
//Input:D = (A, T, F, C, a_I, a_F);
//visted_CS[] is an array of the visited times of CS
//visted_t[] is an array of the visited times of t
//s is a stack to record the visited CS and fired t
//output:ts[n];
begin
i := 0; j := 1; ts := null; s := null; visited_CS := 0
visited_T := 0; CS[i] := {a_I};
do while CS[i] ≠ NULL
   push(CS[i], s);
   visited_CS[CS[i]] := visited_CS[CS[i]] + 1;
   if enabled(CS[i]) ≠ NULL;
      visitnexttransition(CS[i]);
      Loop;
   else
      Read out the stack s from bottom to top to ts[j];
      j := j + 1;
      if all the CS[i] and t at least shown one time in ts
         exit;
      else
         do while s ≠ NULL
            t := pop(s); CS[i] := pop(s);
            if enbaled(CS[i]) ≠ NULL
               visitnexttransition(CS[i]);
            endif
            Loop;
         enddo
      endif
   endif
   Loop;
enddo
end
visitnexttransition(CS[i])
begin
t := next unvisited transition in enabled(CS[i]);
enabled(CS[i]) := enabled(CS[i]) \ t;
CS[i + 1] = (CS[i] − preset(t)) ∪ postset(t);
i := i + 1;
if visited_CS[CS[i]] == 2
   visitnexttransition(CS[i]);
endif
push(c[t]t, s);
visited_t[t] := visited_t[t] + 1;
end;
```

Figure 2. The algorithm to generate test scenarios from an activity diagram

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

i	$CS[i]$	$enabled(CS[i])$	$fired(CS[i])$	$CS[i+1]$
1	a_0	t_0	t_0	a_1
2	a_1	t_1	t_1	a_2
3	a_2	t_2, t_3	t_2	a_1
			t_3	a_3
4	a_3	t_4	t_4	a_4, a_5
5	a_4, a_5	t_5, t_7	t_5	a_5, a_6
			t_7	a_4, a_8
6	a_5, a_6	t_6, t_7	t_6	a_5, a_7
			t_7	a_6, a_8
7	a_4, a_8	t_5	t_5	a_6, a_8
8	a_5, a_7	t_7	t_7	a_7, a_8
9	a_6, a_8	t_6	t_6	a_7, a_8
10	a_7, a_8	t_8	t_8	a_9
11	a_9	t_9	t_9	a_{10}
12	a_{10}			

Table 2. The current state and transition relationships

ID	Input seq.	expected method seq.	expected output seq.
tc_1	28	$ATM.prompt()$,	Please input amount,
		$ATM.getentry()$,	
		$ATM.prompt()$,	Please input amount,
	150	$ATM.getentry()$,	
		$ATM.prompt()$,	Processing,
		$bank.addtxlog()$,	
		$ATM.printtxlog()$,	Print the note,
		$bank.getnewbalance()$,	
		$ATM.outputcash()$,	Output cash 150,
		$bank.setbalance()$,	
		$ATM.prompt()$	Take your cash and receipt

Table 3. The test case for the test scenario ts_1

put sequence. Given a test scenario, the sequence of its action states and scenario conditions on its transitions could be extracted. ¿From the swim lane in the activity diagram, we could also find the object of each activity state. Thus the sequence of action states can be transformed to the expected executing sequence of object methods. The input parameters could be found in the input action states of the test scenario. The input sequence and output sequence could be extracted from the sequence of corresponding input and output action states of the test scenario. All the guard conditions in the transitions of the test scenario are extracted as the path condition. The expected output can be extracted from the output action states corresponding to the specification and guard conditions. At last, referring to the input domain, output domain and constraints between them in the requirement specification of the SUT, the rational combination of input values and output values are generated by category partition method. At least one test case could be generated from one test scenario. Table 3 lists one test case tc_1 derived from the test scenario $ts1$.

Test cases for all the test scenarios could be generated to form the test suite. And the redundant and impossible test cases should be eliminated from the test suite. We can also incrementally generate test cases for other activity diagrams of the same use case using above gray box testing method. We can use the instrumentation method in [16] to instrument the SUT against each test scenario. After executing the SUT with these test cases, we can collect and compare the results with the expected results in the test cases to find the inconsistency between the design and the implementation. If a test case fails, there must be a defect in one ep, which is the implementation of corresponding test scenario of the activity diagram.

5. A prototype tool

To support the automation of above test case generation method, we have developed a tool named UMLTGF, which automatically generates test cases from UML activity diagrams. UMLTGF can easily import and parse the UML specifications(called MDL plain text file). UMLTGF can systematically generate test cases for each test scenarios derived from the activity diagram. The test suite can also be managed to be reused in the future. There are three use cases as shown in the use case diagram in figure 3. The class diagram of UMLTGF is in figure 4. The detail functions of the three use case are as follows.

