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

Most research on automated specification-based software testing has focused on the automated generation of test cases. Before a software system can be tested, it must be set up according to the input requirements of 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 to obtain the outputs, which are evaluated by a test evaluation tool.

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 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 [9] 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

Testing is a very important but expensive and time-consuming process in software development. It can consume at least 50% of the total costs involved in developing software [1]. It remains as the primary method for discovering faults in software systems even though there is steady advancement in formal methods for program verification. Automation of the testing process could reduce development costs and improve software quality.

Specification-based testing involves three stages [14]: (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 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. There is also a lot of research on automated specification-based software testing focusing on the automated selection or generation of test cases [14]. 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 (Section 3). 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 [9] based on simple depth-first search, which can spend a lot of effort searching along the wrong paths. 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 [13, 15] to determine whether the application of a method or methods can bring the objects to the desired states. Test results (Section 5) show that the heuristic search with reachability tests significantly reduce the search time required to generate a valid sequence of instructions.

2. Background and Related Work

Most research on automated specification-based testing has focused on the automated generation of test cases [6, 14, 18]. For example, Donat developed a technique for generating test cases from specifications that contain quantifications [4]. Offutt and Liu presented a method for generating test cases from specifications written in SOFL, which is a kind of formal specification language [12]. Memon et al. developed a method based on AI planner to generate test cases for testing GUI [10]. Scheetz et al. also applied AI planner to generate test cases from test objectives derived from UML models [18]. Graves et al. conducted empirical study to compare the cost and benefit of several techniques for selecting subsets of test cases for regression testing [5]. Other recent work has focused on automated testing of specific software properties such as safety violation in telephone switching systems [8] instead of general software testing. Chan et al. [2] classified the various integration testing techniques for object-oriented programs into state-based, event-based, fault-based, testing against formal specification (aka. algebraic specification and contract specification [3]), and deterministic and reachability techniques.

In comparison, 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 [17] and its successor, ADL2 [11].

ADL provides a framework for specifying the semantics of a software component such as a function or a module. Given an ADL specification, the ADL Translator can automatically generate a test program that executes the function or module under test and checks the test results. To support the automated generation of test programs, ADL requires the user to supply *auxiliary functions* that define the semantics of the function to be tested. We call this type of specification system an *opened specification system*. In addition, the user also needs to provide implementations of the *provide functions* for constructing the required test data and the *relinquish functions* for destroying the test data.

The strength of an opened specification system is that it can be used to specify a single function or to partially specify a module, and test program can be generated to test the function or partially specified module. However, an opened specification system also has the following shortcomings:

- An opened specification is incomplete—it does not contain enough information for generating test data by itself. In testing complex software components, the user cannot avoid the need to provide supporting functions such as ADL's *auxiliary*, *provide*, and *relinquish functions*. Additional programming effort is required to implement these supporting functions, which may not have any use other than for testing. Consequently, test programs cannot be generated from the specification alone, and test program generation cannot be fully automated.
- Supporting functions for testing complex modules may be quite complex themselves and should be subjected to testing also. Although testing of supporting functions can be accomplished by specifying them in ADL, such a requirement is not enforced by ADL. Moreover, testing of these supporting functions may, in turn, require other supporting functions.

Our research complements the current research on automated specification-based software testing in two ways: (1) It proposes a *closed specification system* (Section 3) that can overcome the above shortcomings of opened specification systems. (2) It proposes a scheme that combines automated test data generation (i.e., system setup), test execution, and test validation into a *single* test program. The test program is generated automatically given the class specifications and the test cases. When it is executed, it will perform system setup and test data generation, test execution, and test validation automatically.

To fulfill these goals, the specification must be defined for an entire object class instead of a single function. The semantics of the class methods are specified in terms of other methods which are, in turn, specified in their own class specifications. In other words, all the methods used in a class specification are defined in the same specification or in other class specifications, and the methods can be defined mutually recursively. So, a closed specification is a form of algebraic specification that emphasizes the completeness of semantic information within the specification. The target programming language is Java because it is practically useful and is simpler to handle than is C++.

