2005

An Ant Colony Optimization Approach to Test Sequence Generation for State-Based Software Testing

Huaizhong Li
Edith Cowan University

Chiou Peng Lam
Edith Cowan University

An Ant Colony Optimization Approach to Test Sequence Generation for State-based Software Testing

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

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

Abstract

Properly generated test suites may not only locate the defects in software systems, but also help in reducing the high cost associated with software testing. It is often desired that test sequences in a test suite can be automatically generated to achieve required test coverage. However, automatic test sequence generation remains a major problem in software testing. This paper proposes an Ant Colony Optimization approach to automatic test sequence generation for state-based software testing. The proposed approach can directly use UML artifacts to automatically generate test sequences to achieve required test coverage.

1. Introduction

Software testing remains the primary technique used to gain consumers' confidence in the software. Unfortunately, it is always a time-consuming and costly task to test a software system [2]. Obviously, techniques that support the automation of software testing will result in significant cost saving.

The application of artificial intelligence (AI) techniques is an emerging area of research in Software Engineering (SE). A number of published works (for examples [3], [13]) have begun to examine the effective use of AI for SE related activities which are inherently knowledge intensive and human-centered. Four key areas of software development have been identified where the applications of AI will have a significant impact: (1) Planning, monitoring, and quality control of projects, (2) The quality and process improvement of software organizations, (3) Decision making support, and (4) Automation.

The SE area with a more prolific use of AI techniques is software testing. The focus of techniques mainly involved the applications of genetic algorithms (GAs), for examples, [9], [12]. Other AI techniques used for test data generation included the AI planner approach [8] and simulated annealing [14]. However, efficiency of the generation procedure and the feasibility of the generated test data were frequently concerned in the application of the AI techniques.

Recently, Ant Colony Optimization (ACO) has being applied in software testing (see, for examples [4] and [11]). Namely, [4] described an approach involving ACO and a Markov Software Usage model for deriving a set of test paths for a software system, and [11] reported results on the application of ACO to find sequences of transitional statements in generating test data for evolutionary testing. However, the results obtained so far are preliminary, the associated test data generation procedures are difficult to be automated, and none of the reported results directly addresses specification-based software testing.

In this paper we propose to use UML Statechart diagrams and ACO to generate test sequences for state-based software testing. The advantages of the proposed approach are that the UML Statechart diagrams exported by UML tools can be directly used to generate test sequences, and the automatically generated test sequences are always feasible, non-redundant and achieve the required test adequacy criterion.

This paper is structured as follows. Section 2 briefly discusses software testing and ACO. Section 3 presents an ACO approach to test sequence generation, and the conclusion is found in Section 4.

2. Software Testing

There are three main activities associated with software testing: (1) test data generation, (2) test execution involving the use of test data and the software under test (SUT) and (3) evaluation of test results. The key question addressed in software testing is how to select test cases with the aim of uncovering as many defects as possible. Since exhaustive testing is

Proceedings of the Fifth International Conference on Quality Software (QSIC'05)
1550-6002/05 $20.00 © 2005 IEEE

impossible in terms of cost, and no realistic amount of systematic testing can guarantee the absence of errors, the key question is when and how do we determine whether testing has been conducted adequately. In order to reduce cost and time as well as to improve the quality of the software, any extensive testing would require the automation of testing process. Of the three activities mentioned above, test data generation and evaluation of test results are the most labor intensive and thus would benefit most from automation.

The process of test data generation involves activities for producing a set of input test data that satisfies a chosen testing criterion. Horgan [7] has shown that test cases selected on the basis of test adequacy criteria are more effective at discovering defects in the SUT. While it is possible to manually generate an effective set of test cases, the manual generation procedure is very tedious and labor intensive. A cost-effective approach is to automate the test data generation while ensuring that the given criterion is met.

A variety of techniques for test data generation have been developed previously. These techniques can be categorized as structural testing and functional testing. Most existing works in automated test data generation using AI involve the use of GAs and are mainly in the areas of structural testing and temporal behavior testing. The ultimate aim of using genetic algorithms for structural testing is to generate a set of test cases that provides the highest possible coverage of a given structural testing criterion. The test objectives are expressed numerically and are used subsequently to formulate a suitable fitness function that evaluates the suitability of the generated test cases.

