Heuristic Search with Reachability Tests for Automated Generation of Test Programs

Wee Kheng Leow, Siau Cheng Khoo, Tiong Hoe Loh, and Vivy Suhendra
Dept. of Computer Science, National University of Singapore
leowwk, khoosc, lohtiong, vivysuhe@comp.nus.edu.sg

Abstract

Our research complements the current research on automated specification-based testing by proposing a scheme that combines the setup process, test execution, and test validation into a single test program for testing the behavior of object-oriented classes. The test program can be generated automatically given the desired test cases and closed algebraic specifications of the classes. The core of the test program generator is a partial-order planner which plans the sequence of instructions required in the test program. A first-cut implementation of the planner has been presented in [4] based on simple depth-first search. This paper presents a more efficient and effective heuristic search algorithm that performs reachability tests using the Omega Calculator. Test results show that heuristic search with reachability tests significantly reduce the search time required to generate a valid sequence of instructions.

1. Introduction

Specification-based testing involves three stages [8]: (1) test case generation, (2) text case execution, and (3) test result evaluation. The first stage generates test cases from a software system's specification. Before the system can be tested, it must be properly set up, i.e., prepare the input variables and data used in the tests according to the requirements stated in the test cases. This setup process is usually performed manually, especially when testing complex data structures. After the system is properly set up, a test execution tool runs the system according to the test cases and pre-recorded test scripts to obtain the outputs, which are checked by a test evaluation tool.

Test execution and test result evaluation are easy to automate, and tools for these stages are already available. Most research on automated specification-based testing has focused on the automated generation of test cases [2, 3, 5, 6, 8, 10]. There is not much research on automated generation of test programs that combine system setup, test execution, and test validation into a single framework, except for the well-known ADL (Assertion Definition Language) system [1]. However, ADL requires the additional programming effort from the user to supply *auxiliary functions* that define the semantics of the function to be tested, *provide functions* for constructing the required test data, and *relinquish functions* for releasing the test data.

Our research complements the current trend by proposing a scheme that combines the setup process, test execution, and test validation into a *single* test program for testing the behavior of object-oriented classes. The test program can be generated automatically given the desired test cases and *closed algebraic specifications* of the object classes [4]. After compiling and linking with the object classes under test, it can be executed to perform test case setup, test execution by invoking the class methods, and test results verification, all in a single program. This scheme provides great convenience in automated specification-based testing by removing the need to perform manual system setup and invoking separate tools for test execution and test evaluation.

The core of the test program generator is a partial-order planner which plans the sequence of instructions required in the test program. A first-cut implementation of the planner has been presented in [4]. In this paper, we presents a more efficient and effective heuristic search algorithm for finding a valid plan. It performs *reachability tests* on objects using Omega Calculator library [7, 9] to determine whether the application of a method or methods can bring the objects to the desired states. Experiment results (Section 3) show that the heuristic search with reachability tests significantly improves the search efficiency and effectiveness of the algorithm.

2. REBID Planner

The detailed algorithm of REBID is given in [4]. The heuristics used in the insertion of new plans into the search queue are:

- Reachable plans are inserted at the front of the queue, and sorted in decreasing number of state labels in the test case that the plans affect. An *affected* state label

is a state label in the test case whose state is changed by the application of a method. In other words, a plan is inserted nearer to the front of the queue if its most recently included instruction affects more state labels.

- Unreachable plans are inserted at the front of the queue behind the reachable plans, and ordered in the same manner as reachable plans.

Note that reachability test is performed for a single method applying on a single target object. A plan is reachable if the application of the method on the target object satisfies the constraints on the object, which can be just a subset of all the constraints in the test case. Unreachable plans are not discarded immediately because, in some cases, a desired goal state is not reachable by applying only one method. They are retained in the queue but given a lower priority for further expansion. Therefore, the heuristic search is guaranteed to find a valid plan and terminate.

2.1. Reachability Tests

Reachability tests determine whether the the preconditions of a method are satisfied, and whether the postconditions of a method invocation imply the constraints on an object. If the constraints are satisfied, then the goal state of the object (represented as constraints) is reachable after applying the method on the object in the test program.

