Automated Generation of Test Programs From Closed Specifications of Classes and Test Cases

Wee Kheng Leow, Siau Cheng Khoo, and Yi Sun
Dept. of Computer Science, National University of Singapore
leowwk, khoosc, sunyi@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 and databases. 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 evaluated by a test evaluation tool.

This paper 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 specifications of the classes. With closed specifications, every class method is defined in terms of other methods which are, in turn, defined in their own class specifications. The core of the test program generator is a partial-order planner which plans the sequence of instructions required in the test program. The planner is, in turn, implemented as a tree-search algorithm. It makes function calls to the Omega Calculator library, which solves the constraints given in the test cases. A first-cut implementation of the planner has been completed, which is able to handle simple arithmetics and existential quantifications in the class specifications. A soundness and completeness proof sketch of the planner is also provided in this paper.

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]. Although there has been steady advancement in formal methods for program verification, testing remains the primary method for discovering faults in software systems. Automation of the testing process could reduce development costs and improve software quality.

Specification-based testing involves three main stages [16]: (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 and databases. 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 evaluated by a test evaluation tool.

Test execution and test result evaluation are easy to automate, and tools for these stages in software testing are already available. There has also been much research on automated specification-based software testing focusing on the automated selection or generation of test cases [16]. This paper complements this trend of research 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 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 validation of the results returned by the class methods, 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.

2. Background and Related Work

Most research on automated specification-based testing has focused on the automated generation of test cases [8, 16, 20]. For example, Donat developed a technique for generating test cases from specifications that contain quan-

tifications [4]. Offutt and Liu presented a method for generating test cases from specifications written in SOFL, which is a kind of formal specification language [14]. Memon et al. developed a method based on AI planner to generate test cases for testing GUI [11]. Scheetz et al. also applied AI planner but it was used to generate test cases from test objectives derived from UML models [20]. 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 [10] 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 [19] and its successor, ADL2 [13].

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.

The research work described in this paper 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 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 itself. The target programming language is Java because it is a practically useful language 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.

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 [18]. Moreover, the *partial-order planner* can plan a sequence of instructions that are only partially ordered but not totally ordered [18]. As discussed above, AI planner has also been applied to generate test cases from specifications [11, 20]. So, it is a very useful tool for automated software testing.

Our planner is, in turn, implemented as a tree-search algorithm (Section 5). It makes function calls to the Omega Calculator library [17], which solves the constraints given

by the test cases and obtains valid variable instances. A first-cut implementation of the planner has been completed, which is able to handle simple arithmetics and existential quantifications in the class specifications. A soundness and completeness proof sketch of the planner is also provided in this paper (Section 4.4).

3. Closed Specifications of Classes

In our system, the behavior of the classes are specified using an ADL-like specification language. The following example shows the specification of two classes: `Student`, which is an atomic class, and `Course`, which is an aggregate class.

```
class Student {
  Student(String name)
  { name != null   // precondition
    --> #name = name  // postcondition
  }

  String name()
  { true --> name() = #name }
}

class Course {
  Course(String code, int capacity)
  { code != null && capacity > 0
    --> #size = 0 && #code = code &&
        #cap = capacity
  }

  String code()
  { true --> code() = #code }

  int capacity()
  { true --> capacity() = #cap }

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

  void add(Student student)
  { student != null &&
    #size < #capacity
    --> #size = @#size + 1 &&
      exists(#s in Course){#s = student}
  }

  boolean registered(Student student)
  { true
    --> registered(student) =
      exists(#s in Course){#s = student}
  }
}
```

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. Note that the semantics of all the methods in the two classes are completely specified within them. That is, the specification is closed.

Test generation for closed specifications does possess difficulties such as the *cyclic definition problem*—A uses B in its specification and B uses A in turn. Methods for handling these difficulties are described in Section 5.2.

4. Automated Generation of Test Programs

The IEEE Standard 829 [9] defines a Test Case Specification as a document that consists of seven parts:

Part 1 Test case specification identifier

Part 2 Test items: a list of functions that this test case will exercise

Part 3 Input specifications: inputs to the functions

Part 4 Output specifications: expected results of the functions

Part 5 Environmental needs: special hardware or software needed

Part 6 Special procedural requirements: constraints on procedures that exercise this test case

Part 7 Intercase dependencies: a list of test cases that must be exercised before this test case is exercised

Test case generators usually produce information on Parts 1–4 only. Information in Parts 5–7 are included in other documents such as test plans or test procedures ([16], Chap. 2). In our research work, we use Parts 1–4 of a test case to generate test programs. Moreover, our framework is able to generate information regarding intercase dependencies (i.e., Part 7) automatically given the closed class specifications and the test cases (Section 5.2).

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. It is straightforward to automatically generate program codes for steps 2 and 3 but not so for step 1:

- 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 (e.g., create a stack with 10 elements). Additional modifier methods (e.g., stack push) may need to be invoked to bring the object to the required state.

In the remainder of this paper, we will focus on the automated generation of object construction codes. A concise definition of the problem is first defined in Section 4.1, which leads to a planning algorithm called REBID (REcursive BIDirectional planner) for the generation of object construction codes (Section 4.2).

