Towards Fully Automated Test Program Generation from Closed Specifications of Classes

Wee Kheng Leow and Siau Cheng Khoo

leowwk,khoosc@comp.nus.edu.sg

Technical Report No. TRA3/00

3 March 2000

School of Computing
National University of Singapore
3 Science Drive 2, Singapore 117543

Towards Fully Automated Test Program Generation from Closed Specifications of Classes

Wee Kheng Leow and Siau Cheng Khoo
leowwk,khoosc@comp.nus.edu.sg
School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543

Abstract

Among various specification languages, ADL (Assertion Definition Language) and its successor, ADL2, are best known for supporting automated generation of test programs from specifications. Being an opened specification system, ADL requires the user to provide additional supporting functions to define the semantics of a moderately complex function or module. Supporting functions are also needed for ADL to generate test programs for the function or module.

This paper proposes a *Closed Class Specification* (CCS) system for specifying the behavior of a class instead of a function. With CCS, every class method is defined in terms of other methods which are, in turn, defined in their own class specifications. Given a closed specification, it is possible to automatically generate test program for a class without the need of additional supporting functions. This paper describes the framework of CCS's test program generator and illustrates examples of how test programs are generated.

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 checking the correctness of a software. Automation of the testing process could reduce development costs and improve software quality.

Functional testing is an approach that generates test data from a software's specification for checking the software's functional correctness. Among the specification languages available such as CSP [7] and Z [15], ADL (Assertion Definition Language) [14] and its successor, ADL2 [10], are best known for supporting automatic generation of test programs. 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 (ADLT) can automatically generate a test program that executes the function or module under test and checks the test results.

ADL has two key features that support automated generation of test programs:

1. *Auxiliary functions* are used in ADL to define the semantics of the function to be tested. These auxiliary functions may or may not be specified elsewhere, and their implementations are provided by the user. We call this type of specification system an *opened specification* system.

2. Test data is specified using Test Data Description (TDD) language. If the TDD specification includes only literal descriptions of test variables (such as integers and string literals), test data can be generated by ADLT automatically. Otherwise, the user can provide implementations of the *provide functions* for constructing the required data and the *relinquish functions* for destroying the 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 single function or the partially specified module. However, an opened specification system also has the following shortcomings:

- An opened specification is often an incomplete specification—it does not contain enough information for generating test data by itself. In testing sufficiently 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, specifications of these supporting functions may, in turn, require other supporting functions.

This paper proposes a *Closed Class Specification* (CCS) system that overcomes the above shortcomings (Section 2). To fulfill this goal, the specification must be defined for a 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. The target programming language of CCS is Java because it is a practically useful language and is simpler to handle than is C++.

Fulfilling the above requirement of CCS 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. Moreover, specification is a one-time effort. 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.

Given closed specification of classes, it is possible for the CCS system to generate test programs fully automatically. The framework of CCS's test generator will be described in Section 3. 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 overcoming these difficulties are described in Section 3.

2 Closed Class Specification

The proposed *Closed Class Specification* language (CCS) is a closed specification system: all the methods referred to in a class specification are defined in some class specifications.

2.1 Overview of CCS

A CCS of a class consists of the specification of each class method.[1] A method specification consists of five parts: *method signature*, *precondition*, *entry state* (similar to ADL's call-state), *postcondition*, and *error-handling assertion*. The following example illustrates the specifications of the `Student` class and the `Course` class:

```
class Student
{
   Student(String name, String id) throws NullPointerException
      // Creates a student object.
      pre:
         #p1: name != null
         #p2: id != null
      post:
         #q1: #name = name
         #q2: #id = id
      error:
         #e1: NullPointerException

   String name()
      // Returns the name of the student.
      pre: true
      post:
         #q1: name() = #name

   String id()
      // Returns the ID of the student.
      pre: true
      post:
         #q1: id() = #id

   void setGrade(float grade) throws IllegalArgumentException
      // Updates the student's average grade.
      pre:
         #p1: grade >= 0 && grade <= 4
      post:
         #q1: #grade = grade
      error:
         #e1: IllegalArgumentException
```