Fulfilling the above requirement of the closed specification system may, at first glance, appear to be a daunting task for a software that involves many classes. More careful thought, however, reveals that the effort required is really not much more than providing the *auxiliary*, *provide*, and *relinquish* functions for ADL. Once a specification has been defined for a class, it can be readily reused in the specifications of many other classes. On the other hand, the supporting functions developed for testing a particular function or module are less readily reusable for testing other functions or modules. Therefore, in the long run, it is more beneficial to use a closed system than an opened system.

With closed specifications, every class method is defined in terms of other methods which are, in turn, defined in their class specifications. The core of our test program generator is an AI planner that plans the sequence of instructions required in the test program (Section 4.2). The AI planner is an appropriate tool since it is able to sequence the instructions, taking into account the constraints between them [16]. Moreover, the *partial-order planner* can plan a sequence of instructions that are only partially ordered but not totally ordered [16]. As discussed above, AI planner has also been used to generate test cases from specifications [10, 18]. So, it is a very useful tool for automated software testing.

Our planner is implemented as a heuristic search algorithm (Section 4.3). It makes function calls to the Omega Calculator library [13, 15], which solves the constraints given by the test cases and obtains valid variable instances. Furthermore, it uses the Omega Calculator to perform reachability tests on object states (Section 4.4). This method greatly improves the search efficiency and effectiveness of the algorithm.

3. Closed Specifications of Classes

In our system, the behavior of the classes are specified using an ADL-like specification language. Other specification languages can also be used, but we find the ADL syntax more similar to Java, our target programming language for software development. So, we expect Java developers to adopt the ADL syntax more readily than other syntax. The following example shows the specifications of three classes: `Teacher` and `Student`, which are atomic classes, and `Course`, which is an aggregate class.

```
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}
   }

   int max()
   { true
     --> max() = #max
   }

   int size()
   { true
     --> size() = #size
   }
   // Other access 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.
}
```

In this specification, preconditions are specified before the arrow symbol ‘`-->`’ while postconditions are specified after ‘`-->`’. Symbols prefixed with ‘#’ such as `#name` and `#size` refer to *state labels*. They specify the information that is contained in a class without saying how the information is organized and stored in the class. Symbols prefixed with ‘`@`’ refer to the *pre-states* of the objects. For instance, `@#size` refers to the value of `#size` at the entry of the `add` method. Therefore, `@#size` has the same value as the `#size` in the precondition, and the `#size` in the postcondition is equal to `@#size+1`. A method argument must either be bound to a state label (e.g., `name` in constructor `Student`) or appear in the pre- or postcondition. Otherwise, it does not carry any useful information and can be discarded. For the access methods, the postconditions are very simple: the invocation of an access method equals some state labels of the respective class. Note that the semantics of all the methods in the classes are completely specified within them. That is, the specification is closed.

4. Automated Generation of Test Programs

4.1. Overview

A test program that exercises a class method according to a test case consists of three steps: (1) constructs target object and method arguments that satisfy the conditions in the test case, (2) applies the method on the object with the method arguments, (3) checks whether the actual results tally with the expected results given in the test case.

Let us discuss the main ideas using an example. Consider the following test case (specified in a format consistent with IEEE Standard 829 [7]):

Input specification:

```
Course course1 & course1.#max = 10 &
course1.#size = 1 &
Student student1
```

Method Invocation:

```
course1.add(student1)
```

Output specification:

```
Course course1 & course1.#max = 10 &
course1.#size = 2
```

Then, a test program for this test case could be:

```
// Step 1: Object construction
Course course1 = new Course(10);
Student student2 =
   new Student("Mike", 12345);
course1.addStudent(student2);

Student student1 =
   new Student("Mary", 23456);

// Step 2: Method invocation
course1.addStudent(student1);

// Step 3: Test result verification
if (course1.max() == 10 &&
    course1.size() == 2)
then System.out.print("Passed");
else System.out.print("Failed");
```

The first three instructions construct a `Course` object that satisfies the input requirement of the test case. The fourth instruction constructs a `Student` object required for the test. The fifth instruction invokes the method under test, and the remaining instructions perform test result verification. Testing of exception handling can be done in a similar manner by modifying step 2 to catch a possible exception, and then checking whether the postcondition for exception handling is satisfied.