4.1. Problem Statement

The problem of automated generation of object construction codes can be specified as follows:

> Given the closed specifications of classes, the class C of a target object x, and the condition R (which does not contain conflicting terms) that x must satisfy (as described in a test case), generate the object construction codes that, when executed, will create x that satisfies $x.R$, assuming that such object construction codes exist.

The notation $x.R$ means that the methods in R, if any, are applied on x. The condition R is given as a conjunction of terms of the form $x_i.M_i() = v_i$, where $x_i.M_i()$ is the application of access method M_i on x_i, and v_i denotes a free variable, a string literal, or a constant of primitive data type. Conditions with more complex forms can be reduced to this *canonical form* (see Section 4.2 for details).

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,

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

$C\ x = \mathsf{new}\ C(u_1, \ldots, u_n)$; // 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.

A *recursive bidirectional* planner called REBID has been developed to generate object construction codes. 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.

To further facilitate the automation of the testing process, we adopt Gries's semantics of method invocation [6]: $\{P\}\, M\, \{Q\}$. That is, if method M is invoked in a state satisfying the precondition P, then M is *guaranteed* to terminate in a finite amount of time in a state satisfying the postcondition Q. Note that Gries's notation denotes *total correctness* which is slightly different from Hoare's notation of $P\, \{S\}\, Q$ which denotes *partial correctness* [7].

4.2. Recursive Bidirectional Planner

REBID performs bidirectional plan-space search [18] for a plan (i.e., a sequence of instructions). A (partial) plan is a 5-tuple $\langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle$ where $\mathcal{R}$ is the set of test case conditions that the target object must satisfy, $\mathcal{I}$ is a list of instructions, $\mathcal{L}$ is a list of ordering information of the corresponding instructions in $\mathcal{I}$, $\mathcal{B}$ contains variable bindings, and $\mathcal{Q}$ contains constraints on the variables (which are derived from methods' pre- and postconditions).

Each instruction I_i in $\mathcal{I}$ is a 5-tuple $\langle z_k, M_i, \mathbf{u}_i, \mathcal{R}_k, \mathcal{S}_k \rangle$ where z_k refers to an object, M_i is the constructor of z_k or a method to be applied on z_k, $\mathbf{u}_i$ is a list of arguments of M_i, $\mathcal{R}_k$ is a set of conjunctive terms to be satisfied by z_k, and $\mathcal{S}_k$ is the set of conjunctive terms currently satisfied by z_k. $\mathcal{R}_k$ represents the subgoal that an object must satisfy and $\mathcal{S}_k$ represents a part of the subgoal that will be satisfied after instruction I_i is executed.

Note that the object z_k and the condition sets $\mathcal{R}_k$ and $\mathcal{S}_k$ have the same subscripts because the condition sets are associated with an object instead of an instruction. On the other hand, I_i, M_i, and $\mathbf{u}_i$ have the same subscripts because a method and its arguments are associated with an instruction. Each piece of ordering information $\mathcal{O}_i$ in $\mathcal{L}$ is associated with an instruction I_i. It is an ordered list that describes the sequence of instructions before and after I_i.

For notational convenience, we use both condition R_i and condition set $\mathcal{R}_i$ to represent a conjunctive condition. In particular, $R_i = \bigwedge_{r_j \in \mathcal{R}_i} r_j$. Note that, with this notation, $R_i \wedge R_j \Leftrightarrow \mathcal{R}_i \cup \mathcal{R}_j$. Moreover, we define $R_i \leq R_j \Leftrightarrow$

$\mathcal{R}_i \subseteq \mathcal{R}_j$, and $R_i = \text{true} \Leftrightarrow \mathcal{R}_i = \emptyset$ (though $\mathcal{R}_i = \{\text{true}\}$ would work just the same).

Before presenting the planner algorithm, let us briefly describe a procedure that converts the conjunctive terms in a set $\mathcal{R}$ into *canonical forms*:

- Instantiate existential quantifications into unique variables.
- Replace conditions of the form $x.A()$ *op* $y.B()$, where *op* is a comparator, into two conditions $x.A()$ *op* v and v *op* $y.B()$ by introducing the free variable v.
- Replace a multidot method invocation of the form $x.M_0().M_1().\cdots.M_n()$ *op* v by a conjunction of single-dot method invocations $x.M_0() = x_1 \wedge x_1.M_1() = x_2 \wedge \cdots \wedge x_n.M_n()$ *op* v by introducing free variables $x_1, \ldots, x_n$.
- Replace access method invocations by appropriate state labels according to class specifications.

After canonization, all the terms in $\mathcal{R}$ are of the form $x.\#l$ *op* v for some object variable x, label $\#l$, operator *op*, and value v. Examples of the application of this procedure are illustrated in Section 4.3. A set $\mathcal{R}$ in canonical form can be divided into mutually exclusive subsets such that each subset $\mathcal{R}(x_i)$ contains all the terms that refer to a particular object x_i. That is, $\mathcal{R}(x_i) = \{r_j \in \mathcal{R} \mid r_j \equiv x_i.\#l_j \text{ op } v_j\}$, $\mathcal{R}(x_i) \cap \mathcal{R}(x_j) = \emptyset$ for any $x_i \neq x_j$, and $\bigcup_i \mathcal{R}(x_i) = \mathcal{R}$ (i.e., $\bigwedge_i R(x_i) = R$).