[1] Class constants can also be defined in the class specification. Constants are easy to handle in test generation and are omitted in this paper.

```
   float grade()
      // Returns student's average grade.
      pre: true
      post:
         #q1: grade() = #grade
}

class Course
{
   Course(String code, int capacity) throws NullPointerException,
      IllegalArgumentException
      // Creates a course object.
      pre:
         #p1: code != null
         #p2: capacity > 0
      post:
         #q1: #size = 0 && #code = code
         #q2: #capacity = capacity
      error:
         #e1: !#p1 => NullPointerException
         #e2: !#p2 => IllegalArgumentException

   String code()
      // Returns the course code.
      pre: true
      post:
         #q1: code() = #code

   int size()
      // Returns the number of students registered for the course.
      pre: true
      post:
         #q1: size() = #size

   int capacity()
      // Returns max capacity of the course.
      pre: true
      post:
         #q1: capacity() = #capacity

   boolean add(Student student)
      // Registers student into the course.
      pre:
         #p1: student != null && size() < capacity()
      entry:
         #oldsize: size()
```

```
      post:
         #q1: exists #s : #s in Course : #s = student
         #q2: size() = #oldsize + 1
         #q3: add(student) = true
      error:
         #e1: add(student) = false

   boolean drop(Student student)
      // Removes student from the course.
      pre:
         #p1: registered(student)
      entry:
         #oldsize: size()
      post:
         #q1: !registered(student)
         #q2: size() = #oldsize - 1
         #q3: drop(student) = true
      error:
         #e1: drop(student) = false

   boolean registered(Student student)
      // Determines whether the student has registered for the course.
      pre: true
      post:
         #q1: registered(student) =
              (exists #s : #s in Course : #s = student)

   List studentList()
      // Returns a List containing Students registered for the course.
      pre: true
      post:
         #q1: forall #s : #s in Course : studentList().contains(#s)
         #q2: forall #t : #t is Student &&
                 #t in studentList() : registered(#t)
}
```

The specification of the class methods are constructed in terms of five components: (1) primitive data (e.g., boolean, integer) and strings, (2) method arguments, (3) symbolic labels which are defined only within the class, (4) other methods of the class, and, possibly, (5) methods of other classes which are in turn specified in their own class specifications. Class variables (i.e., variables shared by all instances of the class) and instance variables are omitted in CCS because private variables are not accessible to other classes including the test program class, and non-private variables can be replaced by appropriate modifier and accessor methods (e.g., `Student.setGrade` and `Student.grade`).

2.2 Semantics of Equality

To be consistent with Java, CCS retains Java's semantics of the equality operator `==`:

- *Content Equality*: For variables x and y of primitive data type, x `==` y is true if and only if their values are identical.
- *Reference Equality*: For variables x and y of class type, x `==` y is true if and only if they refer to the same object.

The corresponding inequality operator is `!=`.

In Java, to determine whether the content of two objects such as strings are identical, an appropriate method such as `equals` must be invoked. This is inconvenient and inefficient for CCS because strings are widely used in many programs. To treat strings in the same manner as primitive data types, the operators `=` and `/=` are introduced in CCS to mean content equality and inequality respectively. These operators operate only on primitive data types and strings.

2.3 Pre, Post, and Error

The symbols `pre:`, `post:`, and `error:` mark the sections for precondition, postcondition, and error-handling assertion. The meaning of these assertions can be denoted, using the notation of Gries [5], as:

$$\begin{array}{c} \{P\}\ M\ \{Q\} \\ \{\neg P\}\ M\ \{E\}\ . \end{array} \tag{1}$$

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. Otherwise, M will terminate in a finite amount of time in a state satisfying the error-handling assertion E. Note that Gries's notation denotes *total correctness* which is slightly different from Hoare's notation of $P\ \{S\}\ Q$ which denotes *partial correctness* [6]. Class invariance is omitted in CCS because it can be verified from the semantics of the class methods.

The precondition section consists of one or more assertions, each of which is a logical expression. Except for the single precondition `true`, each assertion is assigned a label, e.g., `#p1` and `#p2` in `Student.Student`, which can be used as an abbreviation for the assertion. The labels can also be used in the test program to report the test results associated with the assertions. The label names can consist of a mixture of digits and letters and are defined only within a method specification, i.e., they have *method scope*. A method's precondition `pre` is a *conjunction* of the individual assertions. For example, the precondition of `Student.Student` is `#p1` $\wedge$ `#p2`. Multiple assertions can be written as a single conjunctive expression and assigned a single label (e.g., `#p1` in `Student.setGrade`).

Similarly, the postcondition and error-assertion sections consist of one or more assertions. The method's postcondition `post` and error-assertion `error` are also conjunctions of the individual assertions. In the error-assertion section, an exception name indicates that the named exception is raised. For error reporting, a software designer can decide whether a method should raise an exception (e.g., `Course.Course`) or return an error code (e.g., `Course.add`).

