Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams

Dong Xu
Edith Cowan University

Huaizhong Li
Edith Cowan University

Chiou Peng Lam
Edith Cowan University

10.1109/APSEC.2005.110

Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams

Dong Xu [1]
School of Computer Science,
Shanghai University,
China
d.xu@ecu.edu.au

Huaizhong Li
School of Computer and
Information Science,
Edith Cowan University,
Australia
h.li@ecu.edu.au

Chiou Peng Lam
School of Computer and
Information Science,
Edith Cowan University,
Australia
c.lam@ecu.edu.au

Abstract

Test case generation is one of the most important issues in software testing research and industrial practice. Test scenarios are frequently used to derive test cases for scenario-based software testing. However, the generation of the test scenarios is usually a manual and labor-intensive task. It is desired that test scenarios can be automatically generated. In this paper, we propose an automated approach using adaptive agents to directly generate test scenarios from the UML activity diagrams.

1. Introduction

Software testing remains the primary technique used to maintain quality of the software products and to gain consumers' confidence in software. Unfortunately, it is well-known that testing software is a time-consuming and costly process [4]. Therefore, techniques that support the automation of software testing will result in significant cost and time savings for the software industry. Automatic generation of the test data is essential for the automation of software testing.

Great attention has been given to effectively using UML, which is the industrial de-facto standard for modeling object-oriented software systems, in software testing (see, for example, [6] and the references therein). One of the focused research topics is using UML artifacts for scenario based testing.

Scenarios represent the sequences of executions in a software system. Properly generated test scenarios are essential for the scenario-based software testing to achieve the required test adequacy and to guarantee the software quality [8]. However, one major problem with the generation of test scenarios is that the generation procedures are either completely manual or can not be fully automated [2].

Currently there is a research trend to apply artificial intelligence techniques in software engineering, especially in software testing research (see for example, [5][11][15] and the references therein). One of the much focused areas in applying artificial intelligent techniques to software testing is the automated generation of test data.

Results have been reported of using artificial techniques in test data generation for software testing. The focus of the techniques mainly involves the applications of the genetic algorithms and the Ant Colony Optimization algorithms, for examples, [12][15]. However, automation of the generation procedure, efficiency of the generation algorithms, and the feasibility of the generated test data are frequently concerned. Furthermore, the reported results mainly concern with the state-based testing problems. The problem of generating test scenarios for scenario-based software testing receives less attention.

Recently, an approach has been proposed in [13] to use so-called anti-ant-like agents to automatically generate test threads from the UML activity diagrams. It is shown in [13] that using agents provides a potential avenue to automate the generation of test scenarios for scenario-based software testing. However, the proposed algorithm in [13] may result in redundant exploration of the activity diagrams and hence reduce the efficiency of the generation process. Furthermore, the complexity of the activity programs which are explored in [13] calls for further research.

In this paper, we propose to use adaptive agents to

[1] This author was a visiting researcher at Edith Cowan University when the paper was written.

overcome the difficulties and limitations encountered in [13]. The algorithm proposed in this paper is more efficient, and is capable to deal with the UML activity diagrams which contain more complicated structures.

This paper is structured as follows. Section 2 provides an analysis of the UML activity diagrams. Section 3 presents an adaptive agent approach to test scenario generation. Section 4 briefly describes the tool support for the proposed approach. Section 5 briefly discusses the related work; and the conclusion is found in Section 6.

2. Analysis of the UML Activity Diagrams

In general, an UML activity diagram includes an initial state, the action states, the final states, the fork nodes, the join nodes, the branch nodes, the merge nodes, the transitions and their associated guard conditions, and the final nodes.

The following definition for the UML activity diagrams is modified from [16]:

Definition 1 *An activity diagram is a 8-tuple AD = (A, B, M, F, J, K, T, a_0) where A = $\{a_1, a_2, \cdots, a_n\}$ is a finite set of action states of the UML activity diagram; B = $\{b_1, b_2, \cdots, b_u\}$ a finite set of branches; M = $\{m_1, m_2, \cdots, m_v\}$ a finite set of merges; F = $\{f_1, f_2, \cdots, f_y\}$ a finite set of forks; J = $\{j_1, j_2, \cdots, j_x\}$ a finite set of joins; K = $\{k_1, k_2, \cdots, k_w\}$ a finite set of final states and end flows; T = $\{t_1, t_2, \cdots, t_z\}$ a finite set of transitions which satisfies $\forall t \in T, t = < c > e \vee t = e$ where $c \in C, e \in E, C = \{c_1, c_2, \cdots, c_l\}$ is a finite set of guard conditions, $E = \{e_1, e_2, \cdots, e_s\}$ is a finite set of edges of the activity diagram; and a_0 is the unique initial state.*