REBID can be most succinctly described in terms of the following *nondeterministic* algorithms.[1] Note that although REBID is conceptually a recursive planner, it is easier to describe the algorithm by implementing recursion as iteration. Given a canonical $\mathcal{R}$ that a target object x needs to satisfy, the instruction sequence for constructing x can be generated by applying **MakePlan**:

MakePlan $(x, \mathcal{R})$

1. Initialize condition sets: $\mathcal{R}_0 \leftarrow \mathcal{R}(x)$, $\mathcal{S}_0 \leftarrow \emptyset$.
2. Create initial instruction: $I_0 = \langle z_0, \text{nil}, \text{nil}, \mathcal{R}_0, \mathcal{S}_0 \rangle$.
3. Create initial plan: $\wp \leftarrow \langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle$ where $\mathcal{I} = \{I_0\}$, $\mathcal{L} = \{\mathcal{O}_0\}$, $\mathcal{O}_0 = \{I_0\}$, $\mathcal{B} = \{x = z_0\}$, and $\mathcal{Q} = \emptyset$.
4. Repeat
   (a) If all conditions in $\mathcal{R}$ have been satisfied, i.e., $\bigcup_k \mathcal{S}_k \Rightarrow \mathcal{R}$, and $M_i \neq \text{nil}\ \forall I_i \in \mathcal{I}$, and $\mathcal{Q}$ is satisfiable, then instantiate unbound variables such that $\mathcal{Q}$ is satisfied and return plan $\wp$.
   (b) Else, **CreateObject**($\wp$) or **ModifyObject**($\wp$).

CreateObject $(\langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle)$

1. **Choose** an instruction $I_i = \langle z_k, M_i, \mathbf{u}_i, \mathcal{R}_k, \mathcal{S}_k \rangle \in \mathcal{I}$, for some k, such that $M_i = \text{nil}$, $\mathbf{u}_i = \text{nil}$.
2. **Choose** a constructor $C(\mathbf{a})$ with semantics $\{P\}\, C(\mathbf{a})\, \{Q\}$ and argument list $\mathbf{u}$ such that the condition $P^{\mathbf{a}}_{\mathbf{u}} \wedge (z_k.Q^{\mathbf{a}}_{\mathbf{u}} \Rightarrow z_k.R')$ is true, for some $R' \leq R_k$, with appropriate bindings of free variables in R' to arguments in $\mathbf{u}$ or instantiated state labels of z_k.
3. Set method and arguments of instruction I_i: $M_i \leftarrow C$, $\mathbf{u}_i \leftarrow \mathbf{u}$.
4. Set the conditions that will be satisfied if I_i is executed: $\mathcal{S}_k \leftarrow \{\text{terms in } Q^{\mathbf{a}}_{\mathbf{u}}\}$.
5. Note variable binding: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{bindings in Step 2}\}$.
6. Include constraints: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\text{terms in } P^{\mathbf{a}}_{\mathbf{u}}\} \cup \{\text{terms in } z_k.Q^{\mathbf{a}}_{\mathbf{u}}\}$.
7. **ExpandArguments** $(\langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle,\ i,\ \mathcal{P})$, where $\mathcal{P} = \{\text{terms in } P^{\mathbf{a}}_{\mathbf{u}}\}$, i.e., create arguments in $\mathbf{u}$ that satisfy $\mathcal{P}$.

The keyword **Choose** means nondeterministic selection. The notation $P^{\mathbf{a}}_{\mathbf{u}}$ refers to the condition P with formal arguments in $\mathbf{a}$ replaced simultaneously by actual arguments in $\mathbf{u}$ [6].