2.4 Entry State

The entry state section in the specification of method M, denoted by `entry:`, defines the state of the object when method M is invoked. Each statement of the form

`#`l`:` A`()`

refers to one aspect of the object's state where `#`l denotes a state label with *method scope* and

A() denotes the result of invoking the accessor method A of the same class. The statement simply means that, when the method M is called, the state of the object as revealed by A is given the name #l. If A's return type is a class, say C', instead of a primitive data type, then the class C' must define a `clone` method with *deep copy* semantics. This `clone` method is used in the test program to make an independent copy of the C' object to record the entry state, which is compared against the exit state after the specified method M is invoked. CCS adopts the state label notation instead of entry state operator (such as ADL's `@` and Eiffel's `old`) because it is simple and it conforms to the notations of the precondition, postcondition, and error-assertion sections.

2.5 Object State

State labels can also be defined in a method's postcondition section. A postcondition statement of the form

#l = a

defines a state label #l which denotes the content of the method argument a. For example, in the constructor `Student.Student`, the state label `#name` denotes the content of the argument `name`. These state labels refer to the state of the object *after* the method terminates. They are known as the *object state* labels because they denote information that is contained in the object. Therefore, object state labels are defined within the whole class and can be referred to in the specifications of other methods in the same class. That is, they have *class scope*.

It is important to recognize that object state labels only indicate *what* kind of information an object contains but not *how* it stores the information. By the principle of *information hiding* of the Object-Oriented paradigm, such state information is hidden in the object and is available only by invoking the *accessor methods* of the class. For example, the method `Student.name` returns the name of the student contained in a `Student` object. This functionality is specified in the postcondition of `Student.name`:

```
name() = #name
```

Object-state labels serve a very important role in CCS. To understand the importance, consider an alternative approach to specification in which object state labels are not used. In this case, one could write the postcondition of the constructor `Student.Student` as follows:

```
post:
    #q1: name = name()
    #q2: id = id()
```

This specification is, however, incomplete. It only says that the method arguments `name` and `id` are the same as those returned by the methods `Student.name` and `Student.id`. But, it does not imply an important aspect of object-orientation: that the information `name` and `id` are kept and carried in the object wherever it goes. Furthermore, it is now impossible to write the postcondition of the accessor methods `Student.name` and `Student.id`! The use of object state labels resolves both problems.

2.6 Aggregate Classes

An aggregate class describes objects that contain multiple instances of other objects. Any reasonable semantics of an aggregate class must include a method for adding new elements and a method for checking whether an element has previously been added. Thus, in CCS, an aggregate class conforms to the following *Aggregate Class Semantics*: An aggregate class C must define two methods that are consistent with the following semantics:

1. *Aggregation*: There exists a method M called the *aggregation method* with the following signature and postcondition:

   T M(C' a) E
       `post:`
           `#`l`: exists #`x` : #`x` in `A` : #`x` = `a

   where T is an optional return type, E is an optional `throws` expression, `#`l and `#`x are labels, A is either C or an object state label of C to which `#`x is bound, a is a method argument, and C' is the type of a.

2. *Membership*: There exists a method M called the *membership method* with the following signature and postcondition:

   `boolean `M(C' a)
       `post:`
           `#`l`: `M(a)` =`
               `(exists #`x` : #`x` in `A` : #`x` = `a`)`

For example, the `Course` class defines the method `add` which is consistent with the aggregation semantics and the method `registered` which is consistent with the membership semantics.

In the specifications for aggregation and membership methods, the existential quantifier is used to refer to one of the members of an aggregate. In addition to the existential quantifier, universal quantifier can also be used to refer to all the members of the aggregate. For example, the method `Course.studentList` returns a Java `List` containing `Student` objects. The method's postcondition specifies, using the `contains` method of the `List` class, that members of the `Course` object are also members of the returned `List`, and vice versa.

Quantified variables are used to refer to some or all the members of aggregate classes. Therefore, they always range over the members of aggregate classes or finite sets of integer values that map to the members of the aggregates. These aggregate classes include arrays, classes that implement the `Collection` interface or its subinterfaces, and user-defined aggregate classes. For example, in the postconditions `#q1` of the methods `add`, `registered`, and `studentList`, the quantified variable `#s` ranges over the members of `Course` class. On the other hand, the variable `#t` in `studentList` ranges over the members of `List` class, which is the result of calling `studentList`. The types of the quantified variables can be explicitly declared as in the statement `#t is Student`. Otherwise, the types are inferred from the specification.

If an aggregate class contains more than one aggregate (e.g., two different lists), then object state labels can be used to identify the individual aggregate. For example, the binding condition in the quantifications can be written as `#s in Course.#listA`, or simply `#s in #listA`, which differentiates itself from `#s in #listB`.