A test scenario is defined as follows:

Definition 2 *Let AD = (A, B, M, F, J, K, T, a_0) be an activity diagram. Denote TS the set of test scenarios for AD. $\forall ts \in TS, ts$ is a sequence of action states and transitions, i.e.*

$$ts = a_0 t_0 a_1 t_1 \cdots a_n t_n k \wedge a_i \in A \wedge t_i \in T \wedge k \in K, i = 1,2,\cdots,n.$$

A directed graph is defined as G = (V, E) where V is a set of vertices of the graph and E a set of edges of the graph. A UML activity diagram can be viewed as an activity graph where the vertices are the activity nodes, the object nodes, the branch nodes, the fork nodes, the join nodes, and the initial node, while the edges are the activity edges in the activity diagram. An activity graph is a directed, dynamic graph in which the activity edges may become accessible only after the evaluation of their associated guard conditions.

The fork-join pairs in the UML activity diagrams represent concurrent executions. Due to the concurrency contained in fork-join pairs, it is difficult to derive test scenarios for an UML activity diagram which contains fork-join pairs. Furthermore, in complicated UML activity diagrams, the fork-join pairs can have complicated structures, namely, they may contain nested fork-join pairs, and they may include branches or loops between the fork and the join.

To our knowledge, there is no systematic approach reported in literature to automatically generate all test scenarios for the UML activity diagrams which contain complicated fork-join structures. For example, the approach in [16] generates only incomplete test scenarios for an UML activity diagram under the assumption that the activity diagrams contain forks which only have two outgoing edges. Although the approach reported in [13] does not constrain the outgoing edges of the forks, the generated test scenarios for a fork-join pair only contain the combination of the sequentially connected execution paths between the fork and the join, therefore, may miss some test scenarios for the fork-join pair.

This paper further extends the approach reported in [13] to tackle more complicated UML activity diagrams. In particular, the approach proposed in this paper aims at providing a solution to generate complete test scenarios for the UML activity diagrams which may contain complex fork-join structures.

First of all, we present a simple classification of the fork-join pairs considered in this paper.

2.1 A simple classification of the fork-join pairs

The fork-join pairs considered in this paper can be classified as four types, namely, the Atomic-Fork-Join (AFJ) type, the Simple-Fork-Join (SFJ) type, the Simple-Nested-Fork-Join (SNFJ) type and the Branch-Nested-Fork-Join (BNFJ) type. Note that it is possible that the fork-join pairs in the UML activity diagrams may contain other structures. Research is currently being carried out to investigate the fork-join structures which are not covered here.

Before we provide the definitions for each of the types, we give the following definition:

Definition 3 *Let ID denote the incoming degree of a node in the activity diagram, OD denote the outgoing degree of a node. ID(x) and OD(x) denote the number of incomings and outgoings of the node x, where $x \in A \vee x \in B \vee x \in M \vee x \in F \vee x \in J \vee x \in K$.*

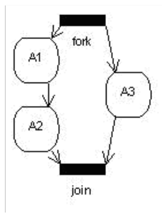


Figure 1

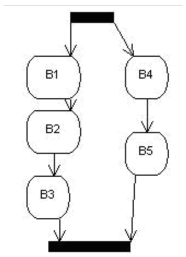


Figure 2

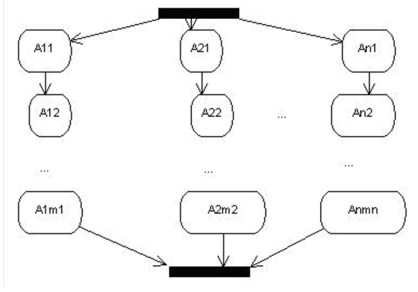
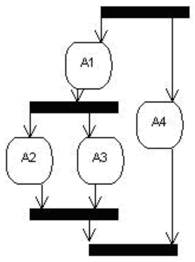
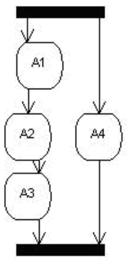


Figure 3





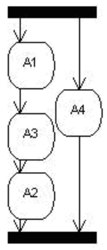
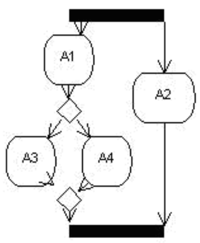
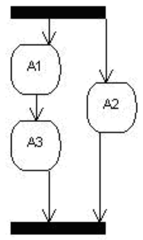


Figure 4 Figure 4.1 Figure 4.2





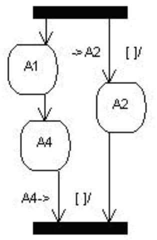


Figure 5 Figure 5.1 Figure 5.2

Especially, ID(a_0*) = 0, OD(*a_0*) = 1, OD(k) = 0,* $\forall k \in K$.

