The State Problem for Evolutionary Testing

Phil McMinn and Mike Holcombe

Department of Computer Science, The University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, UK.
{p.mcminn, m.holcombe}@dcs.shef.ac.uk

Abstract. This paper shows how the presence of states in test objects can hinder or render impossible the search for test data using evolutionary testing. Additional guidance is required to find sequences of inputs that put the test object into some necessary state for certain test goals to become feasible. It is shown that data dependency analysis can be used to identify program statements responsible for state transitions, and then argued that an additional search is needed to find required transition sequences. In order to be able to deal with complex examples, the use of ant colony optimization is proposed. The results of a simple initial experiment are reported.

1 Introduction

Evolutionary testing (ET) is a technique by which test data can be generated automatically through the use of optimizing search techniques. The search space is the input domain of the software under test. ET has been shown to be successful for generating test data for many forms of testing, namely specification testing [9], extreme execution time testing [12] and structural testing [10].

It has also been shown that certain features of programs can inhibit the search for test data, for example flag variables [3, 6]. This paper introduces another such feature: states in test objects. States can cause a variety of problems for ET, since test goals involving states can be dependent on the entire history of input to the test object, as well as just the current input. In addition, guidance must be provided so that statements responsible for state transitions are executed, so as to put the test object into the required states for certain test goals to become feasible. Internal states have hindered test data generation for automotive components used at DaimlerChrysler. The aim of this work is to extend the DaimlerChrysler ET system [14] to enable it to generate test data when presented with such troublesome test objects.

This paper is organized as follows. Section 2 reviews evolutionary testing. Section 3 introduces the state problem with examples. Section 4 discusses the use of data dependency analysis to identify program statements that are responsible for state transitions. Section 5 discusses how this could be applied to the state problem, and argues that an additional search is needed to find required sequences of transitional statements. In the case of simple examples an exhaustive search may be all that is required, however for more complex cases an optimization technique may be needed, and the use of ant colony optimization is proposed. Results of a simple initial ex-

periment are reported. Section 6 then closes with conclusions and outlines future work.

2 Evolutionary Testing (ET)

Evolutionary Testing (ET) uses optimizing search techniques such as evolutionary algorithms to generate test data. The search space is the input domain of the test object, with each individual, or potential solution, being an encoded set of inputs to that test object. The fitness function is tailored to find test data for the type of test that is being undertaken.

This paper discusses the state problem in the context of structural testing. Here the aim is to find test data to execute every structural component of some coverage type, for example all branches of the program's control flow graph, or the execution of every definition-use pair for every variable. In order to retrieve fitness information, the test object must be instrumented.

Previous work [10] has argued that higher levels of coverage are obtained when each structural element of the chosen coverage type is targeted individually as a partial aim. For each partial aim, the minimizing fitness function is made up of two components, namely the approximation level and a branch distance calculation [10, 11]. The approximation level supplies a value indicating how close in structural terms an individual is to reaching the target. For node-oriented coverage types, for example statement coverage, this value is calculated as the number of branching statements lying between branches covered by an individual and the target branch. At the point where the individual diverged away from the target node, a normalized branch distance calculation is computed. This value indicates how close the individual was to evaluating the branch predicate in the desired way. For example if a condition `(x == y)` needs to be executed as true, the branch distance is calculated using `|x-y|`. For the thermostat function in figure 1, and the partial aim where the node 6 must be executed, the fitness values are computed as follows. Individuals reaching node 4 and evaluating the branching condition `heater_on` as false receive an approximation level of 1 and a branch distance of 1 (`1 - heater_on`). On the other hand, individuals reaching node 5 but evaluating the condition as false receive an approximation level of zero, and a branch distance computed using the formula `POWER_THRESHOLD - power`. The value of the fitness function when the target is finally reached is therefore zero.