3 Generating Testing Programs

In general, a system that generates test programs from class specifications should consist of the following stages: (1) *test case selection* which chooses a set of inputs and outputs for testing, (2) *object construction* which produces the program codes for constructing test objects, (3) *test sequencing* which sorts the test cases, and (4) *test check generation* which produces the program codes that call the method under test and check the test results. Most research on specification-based testing has been focused on deriving test cases from specifications. For example, Richardson and Clarke [13] used symbolic execution techniques to derive test cases. Stocks and Carrington introduced the Test Template Framework for generating test cases from specification [16]. Chang et al. developed automated methods for generating test cases from ADL specifications [2, 3]. Most of these techniques can be adapted to CCS's Test Case Selector. In this article, we will focus mainly on object construction and test sequencing which are the most complex parts of CCS's test program generator.

3.1 Test Case Selector

The Test Case Selector chooses, from all possible test cases, a finite subset with a high probability of detecting most errors. Various selection strategies exist [9]. For instance, the *boundary-value analysis* strategy selects test cases that exercise a method at the boundary values of the input and output domains. Take the method `Course.add` as an example. `Course.add`'s precondition specifies that the number of students registered for the course should be smaller than the full capacity. Therefore, a boundary value exists at `size() = capacity()`. Two test cases can be formed: one test case requires a `Course` object with `size() = capacity() - 1` and the other requires a `Course` object with `size() = capacity()`. The first test case should result in satisfying `add`'s postcondition whereas the second should result in satisfying `add`'s error-assertion. The above test cases thus specify the input conditions and the expected results of the tests. The input conditions also specify the types of objects required for the tests.

3.2 Object Constructor

Given the types of objects as specified in a test case, CCS's Object Constructor identifies the methods that need to be invoked to construct the required objects. A simple object such as `Student` can be constructed by invoking just the method `Student.Student`. On the other, to construct a `Course` object that satisfies an input condition such as `size() = capacity()`, enough `Student` objects must also be constructed and added to the `Course` object. In general, to construct a sufficiently complex object which contains other objects as components, the constructors of the component objects must be invoked, and this process recurses.

The input condition R that a constructed object must satisfy is specified in a test case as a *conjunction* of comparison expressions:

$$R \equiv \left(\bigwedge_i A_i = L_i \right) \tag{2}$$

where each A_i is a possibly multidot method call such as $M_1\texttt{()}\,.\,M_2\texttt{()}\,.\,\ldots\,.\,M_n\texttt{()}$ and L_i is an expression in terms of the results of calling other accessor methods. For instance, in the above example test case, $R \equiv$ (`size() = capacity()`). In the following discussion, the notation $x\,.\,R$

will be used to denote the expression R in which method calls are applied to the object x, as in x.M(). Object construction can now be stated as follows:

> Given a constructor C of class C with the semantics $\{P\}\ C\ \{Q\}$, construct an object x of class C that satisfies the condition $x\,.\,R$.

Before applying the object construction algorithm, the class specifications are first processed to replace object state labels in class methods' postconditions by appropriate method calls. This step is necessary because the condition R contains method calls instead of labels. State label substitution is performed as follows:

- For each object state label #l that appears in a method M's postcondition but not in a quantification, find another method M' of the same class whose postcondition contains the assertion M'() = #l. Then, replace #l in M's postcondition by M'().
- For each existential quantification (exists #l : #l in A : L(#l)) where L is a logical expression, find another method M' of the same class whose postcondition contains the assertion M'() = (exists #l : #l in A : L(#l)). Then, replace the entire quantification by M'().

The algorithm for object construction can be most conveniently described in terms of *rules*. The algorithm finds a matching rule and performs the *actions* associated with the rule to generate the program codes for constructing objects. Although the rules are described separately, they can be combined to construct a complex object that satisfies a complex condition.

3.3 Rule 1: Simple Arguments

Condition: *The constructor's signature is* C(**a**), *and either* **a** *is an empty list or* $\mathbf{a} = \{a_1, ..., a_n\}$ *is a list of simple arguments,*[2] *i.e., arguments of primitive data types or of the class* String.

Action: Find a **u** (i.e., a list of expressions) that satisfies

$$P^{\mathbf{a}}_{\mathbf{u}} \wedge (x\,.\,Q^{\mathbf{a}}_{\mathbf{u}} \Rightarrow x\,.\,R)\ . \tag{3}$$

Then, output the following instruction

C x = new C(**u**);

This rule caters to the simplest case of object construction. The notation $P^{\mathbf{a}}_{\mathbf{u}}$ refers to the expression P with arguments in **a** replaced simultaneously by the expressions in **u**. It is straightforward to prove, by applying Gries's *Theorems of Procedure Call* [5], that the object x constructed using this rule (as well as the following rules) indeed satisfies the condition $x.R$.