Now, we are ready to give a definition for the Atomic-Fork-Join type:

Definition 4 *An AFJ is a 4-tuple AFJ = (A, T, f, j) where A and T are the set of action states and the set of transitions, respectively; f is the unique fork in the AFJ; j is the unique join in the AFJ;* $\mathrm{OD}(a) = 1 \wedge \mathrm{ID}(a) = 1, a \in A$ *which means that each action state has one incoming and one outgoing exactly; and* $\mathrm{OD}(f) = 2 \wedge \mathrm{ID}(j) = 2$ *which means that the unique fork only has 2 outgoings, and the corresponding join has 2 incomings. The sequence* $f \xrightarrow{t_{i1}} a_{i1} \xrightarrow{t_{i2}} a_{i2} \cdots \xrightarrow{t_{in}} j$ *is called an execution path of this fork-join.*

Note that only two parallel execution paths are contained in an AFJ. We use *ES* to denote the set of the execution paths $ES = \{es_1, es_2\}$, then $|ES| = 2$. If $S_1 = \{a \mid a \in A,\ a$ occurs in $es_1, es_1 \in ES\}$, $S_2 = \{a \mid a \in A,\ a$ occurs in $es_2, es_2 \in ES\}$, then $S_1 \cap S_2 = \phi$. Moreover, if $U_1 = \{t \mid t \in T, t$ occurs in $es_1, es_1 \in ES\}$, $U_2 = \{t \mid t \in T,\ t$ occurs in $es_2, es_2 \in ES\}$, then $U_1 \cap U_2 = \phi$. Therefore, each action state and each transition in an AFJ only appear in one of the two execution paths.

For example, the fork-join pairs shown in Figure 1 and Figure 2 are both AFJs. Denote the number of test scenarios as NTS. For the fork-join pair in Figure 1, NTS = C_3^1= 3. For the fork-join pair shown in Figure 2, NTS = $C_4^1 + C_4^2$ = 4 + 4×3/2 = 10.

Next, we provide a definition for SFJ:

Definition 5 *A SFJ is a 4-tuple SFJ = (A, T, f, j) where A, T, f and j are the set of action states, the set of transitions, the unique fork, and the unique join in the SFJ, respectively;* $\mathrm{OD}(a) = 1 \wedge \mathrm{ID}(a) = 1, a \in A$ *which means that each action state has one incoming and one outgoing exactly;* $\mathrm{OD}(f) = n \wedge \mathrm{ID}(j) = n$ *where* $n \geq 2 \wedge n \in N$*, N is the set of natural numbers; the sequence* $f \xrightarrow{t_{i1}} a_{i1} \xrightarrow{t_{i2}} a_{i2} \cdots \xrightarrow{t_{in}} j$ *is called an execution path of this fork-join.*

Obviously n parallel execution paths are contained in a SFJ. Hence, $ES = \{es_1, es_2, \cdots, es_n\}$ and $|ES| = n$. If $S_1 = \{a \mid a \in A,\ a$ occurs in $es_1, es_1 \in ES\}$, $S_2 = \{a \mid a \in A,\ a$ occurs in $es_2, es_2 \in ES\}$, $\cdots$, $S_n = \{a \mid a \in A,\ a$ occurs in $es_n, es_n \in ES\}$, then $S_1 \cap S_2 \cap \cdots \cap S_n = \phi$. Moreover, if $U_1 = \{t \mid t \in T,\ t$ occurs in $es_1, es_1 \in ES\}$, $U_2 = \{t \mid t \in T,\ t$ occurs in $es_2, es_2 \in ES\}$, $\cdots$, $U_n = \{t \mid t \in T,\ t$ occurs in $es_n, es_n \in ES\}$, then

$U_1 \cap U_2 \cap \cdots \cap U_n = \phi$. Therefore, each action state and each transition in an SFJ only appear in one of the n execution paths.