ACO is a class of algorithms that simulates the behavior of real ants. The first ACO technique was known as Ant System [5] and was applied to the traveling salesman problem. Since then, many variants have been produced. The ACO algorithms are based on pheromone trails used by the ants which mark out food sources. The trails can be sensed by other ants. ACO is a probabilistic technique that can be applied to generate solutions for combinatorial optimizations problems. The artificial ants in the algorithm represent the construction procedures for the stochastic solutions which make use of (1) the dynamic evolution of the pheromone trails that reflects the ants' acquired search experience; and (2) the heuristic information related to the problem in hand, in order to construct probabilistic solutions.

In order to apply ACO to solve an optimization problem such as test case generation, a number of issues need to be addressed, namely, (1) transformation of the testing problem into a graph; (2) a heuristic measure for measuring the "goodness" of paths through the graph; (3) a mechanism for creating possible solutions efficiently and a suitable criterion to stop solution generation; (4) a suitable method for updating the pheromone; and (5) a transition rule for determining the probability of an ant traversing from one node in the graph to the next.

In the next section, we present an ACO approach to automatically generate test sequences from UML Statechart diagrams for state-based software testing.

3. An ACO Approach to Test Sequence Generation

State-based testing is frequently used in software testing. There are two major problems commonly associated with state-based software testing: (1) some of the generated test cases are infeasible; (2) inevitably many redundant test cases have to be generated in order to achieve the proper testing coverage required by test adequacy criteria. For the first problem, approaches using code execution or model execution techniques have been developed to exclude the infeasible paths. However, to our knowledge, no systematic strategy has been reported to successfully deal with both problems.

The UML Statechart diagrams have been extensively used in state-based software testing. In order to define test adequacy criteria for state-based software testing using the UML Statechart diagrams, the Statechart diagrams have to be flattened to remove all hierarchy and concurrency [2]. It has to be emphasized that the Statechart flattening process is merely used for testing purpose, a flattened Statechart diagram is not necessary a semantic equivalence to the original Statechart diagram.

It is well-known that all-state test coverage requirement is commonly used in state-based software testing. A test suite is said to achieve all states coverage if every state is accessed at least once under test. A test suite for state-based software testing consists of a set of test sequences in the form

$$S_A \rightarrow S_B \rightarrow S_C \rightarrow S_D \rightarrow S_A \rightarrow S_D \rightarrow S_A \rightarrow S_C \rightarrow S_B,$$

or alternatively, $\{S_A, S_B, S_C, S_D, S_A, S_D, S_A, S_C, S_B\}$ for short notation, where S_A, S_B, S_C, S_D are the states in the corresponding UML Statechart diagram, and $\rightarrow$ represents a transition between the two states.

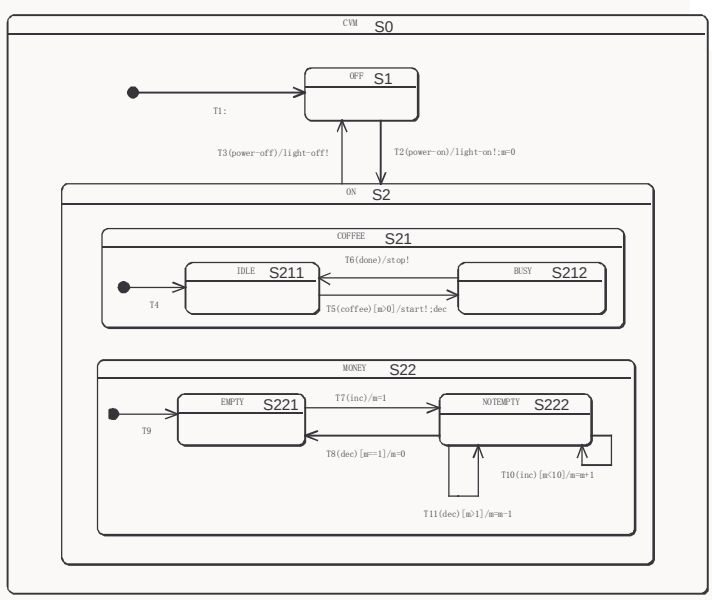

Figure 1 The Coffee Vendor Machine (CVM) Statechart

The proposed approach addresses the automatic generation of test sequences from the UML Statechart diagrams for state-based software testing. The all-states test coverage is used as test adequacy requirement. Specifically, two requirements have been imposed that the generated test suite has to satisfy:

- *All-state coverage*
- *Feasibility* – Each test sequence in the test suite represents a feasible path in the corresponding Statechart diagram

We now proceed to develop the proposed approach.

