Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams

Huaizhong Li and C. Peng Lam

School of Computer and Information Science, Edith Cowan University,
Mt. Lawley, WA 6050, Australia
{h.li,c.lam}@ecu.edu.au

Abstract. The problem of generating the test cases is one of the most important issues in the software testing research and practice. Test threads, especially the thin-threads which are the usage scenarios in a software system, are frequently used to generate test cases for the scenario-based software testing. However, the derivation of the test threads is usually a manual and labor-intensive task. In this paper, we propose an automated approach using anti-ant-like agents to directly generate test threads from the UML artifacts. The generated test threads can then be used to generate and to prioritize the test cases for scenario-based software testing.

1 Introduction

Recently, great amount of attentions have 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, [9] 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. There are two important problems which are generally associated with the scenario-based testing techniques, namely the generation of the test scenarios [1, 3], and the prioritization of the testing scenarios [1, 4].

Properly generated test scenarios are essential for the scenario-based software testing to achieve the test adequacy and to guarantee the software quality. Test thread derivation, especially thin-thread derivation is a frequently used approach for the generation of the test scenarios [1, 3, 5, 15]. Thin-threads, in the forms of thin-thread trees and associated condition trees, can be derived from the scenarios-based business model [1, 5] or directly from the UML artifacts [3], and then test scenarios can be generated from the thin-threads. The generated test threads can also be used to prioritize the test cases for scenario-based software testing [4]. Additional data object tree can be generated to assist in analyzing the content-dependencies which may lead to couplings between the test scenarios [3]. One main problem with the generation of thin-threads is that the generation procedures are either manual/labor-intensive [1, 5], or can not be fully automated [3].

It is well-known that the development of techniques which support the automation of software testing will result in significant cost savings. Recently, the application of Artificial Intelligence (AI) techniques in Software Engineering (SE) emerges as an

F. Khendek and R. Dssouli (Eds.): TestCom 2005, LNCS 3502, pp. 69 – 80, 2005.

area of research that brings about the cross-fertilization of ideas across two domains [8]. It has been identified that one of the SE areas with a more prolific use of AI techniques is software testing [18]. The focus of these techniques involves the applications of genetic algorithms (GAs), for examples [19] and [21]. Recently, efforts have been made to apply Ant Colony Optimization (ACO) algorithms to software testing [11, 20, 21]. However, none of the reported investigations using ACO approaches addresses the generation of test threads from the UML artifacts for scenario-based software testing.

ACO simulates the behavior of real ants. The first ACO technique is known as Ant System [12] and it was applied to the traveling salesman problem. Since then, many variants of this technique have been produced. ACO can be applied to generate solutions for combinatorial optimization problems. The artificial ants in the algorithm represent the stochastic solution construction procedures which make use of (1) the dynamic evolution of the pheromone trails that reflect the ants' acquired search experience; and (2) the heuristic information related to the problem in hand, in order to construct solutions.

Using AI techniques, especially using ant-like agents, provides a potential avenue to automate the generation of test threads for scenario-based software testing. However, the original ACO algorithms as presented in [12] and [13] can not be directly used to tackle the problem of generating test threads from the UML artifacts, as the standard ACO ants are not designed to tackle the graphs which can be converted from the UML diagrams.

In this paper, we propose to use anti-ant-like agents to automatically generate test threads directly from the existing UML activity diagrams. The details of our approach are presented in the next section.

2 Generating Test Threads from the UML Activity Diagrams

Before presenting the details of our approach, we briefly review the principles underlying the representation of the thin-thread tree, the condition tree, and the data-object tree, as well their relationship with the UML activity diagrams.

2.1 The Three Trees

The UML use cases are the good sources for the derivation of the software testing requirements, because they represent high level functionalities provided by the system to the end-users. The use cases are usually not independent. They may have the Extend and the Include dependencies, and the sequential dependencies [7, 10] which stem from the logic of the supported business workflows. The sequential dependencies between use cases can be represented by activity diagrams for all the actors in the system. As the activity diagrams are relatively easy to be interpreted, such a representation facilitates the identification and visualization of these dependencies viewed by the application domain experts. Thin threads can be extracted from the system level activity diagrams.