The fork-join pair shown in Figure 3 is a SFJ. Its NTS is:

$$\text{NTS} = \prod_{i=1}^{n-1} \sum_{j=0}^{m_{i+1}-1} \left(C^{j+1}_{\sum_{k=1}^{i} m_k + 1} \times C^{j}_{m_{i+1}-1}\right)$$, where we assume that $m_1 \geq m_2 \geq \cdots \geq m_n \geq 1, n \geq 2 \wedge C_0^0 = 1$.

Obviously, the SFJ is an extension of the AFJ. In general, most of the fork-join pairs in the UML activity diagrams belong to the SFJ type.

The NTS formula shown above can be deduced easily, and hence the derivation procedure is omitted. The NTS formula can be used to verify whether a proposed method is capable to generate all test scenarios for a SFJ in an activity diagram.

Now, we define the SNFJ type as follows:

Definition 6 *A SNFJ is a fork-join pair which contains a nested SFJ inside. If the nested SFJ is replaced with one of its test scenarios, and as a consequence the host fork-join pair becomes a SFJ, we call such fork-join a SNFJ.*

For example, the fork-join pair shown in Figure 4 is a SNFJ. It can be decomposed into 2 SFJs shown in Figure 4.1 and Figure 4.2 respectively. Obviously, the NTS of a SNFJ = $\sum$ the NTS of the decomposed SFJs. Therefore, the NTS of a SNFJ can be calculated using the above formula for a SFJ.

Finally, we give the definition for the BNFJ type:

Definition 7 *A BNFJ is a fork-join which contains a branch-merge pair inside. If the branch-merge is replaced with one of its test scenarios, and as a consequence the host fork-join pair becomes a SFJ, we call such fork-join a BNFJ.*

For example, the fork-join pair shown in Figure 5 is a BNFJ. A BNFJ can be decomposed into two SFJs shown in Figure 5.1 and Figure 5.2 respectively. Obviously, NTS of a BNFJ = $\sum$ NTS of the two decomposed SFJs.

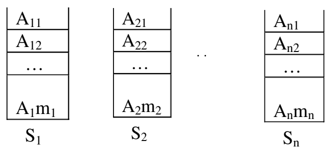
Since the SFJ type forms the basis of the decomposition for the SNFJ type and the BNFJ type, we clearly need to investigate how to generate test scenarios for the SFJ type. In the following, we present an algorithm, called the SFJ algorithm, to generate the test scenarios for a SFJ type.

2.2 The SFJ algorithm

Suppose the SFJ is the one shown in Figure 3. The pseudo codes of the SFJ algorithm are given below.

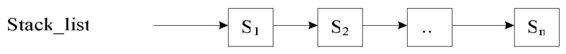
The SFJ Algorithm:

Step 1: Create n action stacks as the following:



where A_{ij}, i=1,2,…n, j=1,2,…m are the action states that appear in the SFJ;

Step 2: Create a stack list, denoted as "st":



Step 3: Create a tree node structure:

```
Struct tree_node:
       Action: action;
       Stack_list: st;
Endstruct;
Create a scenario tree as an
intermediate data structure;
```

Step 4: Create a recursive function to generate the scenario tree for the SFJ. Suppose the root of the execution path tree starts from the fork node, denote here root.action = fork, root.st = st. The function pseudo codes to generate this tree are given below:

```
//create scenario tree for simple fork-
//join and it's a recursive function
createSFJScenarioTree(tree_node:treeNode)
{
       for each stack(S_i) in
       stack_list(st) of the treeNode
         //pop up top element
         action := S_i.popup();
         if S_i. isEmpty() then
              remove the S_i from the
              stack_list(st);
         endif
         create a tree node named
         nodeNew where nodeNew.action :=
         action, node.st = st;
         //adding a child to the current
         //tree node
         tree.addChild(node, nodeNew);
       endfor;

       // operation for each leaf of the
       // current thread tree
       for each tree leaf
         //Is the stack list of the leaf
         //node EMPTY?
         if leaf.st is NOT empty then
             createSFJScenarioTree(leaf);
             //calling recursively
         endif
       endfor
   }
```

Step 5: Generate test scenarios for the SFJ by traversing the scenario tree. Each path from root to a leaf forms a test scenario for the SFJ.

Note that the NTS formula can be used to verify that all test scenarios for a SFJ have been generated.

By using the SFJ algorithm, 10 test scenarios can be generated for the fork-join pair shown in Figure 2, and the generated test scenarios are exactly the same as those listed in Table 2. However, only 6 test scenarios can be generated using the algorithm proposed in [16].

3. Generate Test Scenarios from the UML Activity Diagrams

In this section, we will describe the complete algorithm of using adaptive agents to automatically generate test scenarios from the UML activity diagrams.