This example shows that it is straightforward to generate program codes for steps 2 and 3, especially when the test case specification is written in computer readable format. However, automated generation of program codes for step 1 is non-trivial:

- The method arguments of the object constructor of the target object may be objects as well, and they are required to satisfy the conditions given in the test case. Therefore, the object construction algorithm must be applied recursively to construct the method arguments.
- The object constructor may not be able to create an object that meets the test case conditions (as illustrated in the above example). Additional modifier methods (e.g., `addStudent`) may need to be invoked to bring the object to the required state.

Therefore, in the remainder of this paper, we will focus on the automated generation of object construction codes.

Object construction codes consist of three parts: (1) *argument creation*: create arguments $u_1, \ldots, u_n$ of the target constructor C; (2) *object creation*: create the target object x; and (3) *object modification*: modify the state of x by applying modifier methods $M_1, \ldots, M_m$. For example,

$C_1\ u_1 = \texttt{new}\ C_1(\ldots)$; // part 1
$\ldots$
$C_n\ u_n = \texttt{new}\ C_n(\ldots)$;

$C\ x = \texttt{new}\ C(u_1, \ldots, u_n)$; // part 2

$x.M_1(\ldots)$; // part 3
$\ldots$
$x.M_m(\ldots)$;

Because an argument can also be an object, the codes for creating an argument may also involve three parts, just like object construction codes. Therefore, *recursive planning* is needed to correctly generate the program codes.

4.2. REBID Planner

A *recursive bidirectional* planner called REBID for generating object construction codes has been developed and presented in [9]. REBID starts the planning process by generating object creation code (part 2). This is a good strategy because a class typically has far fewer constructors than modifier methods. Many classes may even have only one constructor. REBID works *backward* to generate the codes for constructing the arguments $u_1, \ldots, u_n$ *recursively* because the construction of the arguments may also involve 3-part codes. At the same time, REBID also works *forward*, if necessary, to generate the codes to bring the the target object to the required state.

Details of REBID is described in [9] in the form of a *nondeterministic* planner algorithm. Here, we give a brief summary of REBID and its deterministic implementation using heuristic search with reachability tests. This search strategy significantly improves the search efficiency and effectiveness of REBID compared to the simple method of depth-first search without heuristics described in [9].

In REBID, the input conditions specified in a test case, as well as the preconditions and postconditions of methods, are regarded as constraints on the object's state labels and method arguments. REBID invokes the Omega Calculator to perform constraint solving to determine whether the conditions can be satisfied.

Nondeterministic REBID

MakePlan

1. Create initial plan with initial instruction for the target object and the object's constraints R. The method for constructing the object is not yet determined.
2. Repeat
   (a) If all the methods of all the instructions have been determined, and all constraints in R have been satisfied, then instantiate the unbound state labels and method arguments, and return the plan.
   (b) Else, CreateObject or ModifyObject.

CreateObject

1. **Choose** an instruction I in the plan whose method is not yet determined.
2. **Choose** an appropriate constructor method that can satisfy at least some of the constraints in R.
3. Set the method and arguments of instruction I.
4. Record the method's preconditions and postconditions as the constraints involved in executing I.
5. Record variable bindings.
6. ExpandArguments of instruction I.

ModifyObject

1. **Choose** the last instruction I that modifies an object such that execution of I still cannot fully satisfy the object's constraints.
2. **Choose** an appropriate modifier method that can satisfy at least some of the constraints.
3. Create a new instruction J with the chosen method and arguments.
4. Include instruction J into the plan *after* instruction I.
5. Record the method's preconditions and postconditions as the constraints involved in executing J.
6. Record bindings of state labels and method arguments.
7. ExpandArguments of instruction J.

ExpandArguments(I)

For each argument u of instruction I that is not a string literal and not a constant of primitive data type:

1. **Choose** an unbound state label from the constraints in R that can be bound to u.
2. Create instruction J for argument u and the constraints on u. The method for constructing argument u is not yet determined.
3. Insert instruction J into the plan *before* instruction I.
4. Record bindings of state labels and method arguments.