ModifyObject $(\langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle)$

1. **Choose** an instruction $I_i = \langle z_k, M_i, \mathbf{u}_i, \mathcal{R}_k, \mathcal{S}_k \rangle$, for some k, such that I_i is the last instruction in $\mathcal{I}$ for object z_k, $M_i \neq \text{nil}$, and $\mathcal{S}_k \not\Rightarrow \mathcal{R}_k$, i.e., method M_i has been identified but conditions have not been fully satisfied.
2. **Choose** a modifier $M(\mathbf{a})$ with semantics $\{P\}\, M(\mathbf{a})\, \{Q\}$ and argument list $\mathbf{u}$ satisfying
$$\{S'_k\} M(\mathbf{a}) \{S'_k\}$$
$$z_k.P^{\mathbf{a}}_{\mathbf{u}} \wedge (z_k.Q^{\mathbf{a}}_{\mathbf{u}} \Rightarrow z_k.R')$$
for some $S'_k \leq S_k$ and R' such that $(S'_k \wedge R') \leq R_k$, with appropriate bindings of free variables in R' to arguments in $\mathbf{u}$ or instantiated state labels of z_k.
3. Update conditions satisfied: $\mathcal{S}_k \leftarrow \mathcal{S}'_k \cup \{\text{terms in } Q^{\mathbf{a}}_{\mathbf{u}}\}$.
4. Create instruction: $I_j = \langle z_k, M, \mathbf{u}, \mathcal{R}_k, \mathcal{S}_k \rangle$.
5. Update plan: $\mathcal{I} \leftarrow \mathcal{I} + \{I_j\}$, $\mathcal{L} \leftarrow \mathcal{L} + \{\mathcal{O}_j\}$ where $\mathcal{O}_j = \{I_j\}$.
6. Update ordering information: $\mathcal{O}_l \leftarrow \mathcal{O}_l + \{I_j\}$, where $I_l = \langle z_k, C_l, \mathbf{u}_l, \mathcal{R}_k, \mathcal{S}_k \rangle$ is an instruction that creates z_k and C_l is a constructor method. That is, I_j comes *after* the instruction that creates object z_k.
7. Note variable binding: $\mathcal{B} \leftarrow \mathcal{B} \cup \{\text{bindings in Step 2}\}$.
8. Update constraints: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\text{terms in } z_k.P^{\mathbf{a}}_{\mathbf{u}}\} \cup \{\text{terms in } z_k.Q^{\mathbf{a}}_{\mathbf{u}}\}$.
9. **ExpandArguments** $(\langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle,\ j,\ \mathcal{P})$, where $\mathcal{P} = \{\text{terms in } P^{\mathbf{a}}_{\mathbf{u}}\}$.

1 Most AI planner algorithms are described as nondeterministic algorithms, which are then implemented as deterministic tree-search [18].

The '+' operators in steps 5 and 6 denote list concatenation.

ExpandArguments ($\langle \mathcal{R}, \mathcal{I}, \mathcal{L}, \mathcal{B}, \mathcal{Q} \rangle, j, \mathcal{P}$)

For each $u_i \in \mathbf{u}_j$ of instruction $I_j = \langle z_m, M_j, \mathbf{u}_j, \mathcal{R}_m, \mathcal{S}_m \rangle \in \mathcal{I}$ that is not a string literal and not a constant of primitive data type:

1. Update condition sets: **Choose**, based on the class specifications, an x_k from the free variables in $\mathcal{R}$ that can be bound to u_i. Update $\mathcal{R}_i \leftarrow \mathcal{R}(x_k) \cup \mathcal{P}(u_i) \cup \{x_k = u_i\}$, $\mathcal{S}_i \leftarrow \emptyset$.
2. Create instruction: $I_l = \langle u_i, \text{nil}, \text{nil}, \mathcal{R}_i, \mathcal{S}_i \rangle$.
3. Update plan: $\mathcal{I} \leftarrow \mathcal{I} + \{I_l\}$, $\mathcal{L} \leftarrow \mathcal{L} + \{\mathcal{O}_l\}$ where $\mathcal{O}_l = \{I_l\}$.
4. Update ordering information: $O_j \leftarrow \{I_l\} + O_j$, i.e., I_l comes *before* the instruction I_j that creates or modifies object z_m.
5. Update variable binding: $\mathcal{B} \leftarrow \mathcal{B} \cup \{x_k = u_i\}$.

In the above algorithm, the nondeterministic **Choose** selects five types of candidates: instructions, class methods, arguments, free variables, and subsets of conditions. There are finite and enumerable numbers of instructions, class methods, free variables, and condition sets. Thus, in the explanatory example in the next section, the correct candidates can be nondeterministically chosen. Selection of arguments is more complicated because there is potentially an infinite number of possible values and they may need to satisfy the preconditions of several methods (see next section for example). 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 collected in $\mathcal{Q}$ and the values can only be determined at the end of the planning process (step 4(a) of **MakePlan**) by proper instantiation of the argument variables.

4.3. A Nondeterministic Example

This section uses the specification example given in Section 3 to explain the nondeterministic algorithm presented in the preceding section. Suppose a test case requires a `Course` object `c0?` that satisfies the condition $\texttt{c0?}.R \equiv$ `c0?.size() = 2 && s1?.name() = "Tim" && c0?.registered(s1?) = true`. For notational clarity, free or unbound variables are postfixed with the '?' symbol. After canonizing $\texttt{c0?}.R$, we obtain two mutually exclusive subsets $\mathcal{R}(\texttt{c0?}) = \{R1, R2\}$ and $\mathcal{R}(\texttt{s1?}) = \{R3\}$:

($R1$) `c0?.#size? = 2`
($R2$) `c0?.#s? = s1?`
($R3$) `s1?.#name? = "Tim"`

Note that the existential quantification of the method `registered` has been instantiated with the unbound label `#s?` and the unbound variable `s1?`.

Given the test case, **MakePlan** first identifies the conditions that must be satisfied by the target object (step 1), which is instantiated as `c0`. These terms are included in the condition set $\mathcal{R}_0$ of `c0` giving $\mathcal{R}_0 = \{R1, R2\}$. In step 2, it creates the first instruction $I_0 = \langle \texttt{c0}, \text{nil}, \text{nil}, \mathcal{R}_0, \mathcal{S}_0 \rangle$, with $\mathcal{S}_0$ initialized to $\emptyset$. Step 3 creates a plan with one instruction I_0 and the first variable binding in $\mathcal{B}$ is `c0?` = `c0`.