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 flattened UML Statechart can be viewed as a directed graph where the vertices are the states of the Statechart diagram, and the edges are the transitions between the states.

We have developed a tool to automatically convert a Statechart diagram to a directed graph. For example, a well-known UML Statechart diagram, the Coffee Vendor Machine (CVM) which is frequently used as a benchmark problem for state-based testing, can be converted into a directed graph CVM = (S, T), where S is the state set and T is the transition set. The original CVM Statechart diagram and the converted graph are shown in Figure 1 and Figure 2 respectively. In the following, we will use the CVM example to help demonstrating the approach.

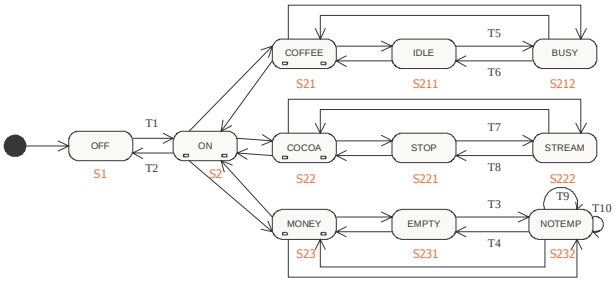

Figure 2 Converted CVM graph CVM = (S, T)

Although hierarchy and concurrency have to be removed from the flattened UML Statechart diagrams, and hence also from the converted graphs, it should be noted that our approach can implicitly deal with testing of concurrency. This is due to the capability of using multiple ants to simultaneously explore the converted graphs.

The converted graphs are directed, dynamic graphs in which the edges (transitions in Statechart sense) may dynamically appear or disappear based on the evaluation of their guards. Therefore, we need to consider the problem of sending a group of ants to cooperatively search a directed graph G. It has been observed that the original ACO algorithms in [5], [6] are difficult to be applied to this type of directed and dynamic graphs to generate test data for the corresponding testing problems. An alternative algorithm has to be proposed in order to use ants to search the graphs for test sequence generation.

Similar to [16], the ants in our paradigm can sense the pheromone traces at the current vertex and those directly connected neighboring vertices, and leave pheromone traces over the vertices.

Each ant at a vertex α of the graph is associated with a four tuple ($\mathbf{S}$, $\mathbf{D}$, $\mathbf{T}$, $\mathbf{P}$):

- *Vertex Track Set* $\mathbf{S}$ = $\{S_i\}$ keeps a vertex track of the ant's walking history
- *Target Set* $\mathbf{D}$ indicates those vertices which are always connected to the current vertex α. For the Statechart diagrams, target sets only exist for the super-states of the composite states, and the target set for a super-state contains the current status of the composite state. For example, the ON super-state for the CVM graph contains the information ON(COFFEE, MONEY). Therefore, the target set for vertex ON is {COFFEE, MONEY}. The target set for vertex COFFEE is either {IDLE} or {BUSY}, but not both because the super-state for these two sub-states only keeps the current status of the composite state
- *Connection Set* $\mathbf{T}$ = $\{T(V_i)\}$ represents the direct connections of the current vertex α with the neighboring vertices. Direct connection means that there is only one directed edge from the current vertex to the destination vertex. $\mathbf{T}$ also documents all the edges spanning from the current vertex. $T(V_i) = 0$ means that the two vertices α and V_i are always connected, $T(V_i) = 1$ means that the two vertices appear to be connected for the current ant at the current vertex α, and $T(V_i) = -1$ indicates that the two vertices are not connected for the current ant, at the current vertex α and for the current time. For the corresponding UML Statechart diagram, the following situations appear:
  - $T(V_i) = 0$ means that either V_i is contained in the target set for α, or α is contained in the target set for V_i. This represents two vertices which are a super-state and its targeted sub-state;
  - $T(V_i) = 1$ means that the transition between the two states is evaluated to be feasible;
  - $T(V_i) = -1$ means that there is no transition between two states α and V_i, or the transition between the two states is infeasible, or V_i is not in the target set of the vertex α

