Automatic Test Case Generation for UML Activity Diagrams*

Chen Mingsong, Qiu Xiaokang, and Li Xuandong
State Key Laboratory of Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, Jiangsu, P.R.China 210093
{chenms,qiuxk}@seg.nju.edu.cn, lxd@nju.edu.cn

ABSTRACT

The test case generation from design specifications is an important work in testing phase. In this paper, we use UML activity diagrams as design specifications, and present an automatic test case generation approach. The approach first randomly generates abundant test cases for a JAVA program under testing. Then, by running the program with the generated test cases, we can get the corresponding program execution traces. Last, by comparing these traces with the given activity diagram according to the specific coverage criteria, we can get a reduced test case set which meets the test adequacy criteria. The approach can also be used to check the consistency between the program execution traces and the behavior of UML activity diagrams.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability

Keywords

random test case, instrumentation, UML activity diagram, test adequacy

1. INTRODUCTION

Testing is an important part of quality assurance in the software life-cycle. As the complexity and size of software systems grow, more and more time and manpower are required for testing. Manual testing is so labor-intensive and error-prone that it is necessary to develop automatic testing techniques in some circumstance. The testing process consists of three parts: test case generation, test execution, and test evaluation. Comparing with the other two parts, test case generation is more challenging and difficult.

The Unified Modeling Language(UML) is a standard visual modeling language that is designed to specify, visualize, construct and document the artifacts of software systems [9, 10]. UML provides a number of diagrams to describe particular aspects of software artifacts. These diagrams can be classified depending on whether they are intended to describe structural or behavior aspects of systems. UML activity diagrams describe the sequential or concurrent control flows of activities. They can be used to model the dynamic aspects of a group of objects, or the control flow of an operation. Also, UML activity diagrams can be used as the models to driven the test case generation.

In this paper, we use UML activity diagrams as design specifications, and consider the automatic approach to test case generation. Instead of deriving test cases from the UML activity diagram directly, we present an indirect approach which selects the test cases from the set of the randomly generated test case according to a given activity diagram. By using the method similar to [6], we first randomly generate abundant random test cases. Then, by running the program with these test cases, we can get the corresponding program execution traces. Last, by comparing these traces with the activity diagram according to the specific coverage criteria, we can prune some redundant test cases and get a reduced test case set which meets the test adequacy criteria. The approach can also be used to check the consistency between the program execution traces and the behavior of UML activity diagrams.

The paper is organized as follows. In next section, we introduce UML activity diagrams and related concepts. In section 3, we review the definition of some basic test adequacy criteria, and present three test adequacy criteria for UML activity diagrams. In section 4, the approach to automatic test case generation for UML activity diagrams is described in detail. The related works and some conclusions are given in the last section.

2. UML ACTIVITY DIAGRAMS

2.1 Notations

Different from the other diagrams in UML, an activity diagram extracts the core idea from the flow charts, the state transition graphs, and Petri nets[8, 9]. The recent major re-

*Supported by the National Natural Science Foundation of China (No.60425204, No.60233020), the National Grand Fundamental Research 973 Program of China (No.2002CB312001), and by the Jiangsu Province Research Foundation (No.BK2004080).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AST'06, May 23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

vision of the UML2.0 has introduced the significant changes and additions[5]. Compared with the UML1.x, the concrete syntax of activity diagrams remains mostly the same. But the abstract syntax and semantics have changed drastically. While in UML1.x, the activity diagrams have been defined as a kind of state machine diagrams. Now, there is no connection between the two, and the meaning of activity diagrams is being explained in terms of Petri net notions like the token, flow, edge weight etc. In this paper, according to UML2.0[5], we adopt the Petri net-like semantics of the activity diagrams.