3.1 Adaptive Agents and Their Behavior

We consider the problem of sending adaptive agents to search an activity graph which corresponds to a UML activity diagram. The objective of the agent's exploration is to build the test scenarios from the corresponding activity diagram.

The adaptive agents proposed exhibit similar behavior of some bacteria, for example the type of bacteria called Maxococcus xanthus. Maxococcus xanthus is a type of myxobacterium which lives in soil. A Maxococcus xanthus can move around the ground to search for nutrients. As it moves, the bacterium can lay down a slime trail which may play the role of greasing the path to easy the movement. Therefore, if a bacterium finds a slime trail, it has a better chance to follow that trail to make its movement easier [14].

Like Maxococcus xanthus bacteria, our agents move around in the directed activity graph to search for nutrients. As these agents move, they also lay down slime trails to mark their paths. However, unlike Maxococcus xanthus, bacteria, our agents actually exhibit opposite behavior with regard to the use of slime trails, namely, too strong slime concentration will turn our agents into stationary spores. A spore can not move around further to search for nutrients. Therefore, once an agent is turned into a spore, it will be removed from the search process.

Like many bacteria, our agents can adapt to the environment. Namely, if nutrients are sufficient at a place such that the workload of an arriving agent is too high, the agent will simply clone itself to easy the workload it has to take. The cloned agents will then be dispersed to different directions to cover the nutrients.

When our adaptive agents are dispatched to search an activity graph, they observe the following rules:

Rule 1: An agent can only move in the directions of the edges at a fixed speed.
Rule 2: An agent can lay down fixed amount of slime over an activity edge when it arrives at the edge.
Rule 3: An agent is turned into a spore if it arrives at an activity edge which possesses too high slime concentration.
Rule 4: An agent is killed if it finds antibiotic substance.
Rule 5: Search an activity edge represents one normal work load for an agent at a particular time. An agent can only take one workload at a time.
Rule 6: An agent can clone itself if it is requested to carry out a higher workload. The cloned agents inherit the complete information of the fathering agent and exhibit the same behavior of the fathering agent.

When using the proposed agents to search an activity diagram, special attentions need to be paid to two types of nodes in an activity diagram:

- *The final nodes are considered as spots which contain antibiotic substance.* An agent is killed if it arrives at a final node.
- *The branch nodes are considered as the spots which contain rich nutrients.* If a branch node has *m* outgoings, then the requested workload for an agent which arrives at the branch node is *m* times higher than its normal workload. Therefore, the agent has to reproduce *m-1* clones at the branch node; each agent will carry on a search in a different outgoing.

The overall behavior of the adaptive agents in our approach is governed by an activity diagram illustrated in Figure 6. Note that for simplicity, it is assumed in this paper that each cyclic loop is executed at most two times. This assumption is the same as the one in [16].

3.2 Algorithm

We define a scenario tree as an interim storing structure. The initial node of an activity diagram is regarded as the root of the tree. Each tree node denotes the elements of the activity diagram such as the action states, the forks, the joins, the branches, the merges, the guard conditions, and the final states. Test scenarios can then be derived from the scenario tree by traversing the tree.