Next, **CreateObject** is executed. In step 1, the only instruction in the plan, I_0, is chosen for expansion. In step 2, the only constructor available, `Course`, is chosen for I_0. This constructor's postcondition does not satisfy any of the terms in $\mathcal{R}_0$. In particular, it says that `size()`, which is the same as `#size`, is equal to 0 instead of 2 as required in $\mathcal{R}_0$. Thus, R' is just taken as true.

At this time, there is insufficient information for instantiating the method arguments. So they are left unbound. To ensure that instruction I_0 can indeed be executed, the arguments must be chosen such that they satisfy the constructor's precondition $P^{\mathbf{a}}_{\mathbf{u}}$. So, terms in $P^{\mathbf{a}}_{\mathbf{u}}$ are collected in the plan's constraint set $\mathcal{Q}$ (step 6 of **CreateObject**).

Now, I_0, the instantiated pre- and postconditions P_0 and Q_0, and the satisfied condition set $\mathcal{S}_0$ at this step are:

I_0 : $\langle$`c0`, `Course`, {`code0`, `cap0`}, $\mathcal{R}_0$, $\mathcal{S}_0\rangle$
$\mathcal{S}_0 = \emptyset$
P_0 : `code0 != null && cap0 > 0`
Q_0 : `c0.#size0 = 0 && c0.#code0 = code0 && c0.#cap0 = cap0`

Step 6 adds terms in P_0 and Q_0 into the constraint set $\mathcal{Q}$. The last step of **CreateObject** executes **ExpandArguments** on I_0. But the arguments of I_0 take values of primitive data types. So, nothing is done.

Following **CreateObject**, the method **ModifyObject** is executed and it nondeterministically chooses the `add` method to modify `c0`. With appropriate variable bindings (see below), the postcondition of `add` can satisfy condition $R2$. The condition $\{S'_k\}M(\mathbf{a})\{S'_k\}$ in step 2, together with step 3, means that the terms in the current $\mathcal{S}_0$ of object c_0 that remain unchanged after applying `add` are kept in the new $\mathcal{S}_0$. In addition, terms in $Q^{\mathbf{a}}_{\mathbf{u}}$ are satisfied by the application of `add`, and are also included into the new $\mathcal{S}_0$. A new instruction I_1 is created, and step 6 updates the ordering information $\mathcal{O}_0$ to indicate that I_1 comes after I_0. Step 7 updates the binding set by adding B_1 into $\mathcal{B}$ and step 8 updates the constraint set by adding P_1 and Q_1 into $\mathcal{Q}$. Now, we obtain:

I_1 : $\langle$`c0`, `add`, {`s1`}, $\mathcal{R}_0$, $\mathcal{S}_0\rangle$
$\mathcal{S}_0 = \{R_2\}$
P_1 : `s1 !=null && c0.#size0 < c0.#cap0`
Q_1 : `c0.#size1 = c0.#size0 + 1 = 1 && c0.#s1 = s1`
B_1 : `s1?` = `s1`, `c0?.#s?` = `c0.#s1`

Note that the existential quantification in `add`'s post-

condition has been instantiated. The quantified variable `#s` which ranges over the aggregate object `Course` is instantiated as a state label `#s1` of object `c0`. The label `#size` is instantiated as `#size0` and `#size1` to represent the pre-state and post-state of `#size`. That is, an ordered sequence of labels is used to denote the conditions satisfied by an object after executing some instructions. The last label in this sequence would denote the object's final conditions.

Next, **ExpandArguments** is executed for creating the argument `s1` of I_1. Step 1 collects the condition terms in $\mathcal{R}$ and $\mathcal{P}$ that should be satisfied by `s1?` and `s1` respectively giving $\mathcal{R}_1 = \{R_3 \wedge$ `s1 != null`$\}$. Step 2 creates a new instruction I_2 for creating `s1`. Step 4 updates the order information O_1 of I_1 to indicate the I_2 precedes I_1.

Next, **CreateObject** nondeterministically chooses instruction I_2 for expansion, which selects `Student` as the constructor. The postcondition of `Student` can satisfy condition terms in $\mathcal{R}_1$ giving:
I_2 : $\langle$`s1`, `Student`, {`"Tim"`}, $\mathcal{R}_1$, $\mathcal{S}_1\rangle$
$\mathcal{S}_1$ = $\mathcal{R}_1$
P_2 : `"Tim" != null`
Q_2 : `s1.#name1 = "Tim"`
B_2 : `s1?` = `s1`, `s1?.#name?` = `s1.#name1`