The nondeterministic **Choose** selects five types of candidates: instructions, class methods, method arguments, unbound state labels, and constraints. There are finite and enumerable numbers of instructions, class methods, state labels, and constraints. So, they can be found with a search algorithm. Selection of method arguments is more complicated because there is potentially an infinite number of possible values and they may need to satisfy some methods' preconditions. If the value of an argument is given in the test case (which is assumed to satisfy the preconditions), then it can be assigned the value. Otherwise, the preconditions have to be recorded as constraints and the values can only be determined at the end of the planning process (Step 2(a) of **MakePlan**) by proper instantiation of the arguments.

4.3. Heuristic Search

Below is a summary of the deterministic implementation of REBID.

Deterministic REBID

1. Make initial plan and insert it into a search queue.
2. Repeat
   (a) Remove the plan at the front of the queue.
   (b) Invoke Omega Calculator to perform instantiation and binding of each state label and method argument to an appropriate value that satisfies their constraints. These constraints may come from the test case or the preconditions or postconditions of the methods in the plan.
   (c) If binding fails, continue with next iteration.
   (d) If binding succeeds and the plan satisfies all the constraints in R, then terminate with the plan.
   (e) Construct new plans by choosing either constructor or modifier methods. Each new plan is constructed with one new method selected.
   (f) Invoke Omega Calculator to perform reachability tests on the chosen methods and the corresponding objects.
   (g) Insert new plans into search queue based on heuristics.

The heuristics used in the insertion of new plans 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 at Step 2(d).[1]

The ordering of the plans by the number of affected state labels tells the planner to first try to expand a plan whose most recently included method affects more state labels in the test case, i.e., trying to satisfy more constraints. If the method is found to satisfy many constraints, then there is a very good chance that it is the correct method to invoke. Furthermore, after satisfying many constraints, the remaining planning problem would be easier to solve because there are fewer remaining constraints to satisfy. On the other hand, a method that affects very few state labels in the test case might be a wrong method because it is relatively easy for many methods to satisfy few (e.g., one) state labels. If a wrong method is chosen, then expanding the plan would not lead to a valid final plan, and a lot of search effort would be wasted in expanding the plan.

Different search strategies can be obtained by changing the method of inserting new plans into the queue. For example, if new plans are always inserted at the front of the queue without heuristics, then the algorithm reduces to depth-first search. On the other hand, if new plans are always inserted at the back of the queue without heuristics, then the algorithm reduces to breadth-first search.

4.4. Reachability Tests

Reachability tests play a crucial role in the deterministic implementation of REBID. They 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. Sequential applications of different methods on the same object is not considered in reachability tests because determining the correct sequence of method invocation is the task of the REBID planner.

1 It is assumed that a valid plan exists for a test case. Verifying whether a test case is viable and testable is beyond the scope of this paper.

One-Step Reachability Tests

First, let us describe one-step reachability test. Single application of a method changes the state of an object from the pre-state to the post-state. In order that the method can be invoked successfully, the pre-state has to satisfy the method's precondition P. The successful invocation of the method changes the object's state in a manner specified in the method's postcondition Q. So, the post-state is given by the postcondition Q. So, the one-step reachability test can be formulated as a *constraint satisfaction* problem:

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

where a_i are the method's arguments, s_j are the object's state labels, and C represents the constraints.

For example, suppose the pre-state of a `course` object is `#max = 1 && #size = 0`. If we want to know whether `setMax(a)` can change the state of `course` object to `#max = 5 && #size = 0`, then we can formulate the following constraint satisfaction problem:

```
{[a, max, size, newmax] :
 a >= size &&
 newmax = a && newmax >= size &&
 max = 1 && size = 0 && newmax = 5}
```

In this example, `#max` is an affected state label. So, two versions of the state label is automatically created by REBID to represent the state label at the pre-state and the post-state. The first line lists the set of method arguments and state labels. The second line comes from the precondition of `setMax(a)`, and the third from the postcondition. The last line comes from the pre-state and the desired post-state.

In REBID, a binding table is maintained to record the values of bound state labels and method arguments. So, REBID can simplify the above constraint satisfaction problem by (1) including only unbound state labels and method arguments in the label set, and (2) replacing the bound state labels and method arguments by their bound values. This method yields the following simplified constraint satisfaction problem:

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

which requires less work on Omega Calculator.