With path-oriented coverage types, the approximation level is formed from the length of all identical path sections. Branch distances are taken at each point in which the flow of execution diverged from the intended path. These are then accumulated and normalized.

```
   int thermostat (int temp, int pow)
   {
     int heater_on;
1    if (temp < TEMP_THRESHOLD)
2      heater_on = 1;
     else
3      heater_on = 0;
4    if (heater_on)
5      if (pow < POWER_THRESHOLD)
6        heater_on = 0;
7    return heater_on;
   }
```

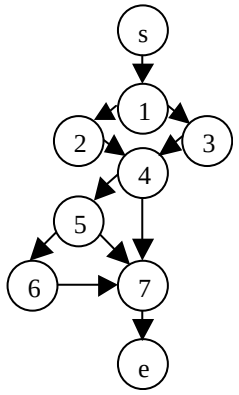

Fig. 1. C Code fragment of the simple thermostat example, with control flow graph (right column) and control flow graph node numbers corresponding to program statements in the left column

3 The State Problem

The state problem occurs with functions at higher system levels that exhibit state-like qualities by storing information in internal variables, which retain their values from one execution of the function under test to the next. Such variables are hidden from the optimization process because they are not available to external manipulation. The only way to change the values of these variables is through execution of statements that perform assignments to them. If such variables can be described as state variables, then these assignments, or definitions, are the transitions of the underlying state machine.

Figure 2 illustrates a function that is a controller for a smoke detector, the style of which mimics real-life code witnessed at our industrial partner. It takes three arguments - the first being the current room smoke level, followed by two arguments used as outputs to signal that the alarm should be switched on or off (the alarm works on a latch whereby after an “on” signal is received; the alarm stays on until it receives an “off” signal). The function is designed to be cycled once every second by the hardware. When the room smoke level becomes higher than a given threshold for a certain period of time, the alarm is raised (lines 13-14). When the room smoke level returns to safe levels for a given time, a special `waiting` flag becomes true (lines 17-18). The alarm then stays on for another 20 seconds (lines 21-22), unless the smoke levels breach acceptable limits again (lines 19 -20). The static storage class is used to declare several local variables. This allows them to maintain their values after the function is executed, however, as they are internal to the function, they cannot be directly optimized by the evolutionary search process.

```
1   const double LEVEL = 0.3;
2   const int DANGER = 5, WAIT_TIME = 20;
3   void smoke_detector(double level,
4                       int* signal_on, int* signal_off)
5   {
6     static int time = 0, off_time = 0;
7     static int detected = 0, alarm_on = 0, waiting = 0;
8     time ++;
9     if (level > LEVEL && detected < DANGER)
10      detected ++;
11    else if (detected > 0)
12      detected --;
13    if (!alarm_on && detected == DANGER)
14    { alarm_on = 1; *signal_on = 1; }
15    if (alarm_on)
16    {
17      if (!waiting && detected == 0)
18      { waiting = 1; off_time = time + WAIT_TIME; }
19      if (waiting && detected == DANGER)
20        waiting = 0;
21      if (waiting && time > off_time)
22      { waiting = 0; alarm_on = 0; *signal_off = 1;}
23    }
24  }
```

Fig. 2. C code for the smoke detector example, line numbers appear in the left column

Of course, states can also exist in system components. This can occur again through the use of the static storage class in C, or in object-oriented languages, through class variables that are protected from external manipulation using access modifiers. In our work however, the focus is only on test objects written using the C language.

The next sections discuss the problems that states in test objects can cause for ET.

3.1 Input Sequences

Partial aims relying on the values of state variables often require input sequences in order for them to be possible. This is because the execution of a series of transitions is required in order to set state variables to desired values. For the smoke detector example, the true branch from line 15 requires the `detected` variable to be equal to the `DANGER` threshold. This requires five calls to the function, each of which must execute the transitional statement on line 10, which increments the `detected` variable. Not until this has been done does the target become feasible.

A problem with the generation of sequences is that it is generally impossible to automatically determine beforehand roughly how long the required sequence is going to be, as different test objects can require radically different sequence lengths.