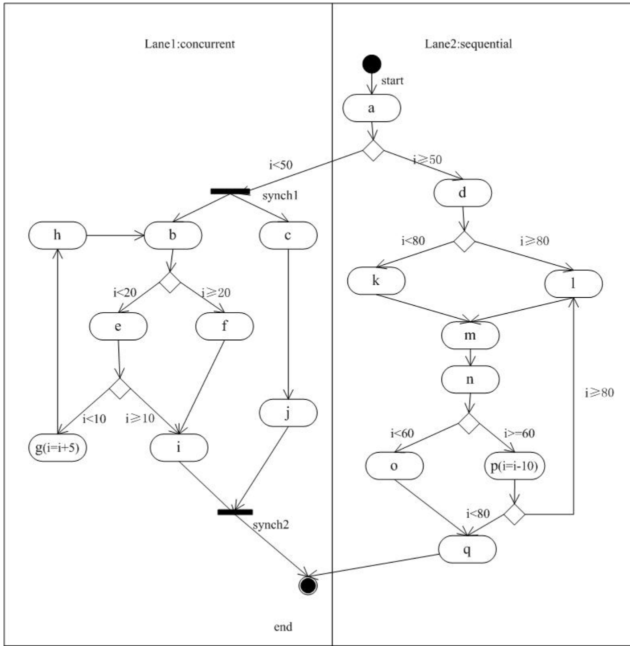

Figure 1: An Example of Activity Diagram

The syntax of the activity diagrams is similar to the original. The model elements consist of nodes, edges and swim lane. The nodes represent processes or process control, including action states, activity states, decisions, 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 outgoing arrows each labeled with a boolean expression to be satisfied to choose the branch. Joins or forks are shown by multiple arrows entering or leaving the synchronization bar. Swim lanes represent the supplier of activities. Figure 1 shows an activity diagram which consists of most elements to describe an operation.

For any activity diagrams considered in this paper, one activity is corresponding with the execution of one member function, and one swimming lane is corresponding with one class. Since we mainly focus on extracting essential information to derive test cases, we formalize an activity diagram as follows.

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

- $A = \{a_1, a_2, \ldots, a_m\}$ is a finite set of activity states;
- $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is a finite set of completion transitions;
- $C = \{c_1, c_2, \ldots, c_n\}$ is a finite set of guard conditions, and c_i is in correspondence with τ_i, Con is a mapping from τ_i to c_i so that $Con(\tau_i) = c_i$;
- $F \subseteq \{A \times C \times T\} \cup \{T \times C \times A\}$ is the flow relation between the activities and transitions;
- $a_I \in A$ is the *initial activity state*, and $a_F \in A$ is the *final activity state*, there is only one transition $\tau \in T$ and corresponding $c \in C$ such that $(a_I, c, \tau) \in F$, and $(\tau', c', a_I) \notin F$ and $(a_F, c', \tau') \notin F$ for any $\tau' \in T$ and corresponding $c' = Con(\tau')$. □

At any time, the current state(denoted by CS) of an activity diagram is represented by a set of activity states.

Definition 2. Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram. The *current state* CS of D is a subset of A. For any transition $\tau \in T$, let

- ${}^\bullet\tau$, $\tau^\bullet$ denote the preset and postset of τ respectively, then ${}^\bullet\tau = \{a \mid (a, c, \tau) \in F,\ Con(\tau) = c,\ a \in A\}$ and $\tau^\bullet = \{a \mid (\tau, c, a) \in F,\ Con(\tau) = c,\ a \in A\}$;
- *enabled(CS)* denotes the set of transitions start from CS, then $enabled(CS) = \{\tau \mid {}^\bullet\tau \subseteq CS$ are all completed and $Con(\tau)$ is satisfied $\}$, for any $\tau \in T$, if $\tau \notin enabled(CS)$, then $\tau \in disenabled(CS)$;
- *firable(CS)* denotes the set of only firable transitions from CS at certain moment, then *firable(CS)*=$\{\tau \mid \tau \in enabled(CS)$ and $(CS - {}^\bullet\tau) \cap \tau^\bullet = \emptyset\}$, and after some τ was fired, the new current state $CS' = fire(CS, \tau) = (CS - {}^\bullet\tau) \cup \tau^\bullet$;
- If $\mid CS \mid > 1$, then we call CS a *concurrent state* and the element in CS is a *concurrent activity*, otherwise CS is a *basic state* and the element in CS is a *basic activity*. □

For generating the test cases according to the approach presented in this paper, we need to compare the the program execution traces with the dynamic behaviors of an activity diagram. We call such a dynamic behavior a *run* of the activity diagram.