  For examples, for an ant at the state EMPTY (S_{221}) in the CVM graph in Figure 2, $\mathbf{T}$ = {$T(S_{22})$, $T(S_{222})$}. If T_7 is feasible, $\mathbf{T}$ = {0, 1}, otherwise $\mathbf{T}$ = {0, -1}; for an ant at the state COFFEE (S_{21}) with target set {IDLE}, $\mathbf{T}$ = {$T(S_2)$, $T(S_{211})$, $T(S_{212})$} = {0, 1, -1} since S_{212} is not contained in the target set for S_{21} at this stage
- *Pheromone Trace Set* $\mathbf{P}$ = $\{P(V_i)\}$ represents the pheromone levels at the neighboring vertices which are feasible to the ant at the current vertex. Unlike other sets, the pheromone left by previous ants over the graph will not vanish, and the succeeding ants will use the remaining pheromone level to adjust their exploration.

Each ant keeps its own $\mathbf{S}$, $\mathbf{D}$, and $\mathbf{T}$, while set $\mathbf{P}$ is left on the graph to be shared by all ants. Ants can sense the pheromone levels on the graph, and modify $\mathbf{P}$ in the exploration of the graph.

The following algorithm is proposed for an ant to explore the directed graph:

Algorithm

1. Evaluation at vertex α
   - *Update the Track* - Push the current vertex α into the track set $\mathbf{S}$
   - *Evaluate Connections* - Evaluate all connections to the current vertex α to determine $\mathbf{T}$. The procedure involves evaluation of all possible transitions from the current states α to other neighbouring states, using the state-transition table associated with the UML Statechart diagram
   - *Sense the Trace* - For the non-negative connections in $\mathbf{T}$, the ant senses and gathers the corresponding pheromone levels P at the other ends of the connections
2. Move to next vertex
   - *Select Destination* - The following prioritized rules are used in ant's selection:
     i) Select the vertex V_i with the lowest pheromone level $P(V_i)$ sensed from the current vertex α
     ii) If vertices V_i and V_j shares the same lowest pheromone level $P(V_i) = P(V_j)$, but $T(V_i) = 0$ and $T(V_j) = 1$, select V_i
     iii) If vertices V_i and V_j shares the same lowest pheromone level $P(V_i) = P(V_j)$ and $T(V_i) = T(V_j)$, randomly select one vertex

     Destination β is the vertex selected using the above rules
   - *Update Pheromone* - Update the pheromone level for the current vertex α to

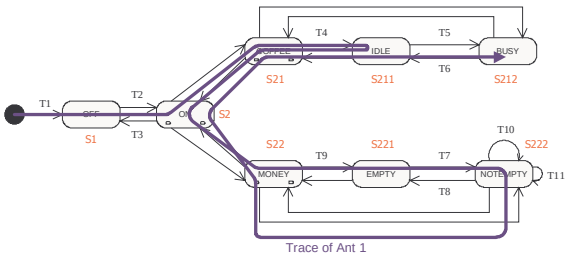

Figure 3 Trace of Ant 1

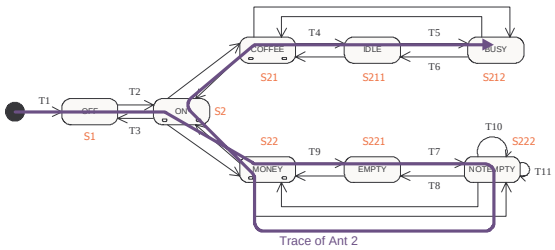

Figure 4 Trace of Ant 2

$P(\alpha) = \max(P(\alpha), P(\beta)+1)$ if $T(\beta) = 1$ or

$P(\alpha) = \max(P(\alpha), P(\beta)+1)+TP$ if $T(\beta) = 0$

Where TP is a high pheromone level which decays in one iteration of the steps, namely, TP quickly decays to 0 before ant's next move at the end of Step 2

- *Move* - Move to the destination vertex β, set α := β, and return to Step 1.