First of all, we represent the guard conditions in

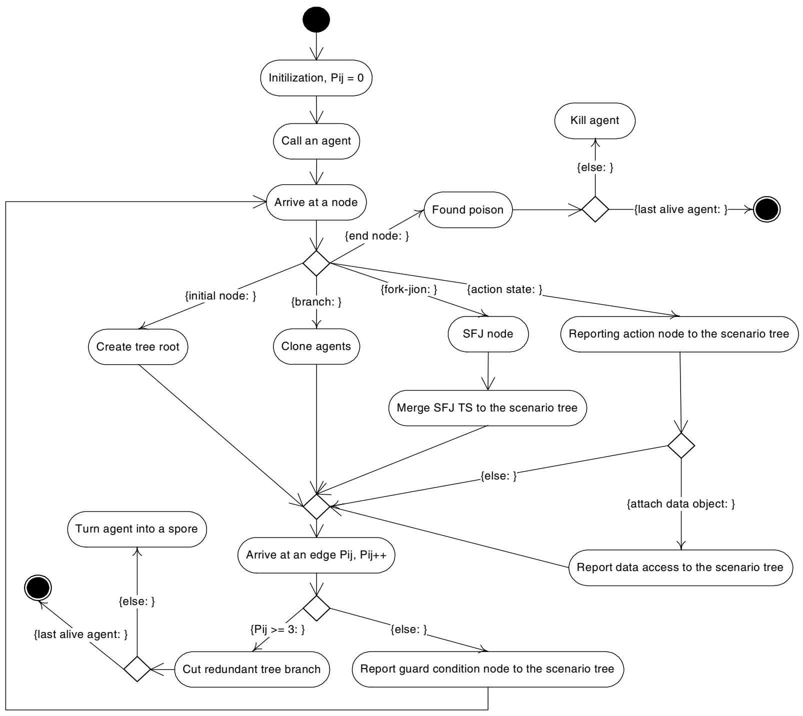


Figure 6 The behavior model of the adaptive agents

forms which are slightly different from the originals. A guard condition which consists of an attribute type and a value are denoted as <Type: value>. In general, there are three kinds of guard conditions in an activity diagram, namely text which is a string description, counter which is an integer mostly used to represent the loop times, and Boolean or Boolean expression which represents a conditional decision. For example, a condition can be denoted as <Boolean: True>.

The pseudo codes of the algorithm, which implement the behavior model shown in Figure 6, are illustrated as follows:

```
/* Initialization */
for every edge (i,j) do
   Pij = 0; /*Set 0 slime level to every
              edge*/
endfor;

Call an agent to traverse the activity
diagram from its initial state.

While (true) do
   Evaluate status at node i;

   /*Report initial node to the scenario
     tree as the root*/
   if (is initial node) do
       Create tree root;
   endif;

   if (found antibiotics) do
       if (is last alive agent) do
          Break;
       else
          Kill agent;
       endif;
   endif;

   /* action state */
   if (access action) do
     Reporting action node to the
     scenario tree;
   endif

   /*Data Reporting*/
   if (access Data) do
     Report data access to the scenario
     tree;
   endif;

   /*Clone agent*/
   If the outgoing edges of the traversed
   node are m, clone m-1 agents to
   traverse the different edges

   /*If arrives at a fork node*/
   if (found fork-join pair) do
     if (the node is a SFJ) do
       Call SFJ algorithm;
     else
       Decompose SNFJ or BNFJ into SFJ;
       Call SFJ algorithm for each SFJ;
       Merge the generated test
       scenarios;
     endif;
     Merge fork-join test scenarios into
     the scenario tree;
   endif;
   Agent moves to an outgoing edge;

   /*Condition reporting*/
   if (found Condition) do
     Report guard condition node to the
     scenario tree;
   endif;
   /*Increased Pij by one when an agent
     arrives at an edge*/
   Pij ++;

   /*loops are only permitted to execute
     two times due to the coverage
     criterion adopted in this paper */
   if (Pij >= 3) do
       Cut redundant tree branch;
       if (is last alive agent) do
          Break;
       else
          Turn the agent into a spore;
       endif;
   endif;

   Agent moves to next node;
endwhile;

/* Generate a test scenario*/
Generate sequence of action states and
transitions formed from the root to a
leaf of the scenario tree.
```

The pseudo codes are straightforward to be followed.

4. Tool support

We have developed a prototype tool called TSGAD (Test Scenarios Generator for Activity Diagrams) using the proposed algorithm to automatically generate test scenarios for given activity diagrams. An activity diagram can be developed using many standard UML tools and exported into a XMI file. TSGAD can directly read a XMI file which includes all UML diagrams, extract the contained activity diagram, and automatically generate test scenarios afterwards.

Due to space limitation, we can not describe TSGAD in details here. However, we use an example to demonstrate the tool. The example shown in Figure 7 is an activity diagram which simulates a test scenario generator. The generator in Figure 7 takes an activity diagram generated using Rational Rose, Poseidon or Visio, and converts the imported diagram into a generic XML format. Afterwards, the generator parses the XML file to derive all test scenarios.

The activity diagram shown in Figure 7 was drawn using Poseidon UML and a XMI file was exported. As shown in Figure 8, the generated XMI file was imported into TSGAD, namely, the intermediate scenario tree was displayed in the left panel; the corresponding activity diagram was displayed in the central panel; while the XML representation for the activity diagram was shown in the right panel. TSGAD automatically generated all test scenarios for the imported activity diagram, as shown in the bottom panel in Figure 8. The details of all generated test scenarios have to be deleted due to space limitation.