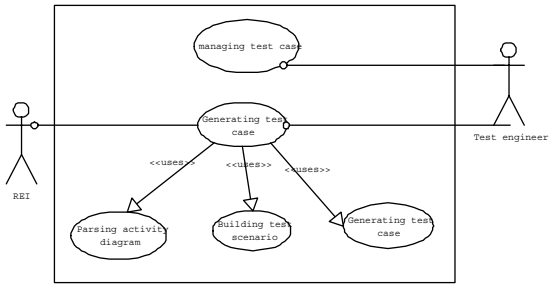

Figure 3. The use case diagram of UMLTGF

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

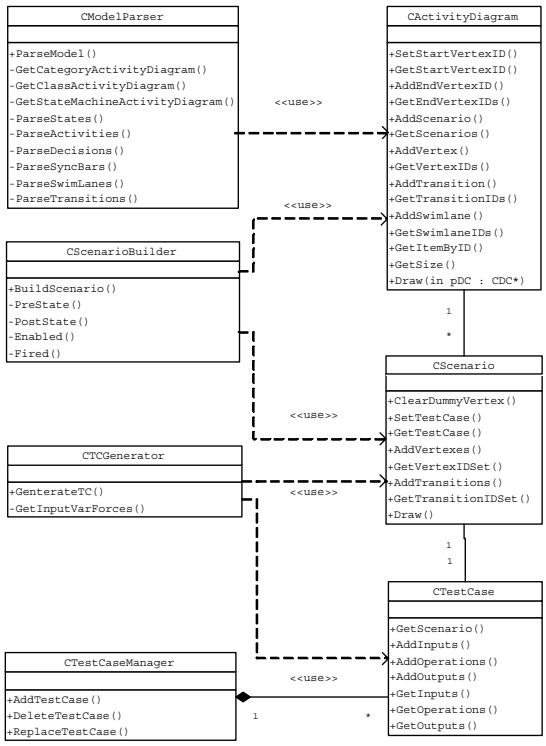

Figure 4. The class diagram of UMLTGF

Abiding by the XMI specification of OMG, **UML model parser** can import the activity diagrams directly from UML modelling tools and analyze the text file of specification (Rational Rose MDL file) with the help of Rose Extensibility Interface[12], then extract the activity states and the transitions information and store them in the data structure so as to be accessed by the test case generator. **Gray-box test case generator** mainly includes test scenario generator and test case generator. The former one analyzes the semantics of the result of the model parser, and derives test scenarios satisfying the gray method criteria, while the latter analyzes each test scenario and generates test case. **Test case manager** manages the generated test cases. It can add, modify, delete the test case to reuse, reduce, maintain the test suite.

The activity diagram in figure 1 was imported and processed by UMLTGF. Then the result was shown in the panel in figure 5. The tree of test scenarios was listed in the top-left of the panel. Once a test scenario was selected, the labelled sequence of the activities and transitions would be represented in the top-right of the panel, and the final test case of selected test scenario would be described in the bottom of the panel.

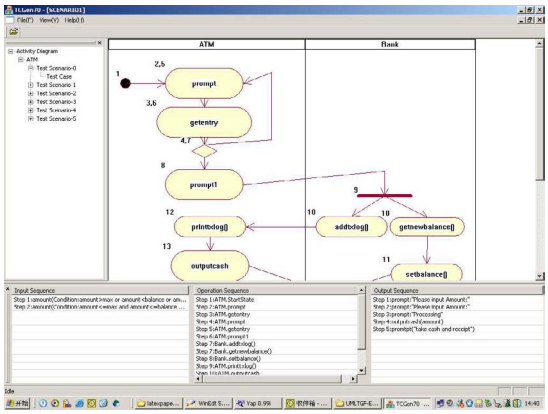

Figure 5. The interface of UMLTGF

6. Related works and conclusion

Even though UML is widely employed in industry and research, only a little part of the reported literatures has addressed its use in the testing phase so far. These methods generate test cases for various testing levels from various UML diagrams. An approach for generating test cases satisfying different coverage criteria from UML state chart is described in [12]. In [13], the state charts are transformed to global finite state machine(GFSM) from which the integration test cases are generated. A number of approaches for generating test cases indirectly from the UML analysis and design models, i.e. use case diagram, sequence diagram, collaboration diagram and class diagram, are proposed in [14, 15, 16, 17]. But most of them need to translate UML description into another formal description and then derive the test from the latter. [18, 19] introduce the approach to generate test from UML activity diagram. In [18], an UML activity diagram is formalized and transformed to a test case model, and the test case could be generated from the test case model. This approach need to considerate transformation cost. A strategy is proposed to derive test scenarios from activity diagrams and to generate test cases from the test scenario in [19], but it only proposes a conceptual idea, and does not give a systematic method. An approach proposed in [20] formalizes the activity diagram to a more rigorous formal activity diagram(FAD), and represents the interactions between the user and the SUT in FAD, then generates formatted test cases from FAD on the basis of the state machine testing method. However it only focuses on the inputs and outputs based on black box method, and does not consider complex behaviors.