Similar to [16], it can be shown that all vertices can be visited within limited steps (upper bound). The details however are omitted due to space limitation. The algorithm for an ant stops when one of the following two conditions is satisfied:

- The track set **S** contains all vertices of the graph which means the coverage criterion has been satisfied, i.e., all states have been visited at least once;
- The search upper bound has been reached. In this case, the ant fails to find a sequence which achieves the required coverage. More ants will have to be deployed in order to find a solution.

In the above algorithm, TP is used to encourage an ant to perform forward exploration, or equivalently to prevent an ant from immediately moving back to the previously visited vertex. In the Statechart diagram sense, TP prevents an ant from doing redundant moves between a super-state and its sub-states.

Next we demonstrate the proposed algorithm using the CVM given in Figure 1. Although our approach can use multiple ants to cooperatively explore the CVM, for demonstration purpose and for clarity, we only send ants one by one in a sequential manner in the following demonstration.

We first send Ant 1 to walk the directed graph CVM, starting from the default state S_1. Using the proposed algorithm, Ant 1's trace is recorded in the trace set **S**. Table 1 provides details about Ant 1's exploration. Initially Ant 1 has to make a random decision at vertex S_2 according to our algorithm. Assume that Ant 1 randomly selected to move from S_1 to S_{21}, its trace is illustrated in Figure 3 which provides a feasible test sequence for the CVM Statechart:

Test Sequence 1 = {S_1, S_2, S_{21}, S_{211}, S_{21}, S_2, S_{22}, S_{221}, S_{222}, S_{22}, S_2, S_{21}, S_{211}, S_{212}}

One test case, namely Test Sequence 1, satisfies the all-states coverage requirement. However, it may also be observed that Test Sequence 1 as shown in Figure 3 is not the shortest test sequence. If shortest test sequences are required to form the generated test suite, more ants may be deployed sequentially or repeatedly to the graph, or multiple ants can be deployed simultaneously to explore the possibility of getting shorter test sequences for the required coverage.

We continue the simple sequential example. Ant 2 is sent to explore the CVM graph afterwards. Similar to Ant 1, Ant 2 starts from the default state S_1. Since this is a sequential deployment, Ant 1 has left pheromone trace as indicated in the last row of Table 1; therefore, Ant 2 doesn't have to make a random decision at vertex S_2 anymore. It is easy to know that the test sequence created by Ant 2 is

Test Sequence 2 = {S_1, S_2, S_2, S_{22}, S_{221}, S_{222}, S_{22}, S_2, S_{21}, S_{211}, S_{212}}

Test Sequence 2 is the shortest one for all-states coverage requirement using single ant sequentially, as illustrated in Figure 4. Note that if Ant 2 is repeatedly deployed to a fresh CVM graph which does not have pheromone trail, and Ant 2 selects to move to the alternative vertex S_{22} in the random decision, Ant 2's trace will be same as test sequence 2 above.

For Statechart diagrams of the CVM scale, it is often sufficient to consecutively or repeatedly send single ant to explore the converted graphs. In general, for more complicated Statechart diagrams, multiple ants have to be sent to explore the converted graphs simultaneously in order to accelerate the exploration process. Each ant is assigned a unique ID which represents its priority in the cooperative team. Each