In Omega Calculator syntax, the above constraint satisfaction problem is written as

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

Evaluation of this expression in Omega Calculator yields

```
{[5]}
```

which is a set that contains the possible value. This means the constraints can be satisfied, and in addition, `a` can be bound to the value 5. If the constraints cannot be satisfied, then Omega Calculator will return `False`. So, Omega Calculator performs not only constraint satisfaction but also instantiation and binding of values to the state labels and method arguments. The bound labels and their values are recorded in REBID's binding table. Essentially the same procedure is performed in Step 2(b) of the deterministic REBID for instantiation and binding.

Multi-Step Reachability Tests

Multi-step reachability test is performed using Omega Calculator's `reachable` function. Its syntax is

R := reachable of *goal-state-label* in (*list of state labels*)
{ *state-label* : *start-state* |
state-label -> *state-label* : *transition* };

The | symbol means "or". For example, suppose the pre-state of a `course` object is `#max = 10 && #size = 0`, and we want to know whether multiple invocations of `addStudent(s)` can change the state of `course` object to `#max = 10 && #size = 5`. In this case, according to the specification of `addStudent`, only `#size` is affected by the method. So, form a state by the name `t` that contains only `size`, which denotes the pre-state of `#size`. Because the state is changed by the method, the post-state of `t` contains a new label `newsize`.

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 transition that changes the state. The `exists` clause specifies the constraints on the state transition, which has the same syntax as Eq. 1. The fourth line comes from the method's precondition and the last two lines from its postcondition. The symbol `null` is substituted with a hash value that represents a known constant; otherwise, Omega Calculator will regard it as an unknown label.

The above multi-step reachability test can also be simplified by REBID using the binding table into:

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

The value 0 is retained in the start state of `t` because the state label `#size` is affected by the method. On the other hand, `#max` can be removed because it is not affected. The existential quantification in the method's post-condition is also removed because it only specifies a binding of the object's state label with the method argument, which is recorded in the binding table.

Evaluation of the 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, an intersection is performed between the returned set and the test case requirement (i.e., `#size = 5`) because, in general, two sets with non-empty intersection (i.e., can satisfy both sets) may not contain each other as a subset:

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

which yields the answer

```
{[5]}
```

This means that the reachability test is successful. If the test fails, Omega Calculator will return

```
{[size] : FALSE}
```

which means that the intersection is an empty set.

Note that the multi-step reachability test formulated in this section can also be used to perform single-step reachability test. But, Omega Calculator takes more time to perform single-step reachability test using the `reachable` function than simple constraint satisfaction. Therefore, in REBID, single-step reachability test is solved using constraint satisfaction and multi-step reachability test is solved using the `reachable` function.

5. Experiments and Discussions

Five variations of the REBID planner were tested:

- 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.

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.

Three test cases were performed based on the example specifications given in Section 3. These specifications were chosen because they are simple and straightforward, and yet rich enough to illustrate various important aspects of the REBID algorithm. If more complex specifications were chosen, then it would be very difficult to really study and understand the behavior of REBID, let alone using it in practical applications.

5.1. 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 course1
course1.#max = 10
course1.teacher = teacher1
teacher1.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. It had already searched through 373 plans with a search depth of 106. This happened because of the following reason. DFS happened to choose the constructor that constructs a `Course` object with `#max` = 1 first. Then, it picked `incMax` to try to increase `#max`. After choosing `incMax`, the precondition of `decMax` became satisfiable, and `decMax` was always tried first before `incMax`. So, DFS was trying alternate invocations of `incMax` and `decMax`, which could not produce a valid plan no matter how long the sequence of instructions was.