A thin thread and a use case serve similar functionality, i.e., they both describe system scenarios. However, a thin thread contains more information than a use case, and thin threads for an application are organized into a tree style which is suitable for various analyses such as dependency analysis, risk analysis, traceability analysis, coverage analysis, completeness and consistency checking, and test scenario/test case generation [1]. Thin-threads that share certain commonalities can be grouped together to form a thin-thread group. Such grouping can be recursive, i.e., a collection of lower level thin-thread groups that share some commonalities can be further grouped to form a higher level thin-thread group. All thin-threads and thin-thread groups can be arranged hierarchically to form a thin-thread tree. Furthermore, conditions are generally associated with each thin thread or each thin thread group to identify their activation constraints [1]. A thin-thread can only be activated if its affiliated conditions are satisfied. The conditions can also be grouped and organized into a tree style.

On the other hand, the UML activity diagrams can contain data storage objects which can be read and/or updated by sub-scenarios. These data objects can affect or be affected by the associated conditions. For example, a multi-processing system or an interactive system may experience the racing problem in which the execution result is affected by the execution sequencing of the sub-scenarios, if two or more sub-scenarios update the same data objects in certain situations. Similar to the conditions, the data storage objects can also be classified and organized into a tree style. However, there is a difference between the classification of the data objects and that of the conditions, namely the hierarchy of the data objects closely resembles a normal data storage structure in a relational database. The top level of a data object hierarchy contains the data objects, and a leaf node only contains part of a data object since different sub-scenarios may operate on the different parts of the same data object. The operation attributes, namely reading or updating, are assigned to the leaf nodes to help identifying the data dependencies between the thin-threads. The data object tree was not part of the standard thin-thread based approaches reported in literature [1, 5, 22]. It was proposed in [3] to extend the thin-threads to capture the important content-dependent coupling relationships between the thin-threads.

Consequently, thin-threads, conditions and data objects can all be arranged hierarchically to form the thin-thread tree, the condition tree and the data-object tree, respectively. There are complex relationships amongst the three trees. A thin thread is composed of a group of sequential sub-scenarios, with each sub-scenario associated with one or more conditions in the condition-tree and one or more data object in the data-object tree. The thin-threads may share common sub-scenarios, conditions and data-objects.

Next we present the details of the proposed approach which aims at automating the generation of the trees.

2.2 Using Anti-Ant-like Agents to Build the Three Trees

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 a directed 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 extended activity diagram, namely an ATM machine, is shown in Figure 1. This activity diagram contains a data object *Account* which can be accessed by various activity nodes.