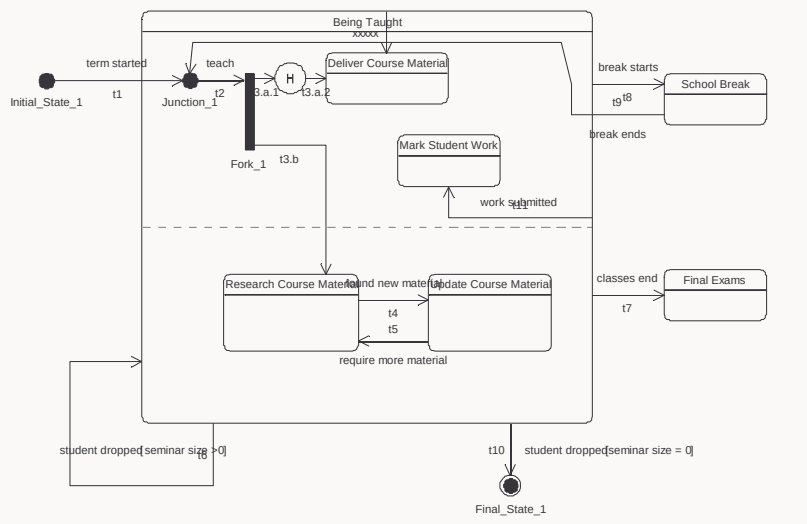

Figure 5 The "Being Taught" Statechart Diagram

ant maintains its own S, D, T sets, but shares with other ants the pheromone set P. The proposed algorithm works for the multiple ants case with only one exception, namely when there are two or more ants at vertex α, they have to leave α according to their priorities in the team. The ant with a higher priority leaves and sets pheromone level for α first, followed by lower priority ants. The final pheromone level which is left over α is the highest one set by all ants.

We have developed a prototype tool called Dynamic Ant Simulator (DAS) using the proposed algorithm to automatically generate test sequences for given Statechart diagrams. A Statechart diagram can be developed using many standard UML tools and exported into a XMI file. The exported XMI file contains the Statechart diagram as well as other UML diagrams. DAS can directly read a XMI file which contains all UML diagrams, extract the Statechart diagram, flatten the extracted Statechart diagram, and convert the flattened Statechart diagram into a directed dynamic graph. A user can then manually or randomly deploy ants into nodes in the directed dynamic graph, and DAS will automatically generate feasible test sequences afterwards.