Reachability tests are performed on one object at a time. Single application of a method on an object involves a *one-step* reachability test. Multiple, repeated application of a method on an object involves a *multi-step* reachability test.

One-Step Reachability Tests

A one-step reachability test can be formulated as a *constraint satisfaction* problem:

$$\{[a_1, \ldots, a_m, s_1, \ldots, s_n] : P \wedge Q \wedge C\} \qquad (1)$$

where a_i are the method's arguments, s_j are the object's state labels (including both pre-state and post-state), P is the pre-condition of the method to be invoked, Q is the post-condition, and C links P and Q to the existing test state.

Consider the following (partial) specification of three classes: `Teacher`, `Student` and `Course`:[1]

```
class Course {
   Course()
   { true
     --> #max = 1 && #size = 0}
   Course(int max)
   { max > 0
     --> #max = max && #size = 0}
   void setMax(int max)
   { max >= #size
     --> #max = max && #max >= #size}
   void incMax()
   { true
     --> #max = @#max + 1}
   void decMax()
   { #max > #size
     --> #max = @#max - 1}
   void setTeacher(Teacher t)
   { t != null
     --> #teacher = t}
   void addStudent(Student s)
   { s != null && #size < #max
     --> #size = @#size + 1 &&
         exist(#s in Course){#s = s}}
   void deleteStudent(Student s)
   { #size > 0 &&
     exist(#s in Course){#s = s}
     --> #size = @#size - 1 &&
         !exist(#s in Course){#s = s}}
   // Other access/constructor methods omitted.
}
class Teacher {
   Teacher(String name, int id)
   { name != null && id > 0
     --> #name = name && #id = id}
   // Access methods omitted.
}
class Student {
   Student(String name, int id)
   { name != null && id > 0
     --> #name = name && #id = id}
   // Access methods omitted.
}
```

Suppose the pre-state of a `course` object is `#max = 1 && #size = 0`. If we want to see if `#max` can be set to `5` via the method `setMax(a)` in one step, we can write:

```
{[a, max, size, newmax] :
 a >= size &&                          # pre-cond
 newmax = a && newmax >= size &&       # post-cond
 max = 1 && size = 0 && newmax = 5}  # test states
```

In this example, `#max`, the affected state label, is represented by its pre- and post-state labels. In REBID, a *binding table* is maintained to record the values of bound state labels and method arguments. This can be used to simply the above problem to:

```
{[a] : a >= 0 && 5 = a && 5 >= 0}
```

Evaluation of this expression in Omega Calculator yields `a = 5`. This means the constraints can be satisfied by binding `a` to the value 5. The bound labels and their values are recorded in REBID's binding table.

Multi-Step Reachability Tests

Multi-step reachability test is performed using Omega Calculator's `reachable` function. For example, suppose the pre-state of a `course` object is `#max = 10 && #size = 0`, and we want to know if multiple invocations of `addStudent(s)` can change the object state to `#max = 10 && #size = 5`. In this case, REBID forms a start state named `t` that contains `size` – the pre-state of

1 Here, preconditions are specified before the arrow symbol '`-->`' while postconditions are specified after '`-->`'. Symbols prefixed with '`#`' such as `#name` and `#size` refer to *state labels*. Symbols prefixed with '`@`' refer to the *pre-states* of the objects. Kindly refer to [4] for a detail description of the symbols.

#size. In Omega Calculator, the above multi-step reachability test can be written as

```
R := reachable of t in (t) {
  t : {[size, max] : size = 0 && max = 10} |
  t -> t: {[size, max] -> [newsize, max] :
    exists([s] : s != null && size < max &&
           newsize = size + 1 &&
           exists([s1] : s1 = s))
}};
```

The second line defines the *start state* of t. The third line indicates the *state transition* that changes the state. The exists clause specifies the constraints on the state transition, which has the same syntax as Equation 1.[2]

The above test will be simplified by REBID using its binding table. Evaluation of the simplified expression in Omega Calculator yields

```
R := {[size] : 1 <= size <= 10}
```

which gives a set of possible solutions to t. To verify whether the solutions satisfy the test case requirement, a non-empty test is performed on the intersection of the returned set and the test case requirement (i.e., #size = 5):

```
R intersection {[5]};
```

which yields a non-empty answer size = 5.