5. Related work

In [16], the concept of basic paths (BPs) for the activity diagrams is defined. From BPs, the test scenarios are derived by depth first search. However, a detailed walkthrough of the proposed algorithm shows that some test scenarios are not generated, especially when the test scenarios are derived from the fork-join parts of the activity diagrams. For example, only 6 execution paths are generated for the fork-join structure shown in Figure 2 using the algorithm proposed in [16]. Furthermore, it has been assumed in [16] that the activity diagrams have pairs of branches and merges, and pairs of forks and joins; a fork in an activity diagram can only have two outgoings. Obviously, these assumptions limit the applicable scope of the algorithm proposed in [16]. The algorithm proposed in this paper removes most of the constraints imposed in [16], and the proposed algorithm is more efficient due to the use of the adaptive agents.

An algorithm using anti-ant-like agents is given in [13] to generate thin threads from the activity diagrams. While the algorithm proposed in [13] doesn't rely on the assumptions used in [16], the efficiency of the test thread generation is not optimal. For example, a group of ants is used to explore an activity diagram; each ant starts from the initial state of the activity diagram such that some parts of the activity diagram are explored many times. In contrast, the algorithm proposed in this paper uses one agent to start the initial exploration.

More agents are adaptively reproduced if and only if it is necessary. Therefore, redundant exploration of the activity diagrams is avoided.

Similar to [16], the algorithm proposed in [13] can not generate complete test scenarios for fork-join pairs in general. In comparison, the algorithm proposed in this paper can generate all test scenarios for these fork-join structures described in Section 2.

6. Conclusion

This paper presented an automated approach to generated test scenarios from the UML activity diagrams. Using the developed algorithm, an adaptive

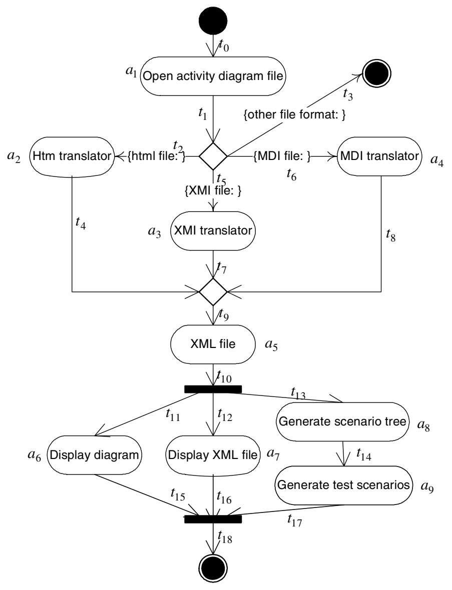
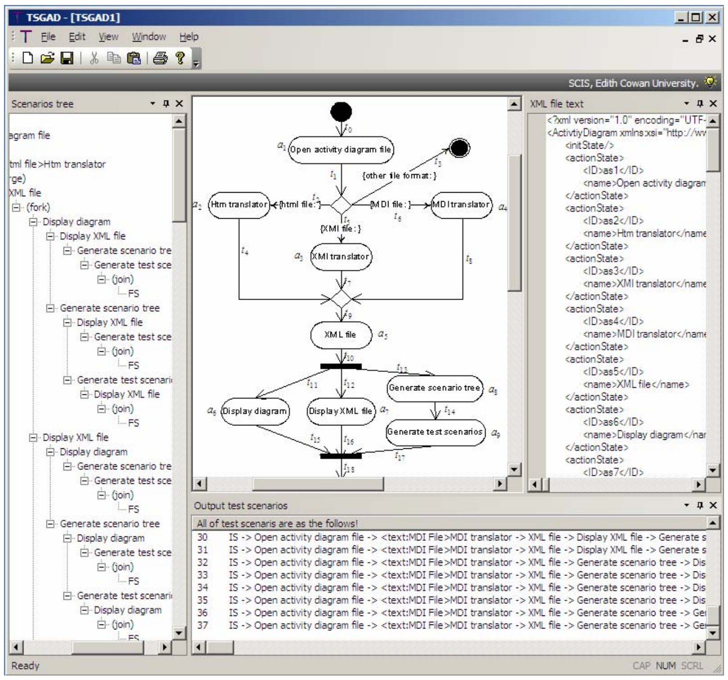