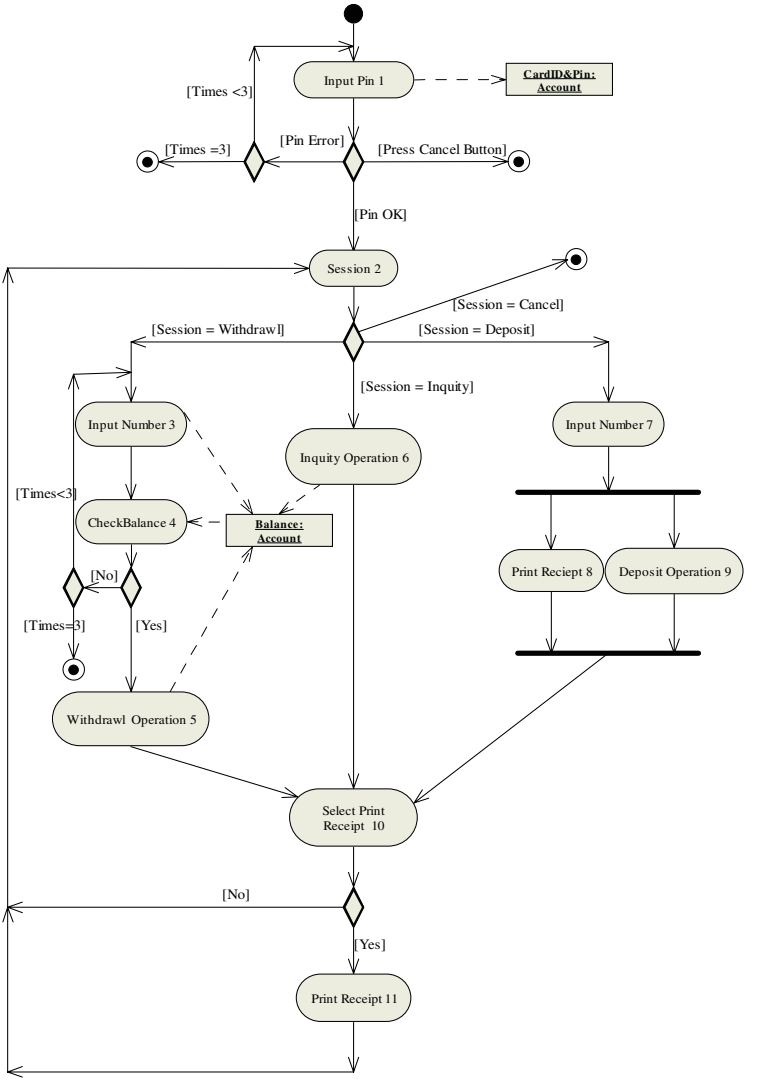

Fig. 1. An ATM Activity Diagram

An activity graph is a directed, dynamic graph in which the activity edges may only become accessible after the evaluation of their guards. It is difficult to apply the original ACO algorithms directly to this type of dynamic graphs to generate test threads.

Inspired by ACO algorithms, we consider the problem of sending a group of ant-like agents to cooperatively search a directed graph G. The objective of the ant exploration is to build the three trees as discussed in the previous sub-section.

The behavior of an artificial ant in our approach is governed by a state machine diagram illustrated in Figure 2. An artificial ant at a node in our paradigm can sense the pheromone trails on the edges emanating from the current vertex, and leave pheromone trails over the edges when it moves between the nodes.

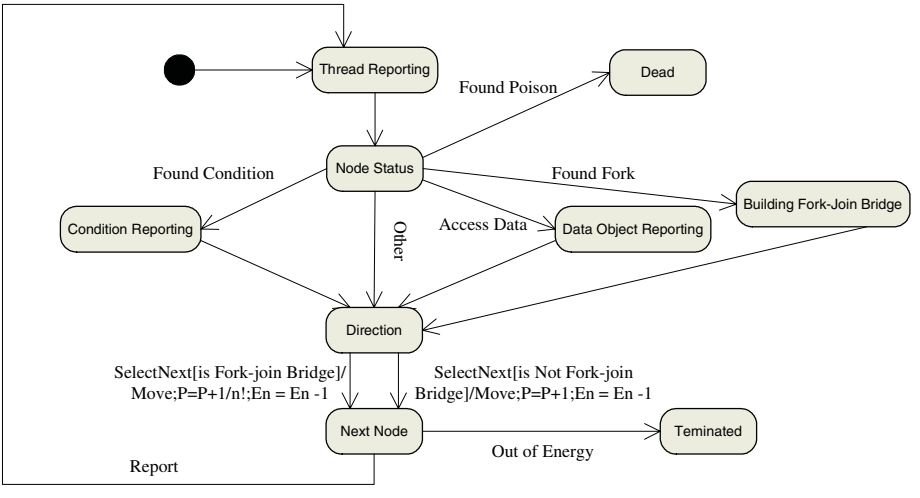

Fig. 2. Behavior of an Artificial Ant - The State Machine Diagram

Unlike the approach in [3], it is not necessary for the current framework to convert a UML activity diagram into an activity hypergraph first and then process the convert hypergraph which were two steps that could not be fully automated in [3]. The UML activity diagrams, in the form of XMI files exported from common UML tools can be directly used to generate the trees in the proposed approach. Graph conversion from the activity diagram, if necessary, is done on the fly instead.

There are two special sets of nodes in the activity diagram which need to pay special attention:

- *The final nodes in an activity diagram are considered as the poisoned food sources for the artificial ants.* An artificial ant is killed if it finds the poisoned food.
- *A fork node and its associated join node are considered as the two banks of a river, every path between the fork-join nodes is considered as a pontoon.* An artificial ant can not cross the river without building a pontoon bridge[1] over the river first. Every pair of fork/join nodes and all the nodes between the pair will be converted on the fly to execution sequences, called Fork-Join

[1] A type of temporary bridge which is quickly built using floating pontoons for the passage of troops.