Next, I_1 is selected for expansion again, and `add` is selected as the modifier. The process of creating a `Student` object is executed, resulting in new instructions I_3 and I_4:
I_3 : $\langle$`c0`, `add`, {`s2`}, $\mathcal{R}_0$, $\mathcal{S}_0\rangle$
$\mathcal{S}_0$ = $\{R_1, R_2\} = \mathcal{R}_0$
P_3 : `s2 != null && c0.#size1 < c0.#cap0`
Q_3 : `c0.#size2 = c0.#size1 + 1 = 2 && c0.#s2 = s2`
B_3 : `c0?.#size?` = `c0.#size2`

I_4 : $\langle$`s2`, nil, nil, $\emptyset, \emptyset\rangle$

After creating object for I_4, we obtain:
I_4 : $\langle$`s2`, `Student`, {`name2`}, $\emptyset, \emptyset\rangle$
P_4 : `name2 != null`
Q_4 : `s2.#name2 = name2`

At this point, $\mathcal{R}$ can be satisfied by $\mathcal{S}_0 \cup \mathcal{S}_1$ and all the methods in the instructions have been identified. The remaining unbound variables are bound to appropriately chosen values according to the constraints in $\mathcal{Q}$. In particular, there is no constraint for `code0` and `name2`. So they can be bound to any randomly generated strings such as `"cs101"` and `"Jim"` respectively. The argument `cap0` is bound to `c0.#cap0` which is constrained to be larger than `c0.#size1` which is equal to 1. Therefore, `cap0` can be bound to any value larger than 1 such as 2. The instruction sequence is encoded in $\mathcal{L}$, and the following instructions can be generated from the plan:

```
Course c0 = Course("cs101", 2);  // I0
Student s1 = Student("Tim");     // I2
c0.add(s1);                      // I1
Student s2 = Student("Jim");     // I4
c0.add(s2);                      // I3
```

4.4. Soundness and Completeness

Theorem 1 *REBID is sound and complete.*

Proof sketch: (Soundness) To prove that a plan generated by REBID is correct, we need to show that (1) the sequence of instructions can be executed successfully, and (2) after executing the instructions, the target object satisfies the condition R specified in the test case.

1. Step 6 of **ModifyObject** places object modification instruction after object creation instruction and step 4 of **ExpandArguments** places argument creation instruction before object creation instruction. That is, a method is invoked after its arguments are created, and an object is modified after it is created. In addition, REBID ensures that the precondition of the method in each instruction I_i is satisfied. Therefore, the sequence of instructions can be executed successfully.
2. The methods are chosen to satisfy the condition $P^{\mathbf{a}}_{\mathbf{u}} \wedge (z_k.Q^{\mathbf{a}}_{\mathbf{u}} \Rightarrow z_k.R')$. It follows directly from Gries's *theorem of procedure call* [6] that the constructed object or argument z_k will indeed satisfy $z_k.R'$ because the instruction will terminate successfully (since $P^{\mathbf{a}}_{\mathbf{u}}$ is true) and the postcondition $z_k.Q^{\mathbf{a}}_{\mathbf{u}}$ will imply $z_k.R'$. During the planning process, the condition terms in $\mathcal{R}$ that are satisfied are recorded in the condition sets $\mathcal{S}_k$. When the planner terminates, $\mathcal{S}_k \Rightarrow \mathcal{R}_k$ for each z_k and $\bigcup_k \mathcal{S}_k \Rightarrow \mathcal{R}$. That is, the arguments satisfy their respective requirements, and the target object and its components as a whole satisfy R, if there is no conflict in $\bigcup_k \mathcal{S}_k$. But, there can be no conflict because
   (a) there is no conflict in $\mathcal{R}$ and the sets $\mathcal{R}(x_i)$ are mutually exclusive, and
   (b) in **ModifyObject**, only the conditions that are unchanged (i.e., $\mathcal{S}'_k$) and those that are satisfied by the modifier (i.e., $Q^{\mathbf{a}}_{\mathbf{u}}$) are placed in $\mathcal{S}_k$.

(Completeness) To prove that REBID is complete, we need to show that REBID can generate a correct sequence of instructions if one exists.

1. A class has a finite number of constructors. So, step 2 of **CreateObject** can enumerate all possible choices of constructors.
2. A class has a finite number of modifiers. So, step 2 of **ModifyObject** can enumerate all possible choices of modifiers. In addition, the length of a correct instruction sequence is finite. So, step 4(b) of **MakePlan** can enumerate all possible sequences of modifiers.
3. A method has a finite number of arguments. So, we only need to show that REBID can find the correct arguments.

$$
\begin{array}{lcl}
R & ::= & \phi \mid \neg\phi \mid \exists v.\phi \mid \forall v.\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \\
\phi & ::= & True \mid False \mid a_1 = a_2 \mid a_1 \neq a_2 \\
 & & \mid a_1 < a_2 \mid a_1 > a_2 \mid a_1 \leq a_2 \mid a_1 \geq a_2 \\
a & ::= & n \mid v \mid n * v \mid a_1 + a_2 \mid -a \\
n & \in & \text{Integer Constants} \\
v & \in & \text{Variables}
\end{array}
$$

Figure 1. Syntax of Presburger formulae.