Definition 3. Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram. A run ρ of the activity diagram is a sequence of states and transitions, let

$$\rho = cs_0 \xrightarrow{(\tau_0, c_0)} cs_1 \xrightarrow{(\tau_1, c_1)} \ldots \xrightarrow{(\tau_{n-1}, c_{n-1})} cs_n$$

where $cs_0 = \{a_I\}$ is the initial state, $cs_n = \{a_F\}$ is the final state, if the current state is cs_i and $\tau_i \in firable(cs_i)$, $cs_{i+1} = fire(cs_i, \tau_i) = (cs_i - {}^\bullet\tau_i) \cup \tau_i^\bullet$, $0 \leq i \leq n-1$. □

Based on above definitions, we can parse an activity diagram and extract the essential information in a computer-recognizable format, so as to be processed automatically.

2.2 Simple Paths

From definition 3, we can definite the path of an activity diagram as follows.

Definition 4. Let $D = (A, T, F, C, a_I, a_F)$ be an activity diagram and let

$$\rho = cs_0 \xrightarrow{(\tau_0, c_0)} cs_1 \xrightarrow{(\tau_1, c_1)} \ldots \xrightarrow{(\tau_{n-1}, c_{n-1})} cs_n$$

be a run of D. Let the set $\Delta_i{=}cs_{i+1} - cs_i{=}\{a_1, a_2, \ldots, a_m\}$, $0 \le i \le n-1$ and $m > 0$. Let seq_i be an arbitrary permutation sequence of activities in Δ_i. seq_i is the activity execution segment of transition τ_i in the form $a_1 \to a_2 \to \ldots \to a_m$. A *path* of D is the sequential conjunction of all seq_i in ρ according to the consecutive order of i. The path is in the form $seq_1 \to seq_2 \to \ldots \to seq_{n-1}$. □

In definition 4, a path of an activity diagram is in essence a linear execution form of a run. From the definition, we can find that a run may have several corresponding paths because of the arbitrary permutation.

To calculate all the paths of an activity diagram, we adopt the depth-first search(DFS) algorithm. The existence of the loops will result in the multi-occurrence of some activities in a path. Because full combination of branches and loops will result in path explosion, in this paper, we just consider the *basic paths* defined as follows.

Definition 5. When calculating a path of an activity diagram, if each activity in the path occurs only once, we call such a path a *basic path* of the activity diagram. The diagram composed by all the basic paths of an activity diagram is called a *basic activity diagram* of the activity diagram. □

So far we solve the problem of infinite loops by introducing the basic paths. But the concurrency is still a most complicated issue. Because of the independency of the execution order of the threads, like the composition of the automata, the interleaving of the activities of distinct forked threads will cause state explosion. Usually it requires a huge amount of computer memory. Once such requirement excesses the feasible limitation, the algorithm fails to return a result. Facing with so many paths, we only have to select one representative by adopting the same strategy of partial order used in the model checking. This is the so-called "All from one, one for all"[7]. We call the representative of a set of paths a *simple path* of the activity diagram.

Definition 6. Let A be a set of activities of the basic activity diagram with respect to an activity diagram. A relation $R \subseteq A \times A$ which is reflexive, antisymmetric and transitive is a *partial order relation* on A. Let $\tau \in T$ be a completion transition in the basic activity diagram. Any $a_i \in {}^{\bullet}\tau$ has the relation with any $a_j \in \tau^{\bullet}$, denoted by $a_i \prec a_j$, which is the *partial order relation* of the activity diagram and called *happen-before.* □

Definition 7. In a basic activity diagram, if there exists many basic paths that have the same set of activities and the same partial order relation, we only select one representative from this path set. The selected representative is called a *simple path* of the activity diagram. □

3. TEST ADEQUACY CRITERIA FOR ACTIVITY DIAGRAMS

Objective measurement of the test quality is one of the key issues in software testing. As an essential part of any testing method, test adequacy criterion specifies the requirement of a particular software testing. The adequacy criteria of activity diagrams are based on the matching between the paths of activity diagrams and the program execution traces of the implementation codes. They mainly deal with the coverage of elements in activity diagrams.