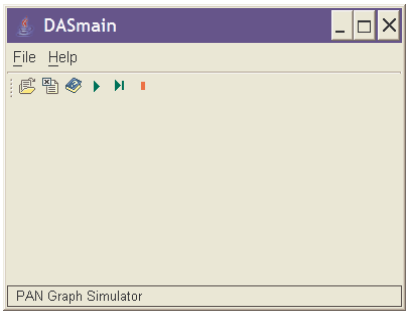

Figure 6 DAS Main Screen

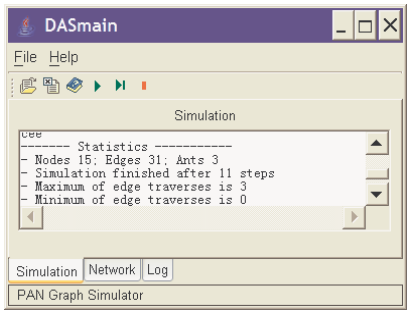

Figure 7 Log Window

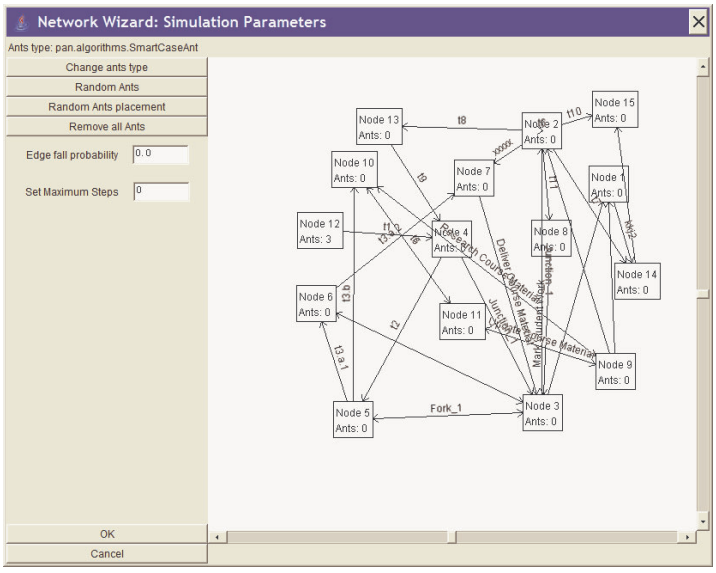

Figure 8 Network Wizard

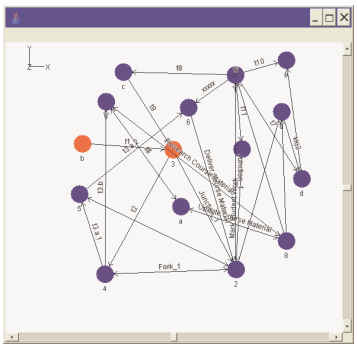

Figure 9 Simulation - Start

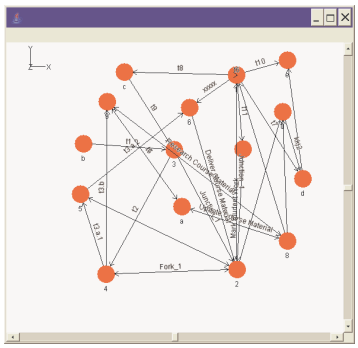

Figure 10 Simulation - End

```
bbbbbbbb
33333333
42244222
95095111
86da6d7c
a2e82e23
------- Statistics ----------
- Nodes 15; Edges 31; Ants 8
- Simulation finished after 5 steps
- Maximum of edge traverses is 8
- Minimum of edge traverses is 0
---------- End ----------
Simulation finished
```

Figure 11 Simulation Log

Due to space limitation, we can not describe DAS in details here. Nevertheless, we use an example to demonstrate DAS. The "Being Taught" Statechart diagram example shown in Figure 5 is taken from [1]. The Statechart diagram was drawn using Poseidon UML and a XMI file was exported which contained both the Statechart and class diagrams.

The DAS main interface is shown in Figure 6. Using the ***File*** menu, the exported XMI file can be loaded into the simulator, and the associated graph is automatically converted and displayed in the Network Wizard window as shown in Figure 8. Ants can then be deployed into the graph randomly or selectively by clicking on a particular node. In Figure 8, 3 ants are deployed to Node 12 which is the initial state in Figure 5. Afterwards, a user can click on the ***Start*** button in the main window to start the generation of test sequences. Ants' exploration activities are visually displayed throughout the generation procedure, as shown in Figure 9 and Figure 10. The Log window shown in Figure 7 keeps ants' traces and provides statistics after the generation procedure is completed.

Figure 11 shows the log produced by deploying 8 ants to Node 12. Each column of the data set forms a test sequence. It was found that for the addressed example, deploying 8 ants to Node 12 produces a test suite which contains the shortest test sequences. Using more ants can not improve the results further.

4. Conclusion

This paper presented an ant colony optimization approach to test sequence generation for state-based software testing. Using the developed algorithm, a group of ants can effectively explore the UML Statechart diagrams and automatically generate test sequences to achieve the test adequacy requirement.

Our approach has the following advantages: (1) the UML Statechart diagrams exported by UML tools are directly used to generate test sequences; (2) the whole generation process is fully automated; (3) redundant exploration of the Statechart diagrams is avoided due to the use of ants, resulting in efficient generation of test sequences.

References

[1] Ambler, S. W., *The Object Primer 3rd Edition: Agile Model Driven Development with UML 2*, Cambridge University Press, 2004.

[2] Binder, R. V., *Testing Object-oriented Systems: Models, Patterns, and Tools*, Addison-Wesley. 2000.

[3] Briand, L. C.,"On the many ways Software Engineering can benefit from Knowledge Engineering", Proc. 14th SEKE, Italy, pp. 3-6, 2002.

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

[5] Dorigo M., Maniezzo, V., Colorni, A., **"**Positive Feedback as a Search Strategy", Technical Report No. 91-016, Politecnico di Milano, Italy, 1991.

[6] Dorigo M., Maniezzo, V., Colorni, A**.,** **"**The Ant System: Optimization by a Colony of Cooperating Agents", *IEEE Transactions on Systems, Man, and Cybernetics-Part B*, Vol. 26, No.1, pp.29-41, 1996.

[7] Horgan, J., London, S., and Lyu, M., "Achieving Software Quality with Testing Coverage Measures", *IEEE Computer*, Vol. 27 No.9 pp. 60-69, 1994.

[8] Howe, A. E., Mayrhauser A. V., and Mraz, R. T., "Test Case Generation as an AI Planning Problem", *Automated Software Engineering*, Vol. 4, pp 77-106, 1997.

[9] Li, H., Lam, C.P., "Optimization of State-based Test Suites for Software Systems: An Evolutionary Approach", *International Journal of Computer & Information Science*, Vol. 5, No. 3, pp. 212-223, 2004.

[10] McMinn, P., "Search-based Software Test Data Generation: A Survey", *Software Testing, Verification and Reliability*, Vol.14, No. 2, pp. 105-156, 2004.