Bridges, by the artificial ants. The details of Fork-Join Bridges will be discussed late.

In our framework, an artificial ant is powered by limited energy. An ant is terminated if it runs out of energy. The main purpose for the introduction of power consumption for the artificial ants is to avoid the situations in which an artificial ant runs into a cyclic loop, or in which an ant is stalled in a part of the activity diagram.

We now present the algorithm for the proposed ant exploration approach:

Algorithm

The pseudo codes of the proposed algorithm are illustrated as following:

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

/* Exploration of a group of m ants */
for k = 1 to m do
  EN_k = Energy; /*Charge every ant with default energy*/

  i = 0; /*Every ant starts from the initial node*/

  while ( EN_k > 0 ) do

     /*Thread reporting*/
     Report threads to the thread tree;

    Evaluate status at node i;

    if (Found Poison) do
       Kill ant;
       Break;
    endif;

     /*Condition reporting*/
     if (Found Condition) do
        Report conditions to the condition tree;
     endif;

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

     /*If arrives at a fork node*/
     if (Found Fork) do
       Building Fork-Join Bridge;
     endif;

     /*Not every edge is freely accessible*/
     Get access conditions for emanating edges from vertex i;

    Evaluate pheromone levels on all emanating edges;
```

```
    /*Find the destine node d which has the minimum pheromone
level, random selection if multiple*/
    Find min P_id;

    Take access conditions on edge (i,d);

    /*Move to the destine node*/
    i = d;

    /*Each move consumes energy*/
    EN_k = EN_k - 1;

    /*Update pheromone over the traversed edge*/
    if (is Fork-join Bridge) do
       P_id = P_id +1/n!; /*n pontoons*/
    else
       P_id = P_id + 1;
    endif;

  endwhile;

  if P_ij >= 1 for every edge (i,j) do
    Stop; /*Every edge has been traversed*/
   endif;