The definition of test adequacy is given in [13, 14] as a measurement function. Let p be a program, and ts be a test case set. To generate test cases for an activity diagram, we consider the following three test adequacy criteria:

- *Activity Coverage* requires that all activity states in the activity diagram be covered. For any $t \in ts$, we can get the program execution trace *pet*. If there exist any function in *pet* whose corresponding activity is not *marked* in the activity diagram, we mark all the corresponding unmarked activities of *pet* and record the test case t. The value of *activity coverage* is the ratio of the marked activities to all activities in the activity diagram;

- *Transition Coverage* requires that all transitions in the activity diagram be covered. For any $t \in ts$, we can get the program execution trace *pet*. If for *pet* there exists corresponding transition that is not *marked* in the activity diagram, we mark all the corresponding unmarked transitions for *pet* and record the test case t. The value of *transition coverage* is the ratio of the marked transitions to all transitions in the activity diagram;

- *Simple Path Coverage* requires that all simple paths in the activity diagram be covered. For any $t \in ts$, we can get the program execution trace *pet*. If a simple path in correspondence with *pet* is not traversed in the activity diagram, we then record the test case t and put a *traversed* mark on the simple path. The value of *simple path coverage* is the ratio of the traversed simple paths to all simple paths in the activity diagram.

4. SELECTING TEST CASES FOR UML ACTIVITY DIAGRAMS

In this section, we give the details of the automatic approach to test case generation for UML activity diagrams. The approach first randomly generates abundant test cases for a JAVA program under testing. Then, by running the program with the generated test cases, we can get the corresponding program execution traces. Last, by comparing these traces with the given activity diagram according to the specific coverage criteria, we can get a reduced test case set which meets the test adequacy criteria.

4.1 Code Instrumentation

For gathering the program execution traces, we need to instrument the program under testing. Program instrumentation is a popular technique in dynamic testing. Its main idea is to insert some probes which are one or more test

```
Instrumentation (Clist, Mlist, Jlist)
//Clist: A list of Class names in an activity diagram;
//Mlist: A list of Method names in an activity diagram;
//Jlist: A list of JAVA code files.
begin
  for each JAVA code file jcf ∈ Jlist
  begin
    while(curToken = jcf.nextToken()! = NULL)
     if curToken=='class' then
        begin
          curToken = jcf.nextToken();
          if curToken is in the Clist then
             curClass = curToken;
          else curClass = NULL;
        end
     if curClass! = NULL & curToken ∈ Mlist then
       begin
         Insert log.write(curClass, curToken,"begin")
            as the first statement of the method;
         tempToken=curToken;
         Locate the curToken to the end of the method;
         Insert log.write(curClass, tempToken,"end")
            as the last statement of the method;
       end
    endwhile
  end
end
```

Figure 2: Instrumentation Algorithm

statements inserted in the original source code for recording dynamic information. When executing the instrumented program, the probes are executed and the execution behavior data is thrown out and can be recorded.

In this paper, we apply the instrumentation technique to trace the real execution orders of the member functions of a program. The algorithm is given in Figure 2, which instruments a JAVA program to extract the class and function information according to the given activity diagram.

4.2 Selecting Test Cases by Activity and Transition Coverage

Comparing with the simple path coverage, the activity coverage and transition coverage are more simple. The main reason is that the matching based selection can be executed on the activity diagram directly, and can record the history information on the activity diagram. According to the definition of coverage, when matching a program execution trace with the activity diagram, we first need to check whether the program execution trace is consistent with the activity diagram. If the program execution trace can not match a path of the activity diagram, the corresponding test case is irrelevant. Then we need to check whether the program execution trace contains some corresponding unmarked activities or transitions. A test case, which results in a program execution trace with some unmarked activities or transitions, is picked out as a candidate test case in the result. At the end of the matching, the unmarked activities or transitions of the matched program execution trace need to be marked for the next selection. The selection process terminates when the corresponding test adequacy criteria is satisfied.

4.3 Selecting Test Cases by Simple Path Coverage

The complexity of selecting test cases by using path coverage criterion is related to the definition of the paths of an activity diagram and to how many defined paths in an activity diagram. The synchronization, concurrency and loops are the main reasons resulting in the complexity. During selecting, the program execution traces must satisfy the semantics of the synchronization such as the join and fork in the activity diagram. Concurrency makes the execution of the forked threads executing independently. So the number of the combination of the activity grows drastically along with the growth of the number of the threads. Also the loops in an activity diagram may result in a path with infinite activities. In this paper, we only consider the simple paths of an activity diagram.