3.2 Disseminated Functionality

Where system functionality is dispersed across a series of components, it is possible that the function under test is not the function responsible for manipulating the state in the desired way. For these more complicated state problems, input sequences must be found to call different functions in a certain order. This is illustrated by the exam marks example in figure 3. The target is the true branch from the `if` statement in the `compare` function (part (a)). This depends on the state of the module portrayed in part (b). Only until the function `has_max` in this module has been executed a certain number of times will the branch in `compare` become feasible.

3.3 Guidance To Transitional Statements.

A sequence of function calls is not always enough to ensure that transitions will be invoked for test goals to become feasible. Extra guidance must be provided to find inputs that ensure that transitional statements are actually executed, and executed in the correct order. In the exam marks example, the incremental statement involving the `num_top` variable in the dependent module must be executed for the test goal in the compare function to become feasible (assuming `last_year_top` is greater than zero). This statement is unlikely to be executed at random since the actual input value that will cause the statement to be reached (where `mark` is equal to 100) occupies a very small portion of the input domain.

The use of control variables can further complicate matters. Control variables are used to model the fact that a system's behavior should change given the fact that certain events have occurred, as with the flags used in the smoke detector (`alarm_on` and `waiting`). Such variables are often implemented as Boolean variables, to indicate the system is in a state or not in a state; or as enumerations, to indicate the system is in one of a collection of states. ET has further trouble with such variables, for the same reason that it has trouble with the closely related flag problem [3, 6]. The evolutionary search receives little guidance from branch distance calculations using these types of variables, due to their "all or nothing" nature. As state problems are not only dependent on the current input but also the history of inputs, the problem is accentuated.

```
// ...
void compare()
{
  if (get_num_top() >
        last_year_top)
  {/* target branch */}
}
// ...
```

(a) Function under test

```
const int MAX_MARK = 100;
static int num_top = 0;
void has_max(int mark)
{
  if (mark == MAX_MARK)
    num_top ++;
}
int get_num_top ()
{ return num_top; }
```

(b) Dependent module

Fig. 3. C code for the exam marks example

4 Possible Solutions

Solving state problems for ET would be seemingly straightforward if some state machine specification of the test object existed. Unfortunately such a model might not always be available, and even if it were, the required information may not be present. Such representations tend to model control only (due to state explosion problems), whereas partial aims in ET may depend also on data states.

An alternative is to identify transitional statements using data dependency analysis [1]. The chaining approach of Ferguson and Korel [5] is relevant here. The chaining approach was not specifically designed for state problems but rather for the structural test data generation for "problem" program nodes, whose executing inputs could not be found using their local search algorithms (hence many nodes were declared problematic in their work, since these techniques often became stuck in local optima). As a secondary means of trying to change the flow of execution at the problem node, data dependency analysis was used to find previous program nodes that could somehow affect the outcome at the problem node. Through trial and error execution of sequences of these nodes, it was hoped that the outcome at the problem node would be changed.

In figure 1, execution of the `if` statement at node 4 as true may have been declared as problematic. The use of a flag variable produces fitness values of zero or one, which provides little guidance to the search for identifying input values for the true case in the event that current test data had led to the false case, or vice versa. In this situation the chaining approach would look for the last reachable definitions of variables used at the problem node. In this instance the variable `heater_on` is the only variable used at node 4, and its last reachable definitions can be found at nodes 2 and 3. Therefore two node sequences, known as *event sequences*, are generated - one requiring execution of node 2 before node 4, and the other requiring the execution of node 3 before node 4. In order to execute node 2 in the sequence <*s, 2, 4*>, the true branch from node 1 must be taken, requiring the search for input data to execute the condition `temp < TEMP_THRESHOLD` as true. This becomes the new goal of the lo-

cal search. If the required test data to execute this branch is found, node 2 is reached and finally the condition on node 4 is also evaluated as true.

In the case where a node leading to a last definition node also became problematic, the chaining approach procedure would recurse to find the last definition nodes for the variables used at these new problem nodes. This led to the generation of longer and longer sequences until some satisfying sequence was found or some pre-imposed length limit was reached.