endfor;
```

The above pseudo codes are derived from the state machine diagram in Figure 2 to reflect an artificial ant's behavior in exploring the activity graphs. Similar to ACO, pheromone trails on edges are used to guide an artificial ant in selecting its direction for next move. However, unlike ACO and the real ants, our artificial ants exhibit repulsive behavior as pheromone trails in our approach are used in such a way that an ant will favor the unexplored or less-explored edges. This results in effective exploration of the activity diagrams, as the addressed problem here is the generation of various test threads which requires effective coverage of all activity edges instead of an optimal path achieved by original ACO algorithms.

Since the artificial ants in our framework exhibit repulsive behavior which is contrary to that of the real ants, we may better name our artificial ants as anti-ants. However, for simplicity, we will still use the name "ants" to call our artificial agents which actually exhibit anti-ant behavior.

The pseudo codes are straightforward to be followed. However, two segments of the pseudo codes, namely Building Fork-Join Bridge and Reporting need to be further explained.

Building Fork-Join Bridge

When an artificial ant arrives at a fork river bank, it has to utilize the pontoons between the two river banks to build a pontoon bridge over the river in order to cross the river. Assume that there are *n* pontoons, and then the procedure to build a fork-join bridge for an ant is:

1. Set $k = 1$;
2. From the remaining pontoons, find the k-th pontoon which has the minimum pheromone level on the first edge; randomly select a pontoon if there are multiple candidates with same minimum pheromone level;
3. Deposit pheromone level $1/k$ to the first edge of the k-th pontoon;
4. If $k = n$, sequentially connect all pontoons in the respective order to form a pontoon bridge; otherwise set $k = k + 1$ and go to step 2;
5. Temporarily replace all the inclusive nodes between the fork node and the join node in the activity diagram with the fork-join bridge constructed by the current ant.

For example, for the fork-join river in the ATM example shown in Figure 1, two consecutive ant explorations build two fork-join bridges as illustrated in Figure 3. Note that an ant deposits $1/n!$ pheromone level over each edge which it traverses on the fork-join bridge.

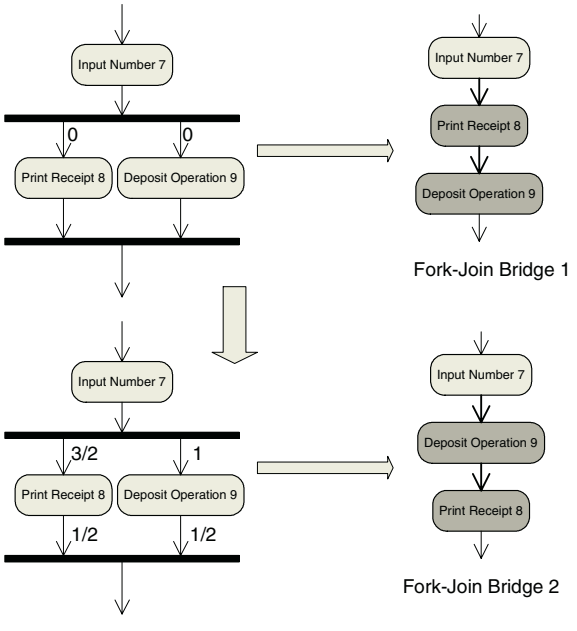

Fig. 3. Building a Fork-Join Bridge

The bridge building procedure ensures that every possible execution sequence combination of the paths between the fork node and the join node will be exercised by the proposed algorithm, and all corresponding traces will be recorded in the three trees. However, the bridge building procedure alone can not guarantee that every activity edge in a path between the fork and the join nodes will be visited by at least one ant. Therefore, the $1/n!$ pheromone level deposition is introduced which ensures

that if there is an unexplored activity edge between the fork and the join nodes, the proposed ant exploration algorithm will not stop. Further exploration of other ants over the paths between the fork and join nodes will favor those edges which have not been fully explored. The number of ants *m* in the proposed algorithm can be increased to allow more exhaustive exploration. Eventually all activity edges between the fork and the join nodes will be visited at least once which serves as one of the necessary conditions for the termination of the ant exploration algorithm.