The action of finding an appropriate **u** requires the use of Artificial Intelligence algorithms for *theorem proving* and *planning* [12]. Essentially, the planner identifies a possible **u** by matching the requirement R with the specification, and the theorem prover verifies whether the **u** identified satisfies the condition Eq. 3. The planner and theorem prover are required to support all the rules for object construction.

Example: Construct x of class Course such that x.$R \equiv$ x.capacity() = 10.

[2] Note that, without loss of generality, the arguments in **a** do not have to appear in exactly the same order as in the method's signature.

$$
\begin{aligned}
\mathbf{a} &= \{\texttt{code}, \texttt{capacity}\} \\
\mathbf{u} &= \{\texttt{"CS102"}, 10\} \\
P^{\mathbf{a}}_{\mathbf{u}} &\equiv (\texttt{"CS102" != null} \wedge \texttt{10 > 0}) \equiv \text{true} \\
\texttt{x.}Q^{\mathbf{a}}_{\mathbf{u}} &\equiv (\texttt{x.size() = 0} \wedge \texttt{x.code() = "CS102"} \wedge \texttt{x.capacity() = 10}) \Rightarrow \texttt{x.}R
\end{aligned}
$$

Note that in the above derivation, boolean, arithmetic, and string operations on numeric and string literals are evaluated. However, method calls such as `x.capacity()` are not evaluated and are simply regarded as symbols. The condition R specifies only the value of capacity. Therefore, an arbitrary string `"CS102"` can be generated for the argument `code`. Having identified $\mathbf{u}$, the following instruction is generated:

```
Course x = new Course("CS102", 10);
```

When this instruction is executed, a `Course` object with a capacity of 10 will be created.

3.4 Rule 2: Class Arguments

Condition: *The constructor's signature is* $C(\mathbf{a},\mathbf{b})$ *where* $\mathbf{a}$ *contains simple arguments and* $\mathbf{b}$ *contains arguments* b_i*,* $1 \leq i \leq n$*, of class type* C_i *other than* `String`*.*

Action: Introduce variables $v_1, \ldots, v_n$ to rewrite $x\,.\,R$ as $x\,.\,R' \wedge v_1\,.\,R_1 \wedge \cdots \wedge v_n\,.\,R_n$, where each $v_i\,.\,R_i$ is either true or a predicate on the results of calling the methods of class C_i. Note that this rewriting is possible because $x\,.\,R$ is a conjunction of comparison expressions (Eq. 2). Next, apply object construction rules recursively to construct each v_i subject to the condition $v_i\,.\,R_i$. Then, find a $\mathbf{u}$ that satisfies

$$P^{\mathbf{a},\mathbf{b}}_{\mathbf{u},\mathbf{v}} \wedge (x\,.\,Q^{\mathbf{a},\mathbf{b}}_{\mathbf{u},\mathbf{v}} \Rightarrow x\,.\,R)$$

where $\mathbf{v} = (v_1, \ldots, v_n)$. Finally, output the instruction

$C\ x$ `= new` $C(\mathbf{u},\mathbf{v})$`;`

following the instructions for creating the objects referred to by $v_1, \ldots, v_n$.

This rule is used to create an object x which requires other objects as arguments to its constructor method. The objects referred to by v_i must be created before x. The proper sequence for creating the objects is called the *construction sequence*. Except for Rule 1, all the other rules define appropriate construction sequences. Rule 2 is a straightforward extension of Rule 1, and the example is omitted.

3.5 Rule 3: Multidot Method Call

Condition: *The signature of the constructor is* $C(\mathbf{a},b)$*,* b *is of class type* C'*, and* $x\,.\,R \equiv L(x\,.\,M()\,.\,M_1()\,.\cdots.\,M_n())$ *where* L *is a logical expression of a multidot method call* $x\,.\,M()\,.\,M_1()\,.\cdots.\,M_n()$ *and the type of* $x\,.\,M()$ *is* C'*.*

Action: Introduce variables $v_1, \ldots, v_n$ to break $x\,.\,R$ into a conjunction of single-dot method calls:

$$x\,.\,R \equiv (x\,.\,M() = v_1 \wedge v_1\,.\,M_1() = v_2 \wedge \cdots \wedge v_{n-1}\,.\,M_{n-1}() = v_n \wedge L(v_n\,.\,M_n())$$

Next, apply object construction rules to construct each v_i in *reverse* order such that $v_i.R_i$ is satisfied. Then, find a $\mathbf{u}$ that satisfies

$$P^{\mathbf{a},b}_{\mathbf{u},v_1} \wedge (x\,.\,Q^{\mathbf{a},b}_{\mathbf{u},v_1} \Rightarrow x\,.\,R)\ .$$

Finally, output the instruction

C x = new C($\mathbf{u}$,v_1);

following the instructions for creating the objects referred to by $v_n, \ldots, v_1$.