3. Experiments and Discussions

Five variations of the REBID planner were tested:[3]

- BFS: Breadth-first search without heuristics.
- DFS: Depth-first search without heuristics.
- 0RT: Heuristic search without reachability test (RT). The plans are ordered by the number of affected state labels without checking whether they are reachable.
- 1RT: Heuristic search with one-step RT only.
- MRT: Heuristic search with one-step and multi-step RT. One-step RT is performed first. Multi-step RT is performed only if one-step RT fails.

Three test cases were performed based on the example specifications given in Section 2.1. These specifications were chosen because they were simple and straightforward, and yet rich enough to illustrate various important aspects of the REBID algorithm.

3.1. Test Case 1

Test Case 1 assessed the performance of the algorithms in constructing a simple object that contained another object as its attribute:

```
Course c1: c1.#max=10, c1.teacher = t1
Teacher t1: t1.name = "Ms Lee"
```

The test performance is as follows:

	BFS	DFS	0RT	1RT	MRT
run time (s)	0.51	> 120	0.01	0.17	0.19
depth	3	> 106	3	3	3
no. of plans	30	> 373	5	5	5

The search algorithms with heuristics were most efficient and they found valid plans by searching only a tree of depth 3 containing 5 plans. MRT took a little longer than 0RT and 1RT because it invoked the reachable function of Omega Calculator, which took a little more time to solve compared to simple constraint satisfaction. BFS could also find a valid plan after searching through 30 plans up to a depth of 3. Therefore, it took longer to find a valid plan.

DFS could not find a valid plan after executing for 2 minutes and was aborted. This happened because DFS chose the constructor to construct a Course object with #max = 1 first. Then, it picked decMax and incMax alternately to modify the value of #max, entering an infinite loop.

DFS is very sensitive to the sequence of method specifications. If we swap the sequence of incMax and decMax in the specification so that DFS always tries incMax *before* decMax, then DFS can also find a valid plan. Nevertheless, the heuristics algorithms significantly shorten the execution time by directing the search along paths that are more likely to succeed.

3.2. Test Case 2

Test Case 2 measured the algorithms' performance in constructing an aggregate object that contained 2 elements.

```
Course c1: c1.#max=10, c1.#size=2
```

For Test Case 2, DFS again could not find a valid plan, as it was trying alternate invocations of addStudent and deleteStudent:

	BFS	DFS	0RT	1RT	MRT
run time (s)	4.87	> 120	> 120	0.54	0.62
depth	6	> 106	> 106	5	5
no. of plans	199	> 373	> 373	9	9

0RT also suffered the same fate as DFS. On the other hand, 1RT and MRT could obtain valid plans, and they executed more efficiently than BFS did.

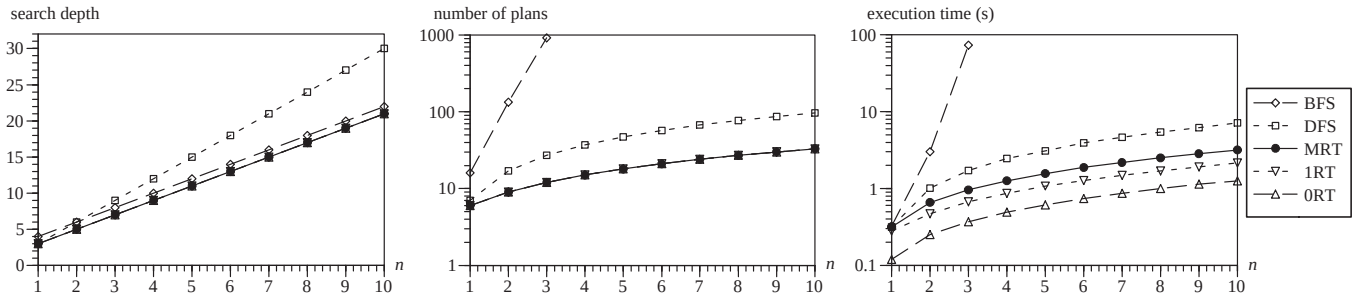
3.3. Test Case 3

Test Case 3 was similar to Test Case 2 except that the algorithms were to construct aggregate objects with maximum number of elements, and the maximum number n varied from 1 to 10:

```
Course c1: c1.#max=n, c1.#size=n
```