The use of the chaining approach to change execution at problem nodes is a useful concept that seems applicable to ET and the state problem. In the chaining approach, problem nodes are those for which input data could not be found with local search methods. With the state problem, difficult nodes are those that are not easily executed with ET, due to the use of internal state variables. The chaining approach found previous nodes that could change the outcome of the target node. For the ET state problem; transitional statements need to be found and executed that manipulate the state, as such guidance is not already provided by the fitness function.

5 Applying the Chaining Approach

Our system (from here on referred to as ET-State) aims to deal with test objects exhibiting state behavior whose structural elements could not be covered using traditional ET. The system identifies potential event sequences. Traditional ET then attempts to find the input sequence that will lead to both execution of the event sequence, and if the identified event sequence leads to the desired state, the structural target also. This is performed using path-oriented fitness functions, so that each transitional statement receives the required guidance that leads to its execution.

Event sequences for state problems will be potentially much longer than those for problem nodes for functions solely dependent on the current input, as identified by the original chaining approach. As will be seen, our system also performs data dependency analysis that is more extensive than the original chaining approach, meaning that the number of nodes available to add to each sequence at each step of its construction is also greater. For complicated examples an exhaustive search of the chaining tree may not always be tractable. We plan to accommodate this by using a further stage of optimization to heuristically build and evaluate sequences – namely the ant colony system.

5.1 Ant Colony System (ACS)

Ant colony system (ACS) [2] is an optimizing technique inspired by the foraging behavior of real ants [7]. In ACS the problem is represented by a graph. Ants then incrementally construct solutions using this graph by applying problem-specific heuristics and by taking into account artificial pheromone deposited by previous ants on graph edges. This informs the ant of the previous performance of edges in previous solutions, since the edges belonging to the better solutions receive more pheromone.

ACS is an attractive search technique to use for building event sequences for the state problem, since the incremental solution construction process allows for straight-

forward incorporation of data dependency procedures for identifying possible transitional program nodes. The space of viable event sequences for a state problem is underpinned by the control and data dependencies of the underlying code structure, and as these factors can be taken into account, ants are prevented from exploring solutions that are unintelligent or infeasible from the outset.

Dorigo *et al.* [4] originally devised ant systems for the traveling salesman problem (TSP). Here, graph nodes correspond to cities in the problem. Initially, graph edges are initialized to some initial pheromone amount τ_0. Then, in t_{max} cycles, m ants are placed on a random city and progressively construct tours through the use of a probabilistic transition rule. In this rule, a random number, q, $0 \leq q \leq 1$ is selected and compared with a tunable exploration parameter q_0. If $q \leq q_0$, the most appealing edge in terms of heuristic desirability and pheromone concentration is always chosen. However if $q > q_0$ a transition probability rule is used. The probability of ant k to go from node i to node j whilst building its tour in cycle t, is given by:

$$p_{ij}^{k}(t) = \frac{[\tau_{ij}(t)] \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in J_i^k} [\tau_{il}(t)] \cdot [\eta_{il}]^{\beta}} \quad \textbf{(1)}$$

where $\tau_{ij}(t)$ is the amount of pheromone on edge (i,j), β is an adjustable parameter that controls the relative importance of pheromone on edges, and η_{ij} is a desirability heuristic value from city i to j. In the TSP, the desirability heuristic is simply the inverse of the distance from node i to j, making nearer cities more attractive than those further away. Ants keep track of towns they have visited, and exclude these cities from future choices.

Every time an edge is selected a local update is performed whereby some of the pheromone on that edge evaporates. This has the effect of making that edge slightly less attractive to other ants, so as to encourage exploration of other edges of the graph. A local update for an edge (i,j) is computed using the formula:

$$\tau_{ij}(t) \leftarrow (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \tau_0 \quad \textbf{(2)}$$

where ρ is the pheromone decay coefficient $0 < \rho \leq 1$.