If we swapped the sequence of `incMax` and `decMax` in the specification so that DFS always tried `incMax` before `decMax`, then DFS could also find a valid plan. The test results for this case are:

	BFS	DFS	0RT	1RT	MRT
run time (s)	0.50	0.73	0.01	0.17	0.19
depth	4	12	3	3	3
no. of plans	33	61	5	5	5

Heuristic search still executed more efficiently. The plans generated by BFS and heuristics search were:

```
Course course1 = new Course(10);
Teacher teacher1 = new Teacher(Ms_Lee, 1);
course1.setTeacher(teacher1);
```

The plan generated by DFS was:

```
Course course1 = new Course();
course1.setTeacher(teacher1);
course1.incMax();
... // repeat 8 times
course1.incMax();
Teacher teacher1 = new Teacher("Ms Lee", 1);
```

This test case shows that, without the use of heuristics and reachability tests, DFS is very sensitive to the sequence of method specifications. On the other hand, the heuristics algorithms significantly shorten the execution time by directing the search along paths that are more likely to succeed.

5.2. Test Case 2

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

```
Course course1
Course1.#max = 10
course1.#size = 2
```

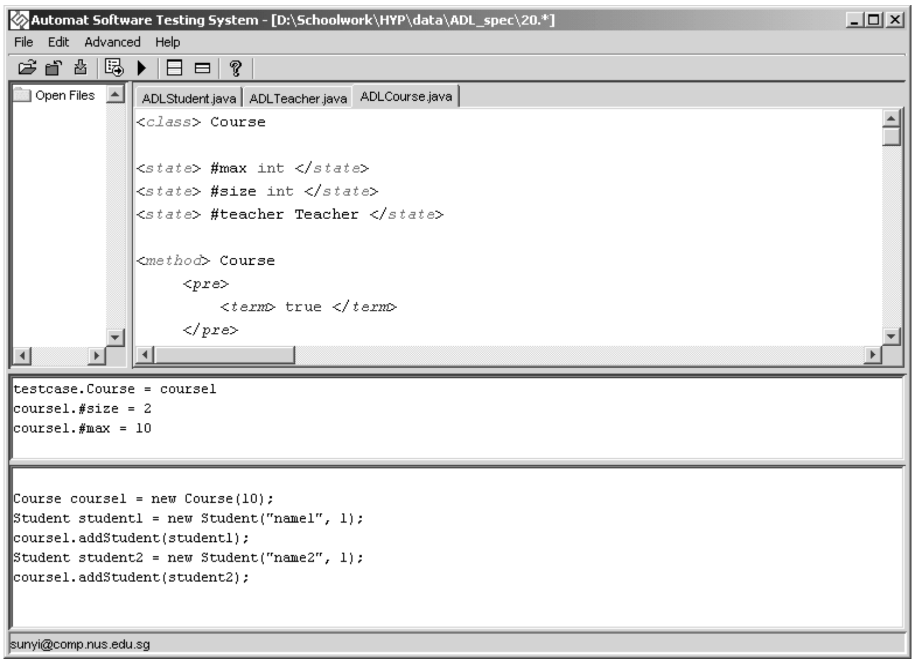
For test Case 2, DFS again could not find a valid plan after executing for 2 minutes and was aborted:

	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

This time, it was trying alternate invocations of `addStudent` and `deleteStudent`. 0RT, without reachability test, also suffered the same problem. On the other hand, 1RT and MRT could obtain valid plans, and they executed more efficiently than BFS did.

Same as for Test Case 1, by swapping the sequence of `addStudent` and `deleteStudent` in the specification, DFS and 0RT can also generate a valid plan. In this case, the test results are as follows:

	BFS	DFS	0RT	1RT	MRT
run time (s)	4.86	4.08	0.26	0.50	0.61
depth	6	14	5	5	5
no. of plans	202	65	9	9	9



```
<class> Course

<state> #max int </state>
<state> #size int </state>
<state> #teacher Teacher </state>

<method> Course
     <pre>
         <term> true </term>
     </pre>
```

```
testcase.Course = course1
course1.#size = 2
course1.#max = 10
```

```
Course course1 = new Course(10);
Student student1 = new Student("name1", 1);
course1.addStudent(student1);
Student student2 = new Student("name2", 1);
course1.addStudent(student2);
```

Figure 1. A screen-shot of the execution of the REBID planner for Test Case 2.