[11] McMinn, P., Holcombe, M., "The State Problem for Evolutionary Testing", Proc. GECCO 2003, LNCS Vol. 2724, pp. 2488-2500, Springer Verlag, 2003.

[12] Pargas, R. P., Harrold, M. J., and Peck, R., "Test-Data Generation Using Genetic Algorithms", *Software Testing, Verification and Reliability*, Vol. 9, pp. 263 - 282, 1999.
[13] Pedrycz, W., Peters, J. F., *Computational Intelligence in Software Engineering*, World Scientific Publishers, 1998.
[14] Tracey, N., Clark, N., .Mander K., and McDermid, N., "A Search Based Automated Test Data Generation Framework for Safety Critical Systems", in *Systems Engineering for Business Process Change (New Directions)*, Henderson P., Editor, Springer Verlag, 2002.
[15] Wagner, I. A., Lindenbaum, M., Bruckstein, A. M., "Distributed Covering by Ant-Robots Using Evaporating Traces", *IEEE Trans. Robot. Auto.*, Vol. 15, pp. 918-933, 1999.
[16] Wagner, I. A., Lindenbaum, M., Bruckstein, A. M., "ANTS: Agents, Networks, Trees, and Subgraphs", Special issue on Ant Colony Optimization (M. Dorigo, G. Di Caro, T.Stützle (eds)), *Future Generation Computer Systems*, Vol. 16, No. 8, pp. 915-926, North Holland, June 2000.

Table 1 Ant 1's exploration details

		S_1	S_2	S_{21}	S_{211}	S_{21}	S_{22}	S_{221}	S_{22}
$\alpha = S_2$	D		{COFFEE,						
	T	-1	0	0	-1	-1	0	-1	-1
	P	1	0	0	0	0	0	0	0
	S	$\{S_1, S_2\}$							
$\alpha = S_{21}$	D		{COFFEE,	IDLE			EMPTY		
	T	-1	0	0	0	-1	-1	-1	-1
	P	1	1+TP	0	0	0	0	0	0
	S	$\{S_1, S_2, S_{21}\}$							
$\alpha = S_{211}$	D		{COFFEE,	IDLE			EMPTY		
	T	-1	-1	0	0	-1	-1	-1	-1
	P	1	1	1+TP	0	0	0	0	0
	S	$\{S_1, S_2, S_{21}, S_{211}\}$							
$\alpha = S_{21}$	D		{COFFEE,	IDLE			EMPTY		
	T	-1	0	0	0	-1	-1	-1	-1
	P	1	1	1	2+2T	0	0	0	0
	S	$\{S_1, S_2, S_{21}, S_{211}, S_{21}\}$							
Details for Ant 1's trace part $S_{21} \rightarrow S_2 \rightarrow S_{22} \rightarrow S_{221} \rightarrow S_{222} \rightarrow S_{22} \rightarrow S_2 \rightarrow S_{21}$ are omitted									
$\alpha = S_{21}$	D		{COFFEE,	IDLE			NOTEMPTY		
	T	-1	0	0	0	-1	-1	-1	-1
	P	1	3+TP	2	2	0	2	1	2
	S	$\{S_1, S_2, S_{21}, S_{211}, S_{21}, S_2, S_{22}, S_{221}, S_{222}, S_{22}, S_2, S_{21}\}$							
$\alpha = S_{211}$	D		{COFFEE,	IDLE			NOTEMPTY		
	T	-1	-1	0	0	1	-1	-1	-1
	P	1	3	3+TP	2	0	2	1	2
	S	$\{S_1, S_2, S_{21}, S_{211}, S_{21}, S_2, S_{22}, S_{221}, S_{222}, S_{22}, S_2, S_{21}, S_{211}\}$							
$\alpha = S_{212}$	D		{COFFEE,	BUSY			NOTEMPTY		
	T	-1	-1	0	-1	0	-1	-1	-1
	P	1	3	3	2	0	2	1	2
	S	$\{S_1, S_2, S_{21}, S_{211}, S_{21}, S_2, S_{22}, S_{221}, S_{222}, S_{22}, S_2, S_{21}, S_{211}, S_{212}\}$							