After a cycle has finished, a global update of the graph takes place on the basis of the best solution found so far (i.e. the shortest tour). This encourages ants to search around the vicinity of the best solution in building future tours. The global update is performed for every edge (i,j) in the best tour using the formula:

$$\tau_{ij}(t) \leftarrow (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \Delta\tau_{ij}(t) \quad \textbf{(3)}$$

where $\Delta\tau_{ij}(t)$ is computed as the inverse of the tour distance.

At the end of the last cycle the algorithm terminates and delivers the shortest tour found.

In the following sections we outline how the basic ACS algorithm is adapted for solving state problems.

Problem Representation. The problem representation for the state problem is simply a graph of all program nodes linked to one another.

However, in the state problem program nodes can be visited more than once in a sequence. This means that arcs belonging to good solutions can also be reinforced more than once. In order to prevent a situation where a sub-path is reinforced to the point where ants begin to travel around the graph in infinite loops, the search space is extrapolated so that nodes in the graph correspond to the *n*th use of some program node in building an event sequence. This expansion of the search space can take place dynamically, so there is no need to pre-impose limits on the number of times each individual program node can be visited.

Solution Construction. At each stage of event sequence construction, each ant chooses from a subset of all program nodes, as identified by the data dependency analysis procedure.

The original chaining approach built event sequences by working backwards from the target node. Data dependency analysis for a problem node ended at the last definitions of variables used at that node. For state problems, we extend the data dependency analysis by considering all nodes that could potentially affect the problem node (for example by analyzing variables used in assignments at last definitions, and so on). The algorithms for doing this are similar to those used to construct backward program slices [8].

Evaluation of Event Sequences. In the ET-State ant algorithm, the heuristic used to evaluate the desirability of the inclusion of a node into the event sequence is also equivalent to the "goodness" of the entire event sequence including that node. In order to evaluate event sequences, they are first executed by traditional ET using path-oriented fitness functions. The fitness information from the best individual is then fed into ET-State. In most cases, the best individual found would likely have a series of diverge points corresponding to nodes depending on states.

When adding a new node to an event sequence, ants use inputs used from the previous sequence to seed the first generation of the evolutionary algorithm for finding inputs leading to the execution of the new sequence.

Pheromone Updates. Local and global pheromone updates are handled in a similar fashion to ACS for the TSP, however $\Delta\tau_{ij}(t)$ for global updates is computed using the fitness of the event sequence as evaluated with ET.

Termination Criteria. As the length of solutions for the state problem is not fixed as they are for tours in the TSP, termination criteria are required to stop ants building infinitely long sequences. Two rules include a) the ant stops if it discovers a solution better than the current best solution found so far, and b) ants can only explore sequences up to a certain number of nodes longer than the current best so far, or the ant fails to improve its solution over the last *n* nodes.

5.2 Simple Initial Experiment

A simple initial experiment was run with a preliminary version of the system. The code used for this study can be seen in figure 4. The aim is for the ants to find an event sequence to get into the true branch from the `if` statement in `fn_test` (part (a)). The outcome of this branch predicate is dependent on functions in the module shown in part (b), which depend on internal state variables.

Four ants were used in ten cycles, the desirability heuristic exponent β was set to 2, the exploration parameter q_0 was set to 0.75, and a pheromone decay ρ of 0.25 was used. Sequences were built using backward dependency analysis. Therefore when adding the first node to its sequence, the ant could only choose to execute `fn_c`. For the second node the ant could execute `fn_c` or `fn_b`, since the assignment statement for `k` in `fn_c` uses the variable `j` that in turn is defined in `fn_b`. For every node after `fn_b` was called, any function from the dependent module could be used, since `fn_b` uses the variable `i` which in turn is defined in `fn_a`.

In the first cycle ants were prohibited from using any node more than once, so that the full definition-use chain from `k` through to `i` could potentially be explored. The greedy desirability heuristic for the addition of nodes, and for evaluating entire sequences, was simply based on the branch distance for the true branch in `fn_test`. In this case, and in order to make the search more complicated, the ants were directed to find the shortest satisfying sequence once a satisfying sequence had been found, by simply rewarding shorter sequences (of course, shorter sequences ultimately require less mental effort on behalf of the human checking the results of the structural tests).