- If a method argument is an object instance of a class, then **ExpandArgument** is invoked for each argument, which creates subproblems that REBID can solve recursively.
- If a method argument is a primitive constant or string literal that is specified in target condition R, then REBID can use the value specified.
- For the case that a method argument is not specified, REBID collects in $\mathcal{Q}$ the preconditions of the methods used in the entire plan. These conditions are used to instantiate unbound arguments in step 4(a) of **MakePlan** after all the instructions are found. Since a correct instruction sequence exists (assumed in the problem statement), then it must be possible to instantiate the arguments for the correct sequence.

Therefore, REBID can enumerate all possible sequences of instructions whose method arguments can be instantiated, and they include the correct instruction sequence. □

5. Implementation

5.1. Deterministic Tree-Search

A prototype of the REBID algorithm has been developed as a proof of concept. The non-deterministic aspect of the algorithm has been realized by a *deterministic tree-search algorithm in depth-first manner*. The tree nodes represent partial plans for potential solutions. Depth-first search is chosen for its simplicity, and it works well for the sample specifications that we have tested so far. It is noted that, in general, a more intelligent search algorithm is needed to avoid searching the entire tree for a solution.

In the prototype, we restrict the description of program specification to *affine relations*. Specifically, the pre- and post-conditions of methods, as well as the condition sets $\mathcal{R}_k$ and $\mathcal{S}_k$ are expressed using *Presburger formulae*. Its syntax is defined in Figure 1. Consequently, the satisfiability of the conditions is known to be decidable. The current implementation uses the Omega Calculator [17] for satisfiability check, as well as instantiation of unbound variables in the final step of the algorithm (Step 4(a) of **MakePlan**).

Moreover, addition and subtraction are restricted to adding and subtracting a constant from a variable. In this case, the search algorithm can terminate even if a plan does not exist because it can decide that further additions or subtractions will cause the plan to deviate further from the requirement. For example, if an `add` method increments the `size` of an aggregate class by 2 instead of 1, then no plan exists for creating an empty aggregate object and then adding elements to the object to get odd-numbered size.

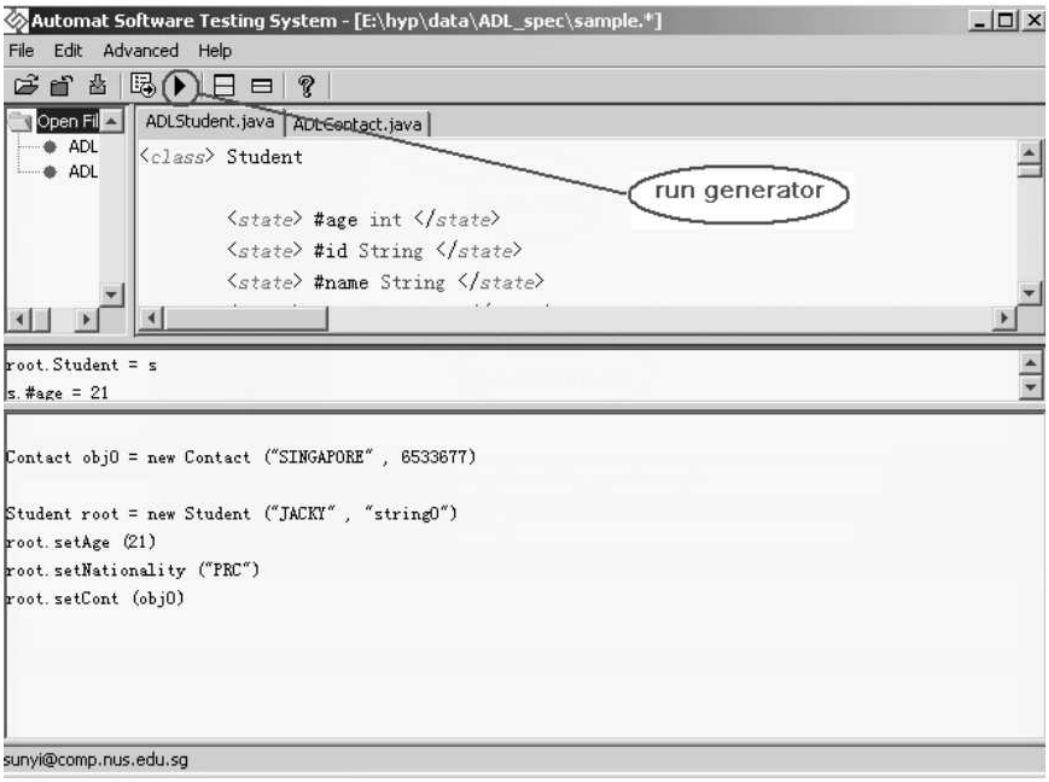
Figure 2 illustrates a screen-shot of the execution of the REBID planner. The top-right pane shows an internal representation of the class specification. The middle pane shows some of the test case conditions that must be satisfied by a `Student` object. The bottom pane shows the sequence of instructions generated for creating the required `Student` object. This example illustrates the construction of the `Student` object, followed by modification of the object's attributes. The modifier method `setCont` requires another object as its input argument, which is created as `obj0` before the `setCont` method is invoked. In summary, this example illustrates that the REBID planner can correctly sequence the instructions that invoke the correct methods.