Example: Consider the following partial specifications of the class `Author` and `Book`.

```
class Author
{
   Author(String name)
      post:
         #q1: #name = name

   String name()
      post:
         #q1: name() = #name
}

class Book
{
   Book(Author author, String title)
      post:
         #q1: #author = author && #title = title

   Author author()
      post:
         #q1: author() = #author

   String title()
      post:
         #q1: title() = #title
}
```

To construct a `Book` object `b` such that `b.author(). name() = "Gries"`, Rule 3 can be applied as follows:

b.R $\equiv$ (b.author().name() = "Gries")
$\equiv$ (b.R_b: b.author() = a $\wedge$ b.title() = "Programming") $\wedge$
(a.R_a: a.name() = "Gries")

Now, apply Rule 1 to construct `a` and `b` in that order:

```
Author a = new Author("Gries");
Book b = new Book(a, "Programming");
```

Rules 1 to 3 handle cases in which the condition R refers to method names that appear in the constructor's postcondition. The next rule handles the case in which the method names do not appear in the constructor's postcondition. In this case, other methods in the same class are sought.

3.6 Rule 4: Supporting Method Call

Condition: *The constructor's signature is* $C(\mathbf{a})$ *and the condition* $x.R$ *refers to a method name* M *that does not appear in* Q.

Action: Let the semantics of M be $\{P'\}\ M(\mathbf{b})\ \{Q'\}$ and suppose that either P' is an invariance of C or C's postcondition Q does not falsify P':

$$\{P'\}\ C\ \{P'\}\ \vee\ Q \Rightarrow P'\ . \tag{4}$$

Next, rewrite $x.R$ as $x.R_C \wedge x.R_M$ such that $x.R_M$ refers to only the method M and $x.R_C$ is an invariance of M:

$$\{R_c\}\ M\ \{R_c\}\ . \tag{5}$$

Then, find $\mathbf{u}, \mathbf{v}$ that satisfy

$$P^{\mathbf{a}}_{\mathbf{u}} \wedge (x.Q^{\mathbf{a}}_{\mathbf{u}} \Rightarrow x.R_C) \wedge P'^{\mathbf{b}}_{\mathbf{v}} \wedge (x.Q'^{\mathbf{b}}_{\mathbf{v}} \Rightarrow x.R_M) \tag{6}$$

Finally, output the instructions

$C\ x$ = new $C(\mathbf{u})$;
$x.M(\mathbf{v})$;

The invariances (Eq. 4,5) play crucial roles in Rule 4. They permit the construction of object x that satisfies only the R_C part of the condition R. They also ensure that the object construction does not falsify the precondition P' of the method M to be called next, and that calling M does not falsify R_C. Then, x can be updated to satisfy the whole R by calling M. The above rule can be generalized to handle cases involving multiple supporting methods.

Example: Construct `x` of class `Student` such that `x.grade() = 4.0`.

$\mathbf{a}$	=	{`name`, `id`}
M	=	`setGrade`
$\mathbf{b}$	=	{`grade`}
$\mathbf{u}$	=	{`"Bond"`, `"007"`}
$\mathbf{v}$	=	{`4.0`}

It is straightforward to verify that the above $\mathbf{u}$ and $\mathbf{v}$ satisfy the condition in Eq. 6. Therefore, the object `x` can be constructed using the following instructions:

```
Student x = new Student("Bond", "007");
x.setGrade(4.0);
```

3.7 Rule 5: Aggregate Class

Condition: *The constructor's signature is* $C(\mathbf{a})$. *The condition* $x.R$ *contains an assertion of the form* $x.m()$ = n *where* n *is an integer. The postcondition* Q *contains an assertion of the form* $m()$ = c *where* $c < n$. *Moreover, there exists a method* M *in class* C *with semantics* $\{P'\}\ M(b)\ \{Q'\}$ *such that* b *is of class type* C', M*'s specification contains an entry state* `#`m:

m(), and Q' contains an assertion of the form m() = #m + i where i in an integer (i.e., M is an aggregation method). Finally, there exists an integer k such that $c + ki = n$.