2 The symbol null will be substituted with a known constant.

3 The performance of the algorithms were measured in terms of (1) execution time (in a Pentium 1.6GHz PC with 256MB RAM), (2) depth of search tree, and (3) total number of plans generated, which reflects the space requirement.



(a) Search depth (b) Number of plans generated (c) search time (sec)

Figure 1. Performance of search algorithms with increasing n.

For Test Case 3, when the original method sequence was used, DFS and 0RT could not generate a valid plan for $n > 1$, 1RT failed for $n > 2$, and BFS failed for $n > 3$, each after running for 2 minutes. DFS, 0RT, and 1RT were stuck for the same reason discussed in Sections 3.1 and 3.2. On the other hand, BFS tried to search through all plans to find the valid one, and it could not complete the search in 2 minutes when $n > 3$.

As for Test Cases 1 and 2, after changing the method sequences in the specification, DFS, 0RT, and 1RT could all generate valid plans. Figure 1 illustrates the algorithms' performance.[4] The heuristic search algorithms were more efficient than DFS. MRT and 1RT were a little slower than 0RT due to the invocation of reachability tests. MRT invoked multi-step reachability tests using Omega Calculator's `reachable` function, which took a little more time than the one-step reachability tests invoked by 1RT. BFS tried to search all possible plans for a valid plan. Its execution time and space requirement (number of plans generated) increased exponentially with n.

4. Conclusion

This paper presented a method of improving the search efficiency and effectiveness of REBID for automated generation of test programs. Using heuristic search with multi-step reachability tests, it can find a correct plan (i.e., instruction sequence) more efficiently than BFS and DFS. Moreover, it can always find a valid plan regardless of the sequence of methods in the class specifications because it can direct the search along paths that are more likely to yield a valid plan. On other the hand, heuristic search with one-step reachability tests and no reachability tests may not be able to find a valid plan for moderately complex test cases.

Further enhancements can be made in the following ways. The current reachability tests return only true or false values. They can be enhanced to return *likelihood of success* based on a measure of the *distance* between the state of the current plan and the goal state. Also, *partial* reachability tests can be performed. That is, if a subset of constraints can be satisfied by the invocation of a method, then partial reachability is obtained. This is especially useful when application of several different modifier methods is required to bring an object to the desired state.

4 In Figure 1(a), BFS's search depths for $n > 3$ are projected values based on its search depths for $n \leq 3$.

References

[1] Assertion Definition Language, adl.opengroup.org.

[2] M. Donat. Automating formal specification based testing. In *Proc. Conf. on Theory and Practice of Software Development*, volume 1214, pages 833–847, 1997.

[3] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic test generation from statecharts using model checking. Technical Report MS-CIS-01-07, Dept. of Computer and Information Science, U. of Pennsylvania, 2001.

[4] W. K. Leow, S. C. Khoo, and Y. Sun. Automated generation of test programs from closed specifications of classes and test cases. In *Proc. ICSE*, 2004.

[5] A. M. Memon, M. E. Pollack, and M. L. Soffa. Using a goal-driven approach to generate test cases for guis. In *Int. Conf. Software Engineering*, 1999.

[6] A. J. Offutt and S. Liu. Generating test data from SOFL specifications. *J. of Systems and Software*, 49(1):49–62, 1999.

[7] The Omega Project. www.cs.umd.edu/projects/omega.

[8] R. M. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society Press, 1996.

[9] W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Comm. of ACM*, 8:102–114, 1992.

[10] M. Scheetz, A. von Mayrhauser, R. France, E. Dahlman, and A. E. Howe. Generating test cases from an OO model with an ai planning system. In *Proc. 10th Int. Symp. on Software Reliability Engineering*, 1999.