Reporting

For simplicity, we only discuss thread reporting here. Condition reporting and data reporting can be tackled in similar ways.

In the exploration of a UML activity diagram, an ant frequently reports its trace to a thin-thread tree. The completed trace of an ant, which is an execution thread, is represented as a branch in the thin-thread tree, as illustrated in Figure 4. When next ant enters and explores the activity diagram, its trace is also reported to the same thin-thread tree. However, for compactness, the part of the new ant's trace which overlaps an existing trace is merged with the existing trace to form the trunk, while the different part is allowed to branch away from the trunk, as shown in the right hand side of Figure 4.

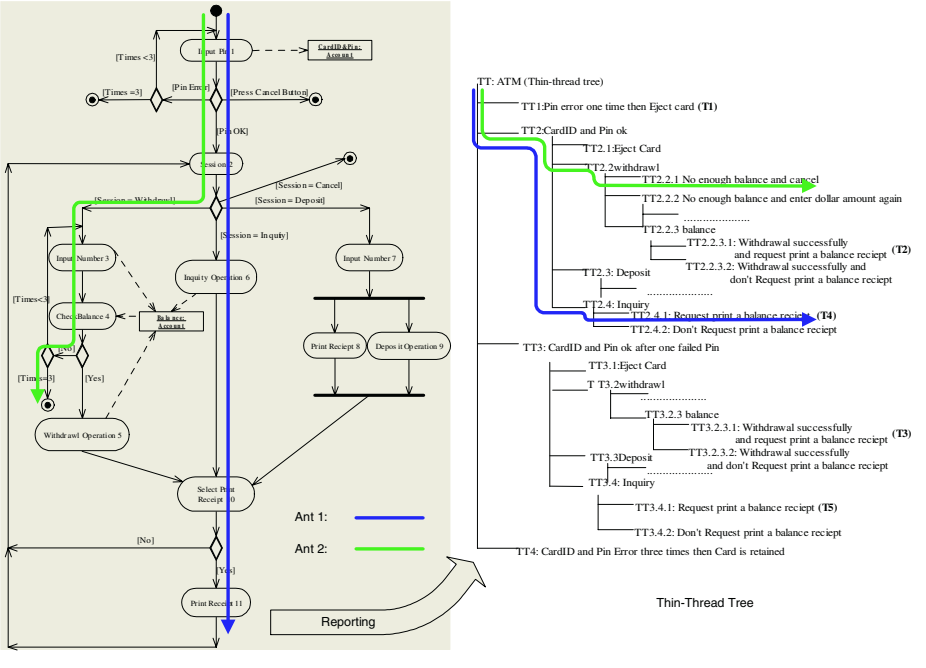

Fig. 4. Thread Reporting

Exploration a UML activity diagram using multiple ant-like agents will result in the automatic generation of three tree type structures, namely the thin-thread tree, the associated condition tree, and the associated data object tree. For the ATM example in Figure 1, application of the proposed approach results in the three trees which are partly shown in Figure 5. The three trees are accordant with the ones reported in [3].

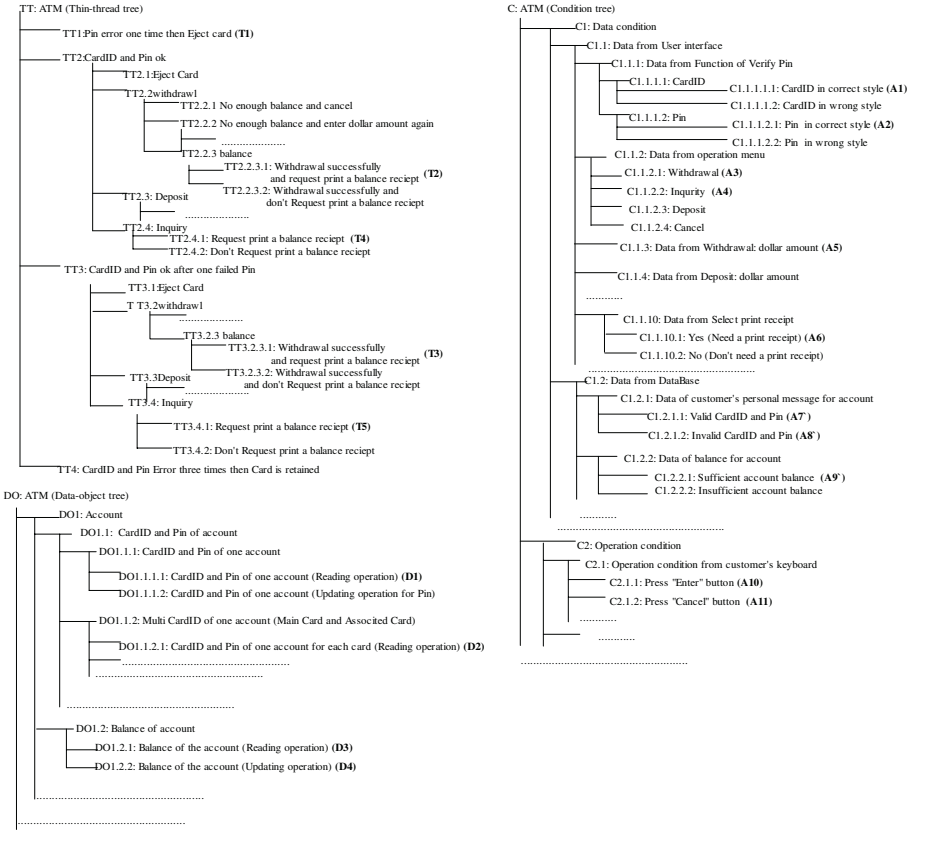

Fig. 5. The Three Trees for the ATM Example

It is possible that some variations may be adopted for the proposed algorithm:

- Use more sophisticated and complicated pheromone updating rules, or use evaporating pheromone deposit.

However, unlike the original ACO algorithms where convergence to an optimal path is desired, the proposed algorithm doesn't encourage cyclic exploration for the artificial ants. In contrary to the original ACO algorithms, the pheromone trails in our approach is used to discourage an artificial ants from exploring an edge which has already been well explored. We believe that the simple pheromone updating rules should serve our purpose well. Adoption of more sophisticated pheromone updating

rules, or using evaporating pheromone deposit may complicate the algorithm without significant improvement. However, further research is required and is being carried out to verify this claim.

- Use goal-oriented approach to guide the ants to effectively explore these unexplored edges.

A goal-oriented evolutionary approach has been proposed in [18] for optimization of state-based test suites for software systems. In the current framework, the artificial ants used to explore the activity diagrams are simple memoryless creatures, they can not pre-fetch the future pheromone trails, and are unable to back-trace. Further research will be demanded to exploit the possibility of using ant-like simple agents in goal-oriented approach for the generation of test threads.

- Deploy ants to randomly assigned initial nodes to start exploration.

While ants can be deployed to random locations to start their exploration, the traces they create may not be meaningful in the sense of test threads. Thus the details of this variation will not be discussed further in this paper.

While the proposed algorithm works well for the exploration of UML activity diagrams of the ATM example scale, further experiments will be performed to verify the efficiency of the proposed algorithm for large scale activity diagrams. Results will be reported in sequential reports.