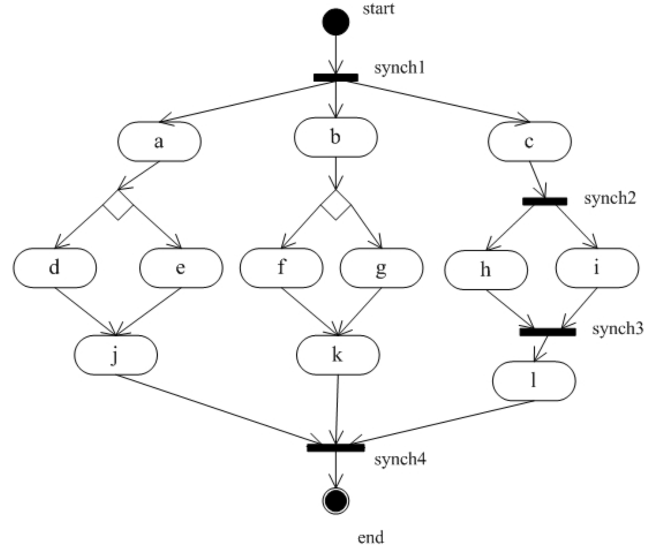

Figure 3: An Example of Basic Activity Diagram

As an example of basic activity diagram in Figure 3, its partial order relation is represented as follows:

$$
\begin{gathered}
< start > \prec < a, b, c >, \\
< a > \prec < d >, \\
< a > \prec < e >, \\
< d > \prec < j >, \\
< e > \prec < j >, \\
< b > \prec < f >, \\
< b > \prec < g >, \\
< f > \prec < k >, \\
< g > \prec < k >, \\
< c > \prec < h, i >, \\
< h, i > \prec < l >, \\
< j, k, l > \prec < end >.
\end{gathered}
$$

The symbol $< a >$ denotes a set of activities which has an element a. Taking the concurrency into account, there are 33600 basic paths in this figure, such as :
$start \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow j \rightarrow f \rightarrow k \rightarrow h \rightarrow i \rightarrow l \rightarrow end$,
$start \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow j \rightarrow f \rightarrow k \rightarrow i \rightarrow h \rightarrow l \rightarrow end$,
$start \rightarrow c \rightarrow b \rightarrow a \rightarrow d \rightarrow j \rightarrow f \rightarrow k \rightarrow h \rightarrow i \rightarrow l \rightarrow end$,
$start \rightarrow a \rightarrow b \rightarrow c \rightarrow e \rightarrow j \rightarrow g \rightarrow k \rightarrow h \rightarrow i \rightarrow l \rightarrow end$,
etc.

We have defined the simple path to avoid the potential loops and concurrency. But we have not presented a practical algorithm to search all the simple paths of an activity diagram. It seems easy that if the activity diagram does not contain any synchronization, we can use the traditional DFS algorithm directly. Otherwise the common DFS algorithm will cause the loss of paths, so we should redefine the path coverage criterion and give an improved DFS algorithm.

Definition 8. Let AG be an activity diagram, BAG be the basic activity diagram of AG, $ps_1 = \{p \mid p$ is a simple path of BAG and p contains no concurrent activity$\}$ and $ps_2 = \{p \mid p$ is a simple path of BAG and p contains at least one concurrent activity$\}$. If for any $ps_3 \subseteq ps_2$ and all the paths of ps_3 can cover all the concurrent activities of BAG at least once, then the path set $ps_1 \cup ps_3$ can cover all the activity diagram. We call such a path set the *simple path coverage set* of the activity diagram. □

For example, we only need two simple paths as follows:
$start \to a \to b \to c \to d \to f \to j \to k \to i \to h \to l \to end$,
$start \to a \to b \to c \to e \to g \to j \to k \to i \to h \to l \to end$
to cover the basic activity diagram in Figure 3. But here it should be stated that we do not restrict the minimum cardinality of the simple path coverage set to satisfy the simple path coverage criterion of the activity diagram.