Figure 1 illustrates a screen-shot of the execution of REBID for Test Case 2. The top-right pane shows an internal representation of the class specification. The middle pane shows the test case constraints that must be satisfied by the required `Course` object. The bottom pane shows the sequence of instructions generated for creating the required `Course` object. The heuristic search algorithms and BFS generated the same plan as given in Fig. 1. The plan generated by DFS was:

```
Course course1 = new Course();
course1.incMax();
... // repeat 8 times
course1.incMax();
Student student1 = new Student("name1", 1);
course1.addStudent(student1)
Student student2 = new Student("name2", 1);
course1.addStudent(student2)
```

5.3. 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 course1
course1.#max = n
course1.#size = 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 5.1 and 5.2. 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 $n > 3$. More details about BFS's behavior is discussed below.

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 2 illustrates the algorithms' performance. 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, even though its search depth increased linearly with n. For $n = 3$, it required 73 sec. For $n > 3$, it could not find a valid plan within 2 minutes and was aborted.

6. 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. More-

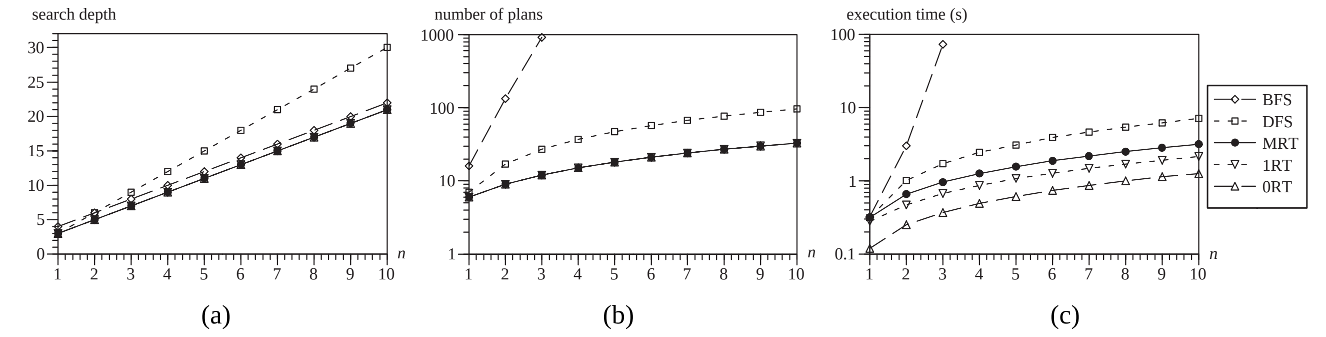


Figure 2. Performance of search algorithms with increasing n. (a) Search depth, (b) number of plans generated, and (c) search time (sec). In (a), BFS's search depths for $n > 3$ are projected values based on its search depths for $n \leq 3$.

over, 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. For example, it can compute the likelihood value from the difference between the desired size and the current size.

Another way to enhance reachability tests is to perform *partial* reachability tests. 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. The cardinality of the reachable subset can also be used a measure of the likelihood of success for heuristic search.

References

[1] B. Beizer. *Software Testing Techniques*. Thomson Computer Press, 2nd edition, 1990.

[2] W. K. Chan, T. Y. Chen, and T. H. Tse. An overview of integration testing techniques for object-oriented programs. In *Proc. of 2nd ACIS Annual Int. Conf. on Computer and Information Science (ICIS)*, pages 696–701, 2002.

[3] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. on Software Engineering and Methodology*, 10(1):56–109, 2001.

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

[5] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. on Software Engineering and Methodology*, 10(2):184–208, 2001.

[6] 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.

[7] IEEE. *IEEE Standard 829-1991: Standard for Software Test Documentation*. IEEE Press, New York, 1991.

[8] L. J. Jagadeesan, A. A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments. In *Int. Conf. on Software Engineering*, pages 525–535, 1997.

[9] 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.

[10] 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.

[11] M. Obayashi, H. Kubota, S. P. McCarron, and L. Mallet. The assertion based testing tool for OOP: ADL2. In *Proc. Int. Conf. Software Engineering*, 1998.

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

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

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

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

[16] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.

[17] S. Sankar and R. Hayes. Specifying and testing software components using ADL. Technical Report TR-94-23, Sun Microsystems Labs, 1994.

[18] 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.