3 Conclusion

This paper extends the previous work in generation of test threads for software testing. In this paper, we propose to use anti-ant-like agents to automatically generate the thin-threads from the UML artifacts. Our approach has the following advantages: 1) the process to generate the thin-threads is simplified because the UML artifacts are directly used; 2) the generation process is *fully* automated; 3) redundant exploration for the test threads is avoided due to the use of the anti-ant-like ants.

References

1. Assistant Secretary of Defense for Command, Control, Communications, and Intelligence (ASD C3I), *End-to-End Integration Test Guidebook*, 2000.
2. F. Basanieri, A. Bertolino, and E. Marchetti, “CoWTeSt: A Cost Weighed Test Strategy”, *Proc. Escom-Scope 2001*, London, 2001.
3. 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.
4. X. Bai, H. Li, and C. P. Lam, “A Risk Analysis Approach to Prioritize UML-Based Software Testing”, *Proc. SNPD 2004*, Beijing, 2004.
5. X. Bai, W. T. Tsai, R. Paul, K. Feng, and L. Yu, “Scenario-Based Business Modeling”, *IEEE Proc. of APAQS*, 2001.
6. S. Bennett, S. McRobb and R. Farmer, *Object-Oriented Systems Analysis and Design Using UML (Second Edition)*, McGraw-Hill Education, 2002.

7. R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.
8. L. Briand, "On the many ways Software Engineering can benefit from Knowledge Engineering", *Proc. 14th SEKE*, Italy, 2002.
9. L. Briand and Y. Labiche, "A UML-Based Approach to System Testing", *Software and Systems Modeling*, 1(1), 2002.
10. A. Cockburn, "Structuring use cases with goals", http://alistair.cockburn.us/.
11. 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.
12. M. Dorigo, V. Maniezzo, and A. Colorni, **"**Positive Feedback as a Search Strategy", Technical Report No. 91-016*,* Politecnico di Milano, Italy, 1991.
13. M. Dorigo, V. Maniezzo, and A. Colorni, **"**The Ant System: Optimization by a Colony of Cooperating Agents", *IEEE Transactions on Systems, Man, and Cybernetics-Part B,* 26(1), 1996.
14. J. Heumann, "Introduction to Business Modeling Using the Unified Modeling Language (UML)", http://www.therationaledge.com/content/mar_01/m_uml_jh.html.
15. J. Horgan, S. London, and M. Lyu, "Achieving Software Quality with Testing Coverage Measures", *IEEE Computer*, 27(9), 1994.
16. C. Kaner, J. Falk, and H. Q. Nguyen, *Testing computer software*, 2nd Edition, John Wiley & Sons, 1999.
17. Y. Kim and C. R. Carlson, "Scenario Based Integration Testing for Object-Oriented Software Development", *Proceedings of the Eighth Asian Test Symposium*, Shanghai, 1999.
18. C. P. Lam, M. C. Robey and H. Li, "Application of AI for Automation of Software Testing", *Proc. SNPD03*, Germany, 2003.
19. H. Li and C. P. Lam, "Optimization of State-based Test Suites for Software Systems: An Evolutionary Approach", *International Journal of Computer and Information Science*, 5(3), 2004.
20. H. Li and C. P. Lam, "Software Test Data Generation using Ant Colony Optimization", to appear in *Proc. ICCI 2004*, 2004.
21. P. McMinn and M. Holcombe, "The State Problem for Evolutionary Testing", *Proc. GECCO 2003*, 2003.
22. W. T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-To-End Integration Testing Design", *Proc. COMPSAC'01*, Chicago, 2001.
23. E. J. Weyuker, "Testing Component-Based Software: A cautionary Tale", *IEEE Software,* 15(5), 1998.