The approach proposed in this paper differentiates with the above mentioned methods mainly as follows. Firstly, test cases could be generated directly from UML activity diagrams systematically. Secondly, our method is com-

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE

pletely based on UML models. Thirdly, the most part of this method could be automated. In our opinion, it will be adopted by industry easily.

It makes full advantage of the black-box method to analyze the expected external behavior, and the white-box method to covers the internal structure of the activity diagram of the SUT to generate test cases. And it is also helpful to find defects in the implementation such as over-implementation and under-implementation, which can not be found by the test cases only generated from the code itself. Using this method, test cases could be generated before or parallel with the code implementation, since design based testing is started up as soon as the design phase ends. It enables testers to reasonably employ the test resources. Directly reusing the design model for generating test cases avoids the cost of building test models or transforming models. Defects in design model could also be detected during the analysis of the model itself. In this case, the defect could be removed as early as possible, thus reducing the cost of defect removal. It could also prevent the testers from being prematurely involved in details of the implementation, so as to reuse the tests generated from essential information of specifications for system migration. It is consistent with the idea of Model Driven Architecture proposed by the Object Manage Group[21], and also is the basis of model driven testing[22], which is our purpose in the future.

Acknowledgement We are grateful to Dr. Peiyu for his critical feedback and constructive suggestions.

References

[1] Object Management Group,UML Specification 1.5, available at http://www.omg.org/uml, 2003

[2] Grade Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 2001

[3] Grade Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language Reference Manual, Addison-Wesley, 2001

[4] Philippe Kruchten, The Rational Unified Process -An Introduction, 2nd edition, Addison-Wesley, Reading, MA, 2000

[5] Cui Meng, Li Xuan-dong, Zheng Guo-liang, Formal Analysis on UML Real-time Activity Diagram, Chinese Journal of Computers, vol. 3,2004(in Chinese)

[6] Robert V. Binder, Testing Object-oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 2000

[7] Beizer. Black-box Testing:Techniques for functional testing of software and systems, John Wiley & Sons,Inc, New York, 1995

[8] Paul C. Jorgrnsen, Software Testing: A Craftsman's Approach , CRC PressInc 1995

[9] Imran Bashir, Amrit L. Goel, Testing Object-oriented Software: Life Cycle Solution, Springer-Verlag New York, Inc, 1999

[10] Hung Q.NguyenTesting Application on the Web:Test Planning for Internet-Based SystemsJohn Wiley & Sons2003

[11] Thoms J. Ostrand, Marc J. Balcer, The Category-Partition Method for Specifying and Generating Functional Tests, Communication of ACM,Vol. 31,No. 6, June 1988

[12] A. J. Offutt and A. Abdurazik, Generating Tests from UML specifications, Proc. 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, pp. 416-429, October, 1999.

[13] Hartmann, J., Imoberdof, C., Meisenger, M., UML-Based Integration Testing, in ISSTA 2000 conference proceeding, Portland, Oregon, 22-25 August 2000, pp. 60-70.

[14] Byoungju Choi, Hoijin Yoon, Jin-Ok Jeon, A UML-based Test Model for Component Integration Test, Workshop on Software Architecture and Component, Japan, pp63-70, Dec. 1999

[15] Basanieri, F., Bertolino, A.: A Practical Approach to UML-based Derivation of Integration Tests. Proceeding of QWE2000, Bruxelles, November 20-24, 3T., 2000

[16] A. J. Offutt and A. Abdurazik, "Using UML Collaboration Diagrams for Static Checking and Test Generation," Proc. 3rd International Conference on the Unified Modeling Language (UML'00), York, UK, pp. 383-395, October, 2000.

[17] Ye Wu, Mei-Hwa Chen and Jeff Offutt, "UML-based Integration Testing for Component-based Software", The 2nd International Conference on COTS-Based Software Systems (ICCBSS). pages 251-260, Ottawa, Canada, February 2003.

[18] Zhang Mei, Liu Chao, Sun Chang-ai, Automated Test Case Generation Based on UML Activity Diagram Model, Journal of Beijing University of Aeronautics and Astronautics(in Chinese), pp.433 437, vol. 27 No. 4, August 2001,

[19] Liu Min, Jin Maozhong, Liu Chao, Design of Testing Scenario Generation Based on UML Activity Diagram, The Engineering and Application of Computer,(in Chinese), Vol. 12,pp.122 124, 2001

[20] Chris Rudram, Generating Test Cases from UML, University of Sheffield ,technique report, available at http://www.dcs.shef.ac.uk2003.

[21] Anneke Kleppe, Jos Warmer, Wim Bast, MDA Explained: The Model Driven Architecture: practice and promise, Addison-Wesley, 2003

[22] R. Heckel, M. Lohmann, Towards Model-Driven Testing, TACoS - International Workshop on Test and Analysis of Component Based Systems, Warsaw, April 13th, 2003,in conjunction with ETAPS 2003

Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)
1530-1362/04 $ 20.00 IEEE