Figure 4 gives a detailed algorithm to calculate a set of simple paths to cover the basic activity diagram of an activity diagram, and Figure 5 gives an supplementary description of a function which is used in Figure 4. $DFSStack$ is a stack which is used to store the states explored by the modified DFS algorithm. $StackOfFiredTransSet$ is also a stack to record the set of fired transitions between two consecutive states in the $DFSStack$. We use the $CurTrace$ to denote the sequence of activities visited according to the states in the $DFSStack$. $AllPath$ saves all the distinct explored paths which start from a_I to a_F. The stack provides several operations to deal with the top element of the stack. $Statck.Pop()$ returns the top element of the stack and removes the top element while $Statck.GetTop()$ only returns the top element. It is worthy to notice the set $enabled(A)$ and the set $firable(A)$. The $enabled(A)$ and $firable(A)$ respectively are a union of all the $enabled$ or $firable$ transitions of elements in A. The set $enabled(A)$ is a constant set, but the $firable(A)$ is not. The $firable(A)$ varies according to the history information of the searching. At the beginning of the searching, for an activity act, the set $enabled(act)$ is the same as $firable(act)$. The complexity of modified DFS algorithm is no more than the common DFS, because the modification of our algorithm does not increase the complexity of common DFS.

4.4 Matching Simple Paths with Program Execution Traces

After searching the simple path coverage set from an activity diagram, we should select the test cases by comparing the simple paths with program execution traces. A simple path is in essence a non-loop trace while the program execution trace can have many loops. So a program execution traces may be much longer than the corresponding simple paths, and the program execution trace may contain some activity nodes which do not exist in the corresponding simple path. Another problem which is worthy of noticing is how to find the representative only once for a simple path.

```
DFSStack = ∅, StackOfFiredTransSet = ∅,
CurTrace = ∅, AllPath = ∅;
AGSimplePathGenerator(D)
cs = {a_I}, precs = { }, cs_firedTransSet = { };
begin
   DFSStack.Push(cs);
  StackOfFiredTransSet.Push(cs_firedTransSet);
  repeat
    cs = DFSStack.Pop(); precs = DFSStack.GetTop();
    StackOfFiredTransSet.Pop();
    precs_firedTransSet = StackOfFiredTransSet.Pop();
    if ∃τ ∈ enabled(precs) & | τ• | > 1 & τ• ⊆ cs
      then ModifyTrans(τ);
    if firable(cs) is not empty then
      begin
        NxState(cs, cs_firedTransSet, newCS, newTransSet);
        continue;
      end
    if a_F ∈ cs then
      begin
        record the curTrace from bottom to the top in
          a sequence seq;
        AllPath.Push_back(seq);
      end
    if DFSStack.size() = 0 then return;
    for each transition τ' ∈ precs_firedTransSet
      begin
        for each activity s ∈ τ'•
          begin
            CurTrace.RemoveFromTheEnd();
            for each transition τ_i ∈ enabled(s)
              firable(s) = firable(s) ∪ {τ_i};
          end
      end
  until DFSStack = ∅;
end
```

```
NxState(cs, cs_firedTransSet, ncs, ncs_firedTransSet)
begin
  for each activity act ∈ cs
    if firable(act) is not empty & τ ∈ firable(act) then
      if nact ∈ ncs = (cs − •τ) ∪ τ• & nact ∉ CurTrace then
        begin
          cs_firedTransSet.add(τ);
          firable(act) = firable(act) − {τ};
          for each act' ∈ τ•
            CurTrace.AddToTheEnd(act');
        end
      else
        begin
          τ.disenable(); break;
        end
  DFSStack.Push(cs);
  StackOfFiredTransSet.Push(cs_firedTransSet);
  if cs and ncs do not have same activities then
    begin
      DFSStack.Push(ncs);
      StackOfFiredTransSet.Push(ncs_firedTransSet);
    end
end
```

Figure 4: The Algorithm for Searching Simple Path Coverage Set

```
ModifyTrans (synch)
begin
  int max = 0;
  for each activity act ∈ synch•
    if max < |enabled(act)| then max = |enabled(act)|;
  if ∃act, that |enabled(act)| = max & |firable(act)|
     < max  &  ∃act', that |enabled(act')| < max &
    |firable(act')| = 0 then
      for each activity nact that nact ∈ synch•  &
        |enabled(nact)| < max
        for each transition tran ∈ enabled(nact)
          firable(nact) = firable(nact) ∪ {tran};
end
```

Figure 5: The Algorithm of Function $ModifyTrans$

Here we assume that the profile name of an activity node in an activity diagram is the same as one function name in corresponding program.