5.2. Test Sequencing

In testing several or all the methods of a class, the various test cases for the class need to be sequenced in an appropriate order. This is not necessary for software testing systems such as ADL that test single function specification. On the other hand, it is important for testing an entire class defined by a closed specification. For example, to test the method `Course.add` at the boundary value of `size() = capacity()`, it is necessary to first add enough instances of `Student` into a `Course` object using the method `Course.add` which is under test. To make the test meaningful and useful for locating program bugs, `add` should first be tested with `size() = 0`, which is satisfied by a freshly constructed `Course` object. Subsequently, `add` can be tested with `size()` greater than 0 but smaller than `capacity()`, and finally with `size() = capacity()`.

Test sequencing consists of three steps. The first step identifies the *dependency* between the methods in a class. A method A is said to *depend on* another method B if one of the followings is satisfied:

1. B is a constructor of the class in which A is defined.
2. A's postcondition contains a call to method B, i.e., B`(b)`.
3. A's postcondition contains a state label #l or an existentially quantified expression E, and B's post-



condition contains a matching assertion of the form B(b) = #l or B(b) = E.

4. As in 3 but with A and B swapped.

Case (1) is obvious: testing of a class method is possible only after an instance of the class has been constructed. Case (2) is also obvious: the test program for method A needs to call method B to check the test result. Cases (3) and (4) refer to the case of *cyclic definition*, i.e., A uses B in its specification and vice versa. Methods involved in a cyclic definition depend on each other and have to be tested as a group called the *cyclic group*.

Typically, a cyclic group consists of a constructor or modifier and some access methods. For example, the constructor `Student` of class `Student` and the accessor `name` form a cyclic group. If an accessor appears in more than one cyclic group, then it can be removed from all except one of the groups, say G, because it needs to be tested only once together with the other methods in group G.

The second step establishes a *partial ordering* between test cases. A test case T is said to *precede* (i.e., tested before) a test case T' if one of the followings is satisfied:

1. T tests method A, T' tests a different method B not in the same cyclic group as A, and B depends on A.

2. Both T and T' test the same method A and the object under test in T has a shorter construction sequence than that in T'.

As discussed in previous paragraphs, a test of a method, say, `add` on a `Course` object with fewer elements should precede the test on an object with more elements.

The final step of test sequencing sorts the class methods according to the partial-ordering of the test cases. A test case T that precedes another test case T' should be tested before T'.

6. Discussions and Conclusion

This paper presented a closed specification system such that every class method is defined in terms of other methods which are, in turn, defined in their own class specifications. With closed specifications, it is possible to automatically generate a test program for testing the behavior of object-oriented classes given their closed specifications and the desired test cases. The test program combines the setup process, test execution, and test validation into a single program so that all three stages of software testing can be executed automatically.

This paper also presented a method for automated generation of the test program. The core of the generator is a AI planner. A first-cut implementation has been completed. In addition, a soundness and completeness proof of REBID is also provided.

We are now working on the following extensions to the current implementation. REBID can be easily extended to generate program codes for testing a method's exception

handling. This can be achieved by including the exception handling semantics $\{\neg P\}\, M\, \{E\}$ where E is the conditions that must be satisfied when the precondition P is violated. Single inheritance of a class can also be easily handled by inheriting the closed specification of the class.

The current REBID implementation does not handle universal quantifications. Universal quantifications can be eliminated through a technique called *generalization*, originally introduced by Suzuki and Ishihata [21]. Given a universally-quantified formula $\forall x\, f(x)$, we compute $\neg eliminate(\neg f(x))$. The function $eliminate$ uses the Fourier-Motzkin variable-elimination method to eliminate x from $\neg f(x)$. This results in a simplified formula with the same integer solutions as the original formula, and is used in place of the universally-quantified formula.

To handle more complex specifications and test cases, a measure of the *likelihood of success* of an instruction is needed to allow REBID to search for a plan efficiently. The likelihood can be defined in terms of the *distance* between the conjunctive terms in the subgoal $\mathcal{R}_i$ and the currently satisfied condition set $\mathcal{S}_i$. Symbolic labels in the terms can be compared syntactically whereas numerical values can be compared numerically. In addition, we require more sophisticated constraint solving tools, in place of Omega Calculator, to handle more expressive specification than Presburger formulae. Some theorem-proving tools, such as Isabelle [12], PVS [15] etc., can be employed here.

The object construction codes generated by REBID can be executed to construct objects that satisfy known conditions. Therefore, they can be reused in other test programs that require objects that satisfy the same conditions. This will reduce the need to run REBID again to generate the same object construction codes.

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] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of ACM*, 12:576–583, 1969.

[8] 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.

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

[10] 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.

[11] 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.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic (LNCS 2283)*. Springer, 2002.

[13] 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.

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

[15] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96 (LNCS 1102)*, pages 411–414. Springer-Verlag, 1996.

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

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

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

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

[20] 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.

[21] N. Suzuki and K. Ishihata. Implementation of array bound checker. In *ACM Principles of Programming Languages*, pages 132–143, 1977.