Ants finished building their sequence if they had found a new best, or they had made no improvement on their sequence for the last five nodes. The results showed promise, as seen in table 1. In ten runs of the experiment, ants found a satisfying sequence in 2-3 cycles. The shortest function call sequence found (<`fn_a`, `fn_a`, `fn_b`, `fn_c`, `fn_c`, `fn_c`, `fn_test`> (or similar sequence of the same length)) was ascertained in an average of 4.9 cycles.

```
// ...
const int target = 32;

void fn_test()
{
  if (fn_c() == target)
  { /* target branch  */ }
}
// ...
```

(a) Function under test

```
static int i=0, j=0, k=0;

void fn_a()
{ i ++; }

void fn_b()
{ j += i * 2; }

int fn_c()
{ k += j * 2; return k; }
```

(b) Dependent module

Fig. 4. C code for the initial experiment

Table 1. Results for the initial experiment

	Average	Best	Worst
Satisfying Event Sequence Found	2.1	2	3
Shortest Satisfying Event Sequence Found	4.9	3	7

No of ants: 4 No of trials: 10

6 Conclusions and Future Work

This paper has introduced the state problem for evolutionary testing; showing that state variables can hinder or render impossible the search for test data. State variables can be dependent on the input history to the test object, as well as just the current input. They do not form part of the input domain to the test object, and therefore cannot be optimized by the ET search process. The only way to control these variables is to attempt to execute the statements that perform assignments to them, statements that form the transitions of the underlying state machine of the system. Such statements can be identified by data dependency analysis. The use of ant colony algorithms is proposed as a means of heuristic search and evaluation of sequences of transitional statements, in order to find one that makes the current test goal possible.

The next step in this work is the complete implementation of the system, which will then be tried on a set of industrial examples. These experiments will be considered successful if coverage of structural elements involving states are achieved which were previously unobtainable by the use of ET or even random testing alone. If not all structural elements involving states can be covered then further analysis will take place to understand the features of these programs that are still causing problems.

Acknowledgements. This work is sponsored by DaimlerChrysler Research and Technology. The authors have benefited greatly from the discussion of this work with Joachim Wegener, André Baresel and other members of the ET group at DaimlerChrysler, along with members of the EPSRC-funded FORTEST and SEMINAL networks.

References

1. Aho A., Sethi R., Ullman J. D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)
2. Bonabeau E., Dorigo M., Theraulaz G.: Swarm Intelligence. Oxford University Press (1999)
3. Bottaci, L.: Instrumenting Programs with Flag Variables for Test Data Search by Genetic Algorithm, Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA (2002)
4. Dorigo M., Maniezzo, V., Colorni A.: Ant System: An Autocatalytic Optimizing Process. Technical report, Politechnico di Milano, Italy, No. 91-016 (1991)

5. Ferguson R., Korel B.: The Chaining Approach for Software Test Data Generation. ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 1, pp. 63-86 (1996)
6. Harman M., Hu L., Hierons R., Baresel A., Sthamer H: Improving Evolutionary Testing by Flag Removal. Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA (2002)
7. Goss S., Aron S., Denenubourg J. L., Pasteels J. M.: Self Organized Shortcuts in the Argentine Ant. Naturwissenschaften, Vol. 76, pp. 579-581 (1989)
8. Tip F., A Survey of Program Slicing Techniques. Journal of Programming Languages, Vol.3, No.3, pp.121-189 (1995)
9. Tracey N., Clark J., Mander K.: Automated Flaw Finding using Simulated Annealing. International Symposium on Software Testing and Analysis, pp. 73-81 (1998).
10. Wegener J., Baresel A. Sthamer H.: Evolutionary Test Environment for Automatic Structural Testing. Information and Software Technology, Vol. 43, pp. 841-854 (2001)
11. Wegener J., Buhr K., Pohlheim H.: Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing. Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA (2002)
12. Wegener J., Grochtmann M.: Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. Real-Time Systems, Vol. 15, pp. 275-298 (1998)