Definition 9. Let *pet* be a sequence of function names in a program execution trace, in the form $f_1\hat{}f_2\hat{}\ldots\hat{}f_{n-1}\hat{}f_n$. Let *sp* be a sequence of activity node profile names of the corresponding simple path, in the form $a_1\hat{}a_2\hat{}\ldots\hat{}a_{m-1}\hat{}a_m$. *pet* can match *sp* if an only if

1. $n \geq m$ and f_1=a_1 and f_n=a_m;
2. The order of the functions in *pet* should be in accordance with a path of the activity diagram;
3. Let $pet_set = \{f_i \mid f_i$ is a function name in *pet* and $1 \leq i \leq n\ \}$ and $sp_set = \{a_i \mid a_i$ is an activity profile name in *sp* and $1 \leq i \leq m\ \}$, then $sp_set \subseteq pet_set$. □

When a simple path is matched, we should delete this path from the simple path coverage set. So the matching process is terminated when the simple path coverage set is empty. In this case we can get the test case set which cover all the simple paths of the activity diagram according to the simple path coverage criterion, otherwise the path coverage and some irrelevant test case information will be reported.

4.5 Tool Prototype

The approach presented in this paper have been implemented into a tool prototype *AGTCG*. Its graphical interface is shown in Figure 6, and allows the users to construct, edit, and analyze Activity Diagram interactively. The tool can instrument a JAVA program according to the given activity diagrams, and use the randomly generated test cases to run the instrumented JAVA program, and gather the corresponding program execution traces. By comparing the program execution traces with the activity diagram, the tool gives the test case sets which satisfy the special test adequacy criteria.

To demonstrate the usability of the tool, we implemented a JAVA program according to the activity diagram depicted in using Figure 1. The swimlane is mapped to the class and the activities are mapped to the member functions. For example, the activity a in Figure 1 is mapped to the member function $Sequential.a()$ and the activity e is mapped to member function $Concurrent.e()$. The program has only an input i whose type is integer. *AGTCG* automatically instruments the program according to the mapping from the activities of the activity diagram to the functions of the program. For simplicity, the random test case generation algorithm generates a coarse test case set which contains all the integers rang form 0 to 100. By running the tool, we get the results in Table 1. From the result, we can only get the reduced valid test cases 50, 60, 80 and the corresponding coverage. Also we get the test cases from 0 to 49 which are the irrelevant test cases that generate the inconsistent program execution traces for the activity diagram in Figure 1.