Action: Rewrite $x\,.R$ as $x\,.R_C \wedge x\,.R_M$ such that $x\,.R_M \equiv (x\,.m\texttt{()} \texttt{ = } n)$ and R_C is an invariance of M:

$$\{R_C\}\ M\ \{R_C\}\ .$$

Introduce variables $x_0, \ldots, x_k$ that are related by the following *sequence condition*

$$x_i\,.\texttt{\#}m = x_{i-1}\,.m\texttt{()}\ \wedge x_k = x,\ 1 \le i \le k$$

and the *invariance condition* $x_i\,.R_C = x\,.R_C$ for all i. Next, find $\mathbf{u}$, $v_1, \ldots, v_k$ that satisfy

$$P^{\mathbf{a}}_{\mathbf{u}} \wedge (x_0\,.Q^{\mathbf{a}}_{\mathbf{u}} \Rightarrow x_0\,.R_C)\ \wedge \bigwedge_{1 \le i \le k} x_{i-1}\,.P'^{b}_{v_i}\ \wedge (x_k\,.Q'^{b}_{v_k} \Rightarrow x_k\,.R_M)$$

Finally, output the instructions

C x = new C($\mathbf{u}$);
C' v_1 = new C'($\mathbf{w}_1$);
$x\,.M$(v_1);
...
C' v_k = new C'($\mathbf{w}_k$);
$x\,.M$(v_k);

where $\mathbf{w}_1, \ldots, \mathbf{w}_k$ are randomly generated arguments.

Example: Construct a `Course` object `x` that satisfies `x.capacity() = 2` and `x.size() = x.capacity()`.

$$\begin{array}{rcl}
M & = & \texttt{Course.add} \\
\texttt{x}.R_C & \equiv & \texttt{x.capacity() = 2} \\
\texttt{x}.R_M & \equiv & \texttt{x.size() = x.capacity()} \\
c, i, k & = & \texttt{0, 1, 2} \\
\mathbf{a} & = & \{\texttt{code, capacity}\} \\
\mathbf{u} & = & \{\texttt{"CS102", 2}\} \\
P^{\mathbf{a}}_{\mathbf{u}} & \equiv & (\texttt{"CS102" != null} \wedge \texttt{2 > 0})\ \equiv \text{true} \\
x_0\,.Q^{\mathbf{a}}_{\mathbf{u}} & \equiv & (x_0\texttt{.size() = 0} \wedge x_0\texttt{.code() = "CS102"} \wedge x_0\texttt{.capacity() = 2})\ \Rightarrow x_0\,.R_C
\end{array}$$

The implication at the last step is obtained from the condition that $x_i\,.R_C = \texttt{x}.R_C$. Next, let us derive the implication for $\texttt{x}.R_M$.

$$\begin{array}{rcl}
x_0\,.P'^{b}_{\texttt{v1}} & \equiv & \texttt{v1 != null} \wedge x_0\texttt{.size()} < x_0\texttt{.capacity()} \\
 & \equiv & \text{true} \wedge \texttt{0} < \texttt{2}\ \equiv \text{true (from invariance condition)} \\
x_1\,.Q'^{b}_{\texttt{v1}} & \equiv & x_1\texttt{.size() = } x_1\texttt{.\#oldsize + 1} \\
 & \equiv & x_1\texttt{.size() = } x_0\texttt{.size() + 1} \text{ (from sequence condition)} \\
 & \equiv & x_1\texttt{.size() = 1} \\
x_1\,.P'^{b}_{\texttt{v2}} & \equiv & \texttt{v2 != null} \wedge x_1\texttt{.size()} < x_1\texttt{.capacity()} \\
 & \equiv & \text{true} \wedge \texttt{1} < \texttt{2}\ \equiv \text{true} \\
x_2\,.Q'^{b}_{\texttt{v2}} & \equiv & x_2\texttt{.size() = } x_2\texttt{.\#oldsize + 1} \\
 & \equiv & x_2\texttt{.size() = } x_1\texttt{.size() + 1} \\
 & \equiv & (x_2\texttt{.size() = 2})\ \Rightarrow \texttt{x}.R_M
\end{array}$$

Thus, the `Course` object can be constructed as follows:

```
Course x = new Course("CS102", 2);
Student v1 = new Student("James", "006");
x.add(v1);
Student v2 = new Student("Bond", "007");
x.add(v2);
```

In this example, the `Student` objects' names and IDs are generated randomly. Nevertheless, Rule 5 can be combined with other rules to construct `Student` objects with specific names and IDs as specified in the condition $x.R$. In general, all the rules can be combined to construct complex objects that satisfy complex conditions. The combination of these 5 rules can handle most, if not all, object construction tasks specified by the Test Case Selector.