Figure 7 An example



```
agram file
tml file>Htm translator
rge)
XML file
⊟ (fork)
   ⊟ Display diagram
      ⊟ Display XML file
         ⊟ Generate scenario tre
            ⊟ Generate test sce
               ⊟ (join)
                  FS
      ⊟ Generate scenario tree
         ⊟ Display XML file
            ⊟ Generate test sce
               ⊟ (join)
                  FS
         ⊟ Generate test scenari
            ⊟ Display XML file
               ⊟ (join)
                  FS
   ⊟ Display XML file
      ⊟ Display diagram
         ⊟ Generate scenario tre
            ⊟ Generate test sce
               ⊟ (join)
                  FS
      ⊟ Generate scenario tree
         ⊟ Display diagram
            ⊟ Generate test sce
               ⊟ (join)
                  FS
         ⊟ Generate test scenari
            ⊟ Display diagram
               ⊟ (join)
                  FS
```

```
<?xml version="1.0" encoding="UTF-
<ActivtiyDiagram xmlns:xsi="http://wv
    <initState/>
    <actionState>
        <ID>as1</ID>
        <name>Open activity diagram
    </actionState>
    <actionState>
        <ID>as2</ID>
        <name>Htm translator</name
    </actionState>
    <actionState>
        <ID>as3</ID>
        <name>XMI translator</name
    </actionState>
    <actionState>
        <ID>as4</ID>
        <name>MDI translator</name
    </actionState>
    <actionState>
        <ID>as5</ID>
        <name>XML file</name>
    </actionState>
    <actionState>
        <ID>as6</ID>
        <name>Display diagram</nar
    </actionState>
    <actionState>
        <ID>as7</ID>
```

```
All of test scenaris are as the follows!
30    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Display XML file -> Generate s
31    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Display XML file -> Generate s
32    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Generate scenario tree -> Dis
33    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Generate scenario tree -> Dis
34    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Generate scenario tree -> Dis
35    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Generate scenario tree -> Dis
36    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Generate scenario tree -> Ge
37    IS -> Open activity diagram file -> <text:MDI File>MDI translator -> XML file -> Generate scenario tree -> Ge
```

Figure 8 Demonstration of TSGAD

agent can effectively explore the UML activity diagrams and automatically generate test scenarios.

Our approach has the following advantages: 1) the UML activity diagrams exported by UML tools are directly used to generate test scenarios, and the whole generation process is fully automated; 2) redundant exploration of the activity diagrams is highly avoided due to the use of adaptive agents, resulting in improved efficiency in the generation of the test scenarios.

References

[1] F. Basanieri, A. Bertolino, and E. Marchetti, “CoWTeSt: A Cost Weighed Test Strategy”, Proc. Escom-Scope 2001, London, 2001.

[2] X. Bai, C. P. Lam, and H. Li, “An Approach to generate the Thin-threads from the UML Diagrams”, Proc. COMPSAC 2004, Hong Kong, 2004.

[3] S. Bennett, S. McRobb and R. Farmer, *Object-Oriented Systems Analysis and Design Using UML*, Second Edition, McGraw-Hill Education, 2002.

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

[5] L. Briand, “On the many ways Software Engineering can benefit from Knowledge Engineering”, Proc. 14th SEKE, Italy, 2002.

[6] L. Briand and Y. Labiche, “A UML-Based Approach to System Testing”, *Software & Systems Modeling*, 1(1), 2002.

[7] K. Doerner and W. J. Gutjahr, “Extracting Test Sequences from a Markov Software Usage Model by ACO”, LNCS, Vol. 2724, pp. 2465-2476, Springer Verlag, 2003.

[8] J. Horgan, S. London, and M. Lyu, “Achieving Software Quality with Testing Coverage Measures”, *IEEE Computer*, 27(9), 1994.

[9] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing computer software*, 2nd Edition, John Wiley & Sons, 1999.

[10] Y. Kim and C. R. Carlson, “Scenario Based Integration Testing for Object-Oriented Software Development”, Proc. of the Eighth Asian Test Symposium, Shanghai, 1999.

[11] C. P. Lam, M. C. Robey and H. Li, "Application of AI for Automation of Software Testing", *Proc. SNPD03*, Germany, 2003.

[12] H. Li and C. P. Lam, “Optimization of State-based Test Suites for Software Systems: An Evolutionary Approach”, *Int. J. Computer & Information Science*, 5(3), 2004.

[13] H. Li, and C. P. Lam, “Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams”, Proc. TESTCOM 2005, LNCS 3502, Montreal, 2005.

[14] M. J. McBride, P. Hartzell, and D. R. Zusman, Motility and Tactic Behavior of Myxococcus xanthus, Myxobacteria II (M. Dworkin and D. Kaiser eds.), American Society for Microbiology, Washington, 1993.

[15] P. McMinn and M. Holcombe, “The State Problem for Evolutionary Testing”, Proc. GECCO 2003, 2003.

[16] 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”, Proc. APSEC’04, 2004.

[17] E. J. Weyuker, “Testing Component-Based Software: A cautionary Tale”, *IEEE Software,* 15(5), 1998.