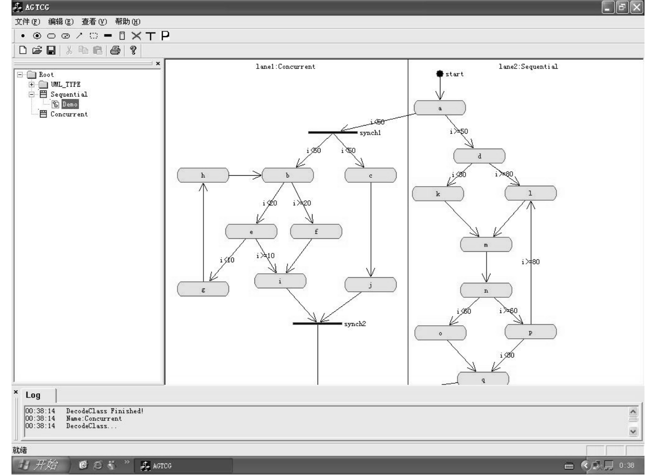

Figure 6: The Interface of $AGTCG$

5. RELATED WORK AND CONCLUSION

Agitator[1], a commercial tool, has a similar approach to AGTCG. It creates very simple test cases(almost random), and then refines them to satisfy test criteria. But the key difference between Agitator and our tool is that Agitator tests are designed for JAVA methods(Unit testing) and the criteria are based on the implementation rather than a model. At present, to our knowledge there are several distinct methods which generate test cases for different testing levels from UML activity diagrams. In [12], a UML activity diagram is formalized and transformed to a test case model, and the test cases could be generated from the case model. This approach need to take the transformation cost into consideration. A strategy was adopted to derive test scenarios from activity diagrams and to generate test cases from the test scenario in [4]. But it only gives a conceptual frame. The systematic method was not proposed. [11] gives an approach to generate test case from UML activity diagrams base on Gray-Box method. It demonstrates a systematic method to generate test cases directly from UML activity diagrams, and many parts of this method could be automated. Unlike above methods, our method has one distinct feature that the execution is totally automated without manual work. Thus it can save the testing cost and reduce the probability of mistakes.

The approach presented in this paper supports the essentially automatic test case generation according to some test adequacy criteria for UML activity diagrams. At the

Table 1: **Results Given by** $AGTCG$

Input test case set	the set $\{i \mid i \in N\ and\ 0 \leq i \leq 100\}$		
Activity and Transition coverage criteria	test coverage	selected test cases	irrelevant test cases
	52.38% & 48.15%	$50, 60, 80$	0–49
Simple Path coverage criterion	test coverage	selected test cases	irrelevant test cases
	50.00%	$50, 60, 80$	0–49
Simple Path coverage set	$start \rightarrow a \rightarrow b \rightarrow c \rightarrow e \rightarrow j \rightarrow i \rightarrow end$		
	$start \rightarrow a \rightarrow b \rightarrow c \rightarrow f \rightarrow j \rightarrow i \rightarrow end$		
	$start \rightarrow a \rightarrow d \rightarrow k \rightarrow m \rightarrow n \rightarrow o \rightarrow q \rightarrow end$		
	$start \rightarrow a \rightarrow d \rightarrow k \rightarrow m \rightarrow n \rightarrow p \rightarrow q \rightarrow end$		
	$start \rightarrow a \rightarrow d \rightarrow l \rightarrow m \rightarrow n \rightarrow o \rightarrow q \rightarrow end$		
	$start \rightarrow a \rightarrow d \rightarrow l \rightarrow m \rightarrow n \rightarrow p \rightarrow q \rightarrow end$		

same time, it can also check the consistency between specifications and corresponding programs. It coincides with Model Driven Architecture(MDA)[3] and the model driven testing[2]. The case studies are our next work.

6. REFERENCES

[1] Agitar. *Agitator and Agitar Management Dashboard 3.0*. Available at http://www.agitar.com.

[2] R. Heckel and M. Lohmann. Towards Model-Driven Testing. In *TACoS - International Workshop on Test and Analysis of Component Based Systems,in conjunction with ETAPS2003*, pages 284–291, April 2003.

[3] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture–practice and promise*. Addison-Wesley, 2003.

[4] M. Liu, M. Jin, and C. Liu. Design of Testing Scenario Generation Based on UML Activity Diagram. *The Engineering and Application of Computer(in Chinese)*, 12:122–124, 2001.

[5] OMG. *UML2.0 Superstructure Specification*. Available at http://www.omg.org/#UML2.0, october 2004.

[6] C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *QoSA/SOQUA*, pages 242–256, 2005.

[7] D. A. Peled. All from One, One for All: On Model Checking Using Representatives. In *Proc. of the* 5_{th} *International Conference on Computer Aided Verification* (CAV'1993), pages 24–31. LNCS 697, Springer, 1993.

[8] J. Peterson. *Petri Nets Theory and the Modeling of Systems*. Prentice-Hall, N.J., 1981.

[9] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2001.

[10] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 2001.

[11] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 284–291, 2004.

[12] M. Zhang, C. Liu, and C. Shun. Automated Test Case Generation Based on UML Activity Diagram Model. *Journal of Beijing University of Aeronautics and Astronautics(in Chinese)*, 27:433–437, August 2001.

[13] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[14] H. Zhu and X. He. A Methodology of Testing High-level Petri Nets. *Information and Software Technology*, 44(8):473–489, 2002.