3.8 Test Sequencer

The Test Sequencer sorts the various test cases for a class into an appropriate order. This stage is not necessary for testing single function specifications such as the opened specification system ADL. 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 or error-assertion contains a call to method B, i.e., B(**b**).
3. A's postcondition or error-assertion contains a state label #l or an existentially quantified expression E, and B's postcondition 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 method and some accessor methods. For example, the constructor `Student` of class `Student` and the accessors `name` and `code` form a cyclic group while the modifier `setGrade` and the accessor `grade` form another 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., to be tested before) another test case T' if one of the following 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. This requirement is satisfied by Rule 5 since an object with more elements will need more instructions to construct.

The final step of Test Sequencer 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'.

3.9 Test Check Generator

Test Check Generation is the final stage of CCS, and it produces the program codes for testing a method and checking the results of the test. The program codes consist of two parts: *method calling* and *result checking*. The method calling code invokes the method under test, passing to it the required arguments, and receiving from it the result returned by the method. The object under test and the method arguments are constructed by the program codes that are generated by the Object Constructor. If an exception may be raised by the method under test, the method calling code also needs to catch the exception. Finally, the result checking code verifies whether the postcondition or the error-assertion is satisfied, and reports the checking result.

In testing a cyclic group of methods, the constructor or modifier method under test will be invoked by the method calling code, whereas the accessor methods will be called by the result checking code. If the result checking code reports that the test result is correct, then all the methods in the group pass the corresponding test case. Otherwise, one or more of the methods in the group fail the test case.

When the postcondition of a method under test contains universal quantifications, the Test Check Generator will produce a loop construct to check every element in the aggregate object. The loop construct iterates over objects that are either constructed and added by an aggregation method (e.g., `#q1` of `studentList`) or obtained from a Java `Iterator` supported by Java `Collection` classes such as `List` (e.g., `#q2` of `studentList`).

4 Conclusion

This paper proposed a Closed Class Specification (CCS) system for specifying the semantics of all the methods in a class. With CCS, all the class methods are specified in terms of other methods which are, in turn, specified in their own class specifications. Given such close specification of classes, it is possible for the CCS system to generate test programs fully automatically based on the specification only. Additional user-provided supporting functions such as ADL's *auxiliary*, *provide*, and *relinquish functions* are not required.

This paper also presented the framework of CCS's test program generator. In particular, the algorithms for test sequencing and automated object construction are discussed in detail. It can be formally proved that the objects constructed using the rules of the algorithm indeed satisfy the conditions specified in the test cases.

CCS's Object Constructor requires the support of a theorem prover and a planner. The theorem prover can be constructed from systems such as *Higher Order Logic* [4] and *Isabelle* [11]. The planner is currently under further research. The basic strategy is to guess possible $\mathbf{u}$ and $\mathbf{v}$ based on the input condition R, construction rules, and class specifications. The possible $\mathbf{u}$ and $\mathbf{v}$ are then submitted to the theorem prover to identify the combinations that satisfy the construction rules.

Continuing along the same line of research, we are also investigating methods for specifying and generating test programs for these aspects of software:

- Formatted string literals.
  For example, the US zip code consists of 5 digits followed by an optional code consisting of a hyphen and 4 additional digits. Such formatted string literals can be specified using a regular expression.

- Input/output text file formats.
  Text files are very simple and useful media for presenting inputs to and collecting outputs from a software. The format of a text file can be specified by an appropriate grammar.

- Graphical User Interfaces.
  Specification of GUIs is a very challenging research. Recent research on generating test cases for GUI [8] may provide insights into formal specification of GUI and automatic generation of GUI test programs.

References

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

[2] J. Chang and D. J. Richardson. ADLscope: An automated specification-based unit testing tool. In *Proc. Automated Software Engineering*, 1998.

[3] J. Chang, D. J. Richardson, and S. Sankar. Structural specification-based testing with ADL. In *Int. Symp. on Software Testing and Analysis*, 1996.

[4] L. Claesen and M. Gordon, editors. *Formal Methods in System Design*, volume 5, numbers 1/2. Kluwer Academic, 1994.

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

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

[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[8] 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.

[9] G. J. Myers. *The Art of Software Testing.* John Wiley, 1979.

[10] 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.

[11] L. C. Paulson. Introduction to isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[12] E. Rich and K. Knight. *Artificial Intelligence, 2nd Ed.* McGraw-Hill, 1991.

[13] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE. Trans. Software Engineering*, SE-11(12):1477–1490, 1985.

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

[15] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992.

[16] P. Stocks and D. Carrington. Test template framework: A specification-based testing case study. In *Proc. Int. Symp. Software Testing and Analysis*, pages 11–18, 1993.