An environment to support micro-incremental class development

Allen Parrish, David Cordes and Dennis Brown

The University of Alabama, Department of Computer Science, Box 870290, Tuscaloosa, AL 35487, USA

Incremental development and testing is widely cited as one advantage of the object-oriented paradigm. To date, most of the work in this area emphasizes incremental development at the "macro" level, i.e., at the application or class hierarchy levels. We believe that incremental development should also be exploited at the individual class level. In particular, classes may contain a variety of methods that place objects of the class into relatively complex states. By organizing and developing an individual class in an incremental fashion, one can (a) develop and test "partial classes" and (b) generate simple states for test objects prior to generating more complex ones. This process realizes two benefits: it simplifies debugging by reducing the size of the search space when tracking down defects, and it makes regression testing more efficient. This paper reports on a development environment designed to support "micro-incremental" class development. This environment integrates several different components and techniques. We discuss each component of the environment individually, and then illustrate the use of the environment in a case study.

1. Introduction

Incremental development and testing is widely cited as one of the many advantages in the object-oriented paradigm. However, most previous work emphasizes incremental development at the "macro" level, i.e., at the application or class hierarchy levels [Harrold *et al.* 1992]. Because individual classes are themselves often nontrivial, we believe that it is useful to define "micro-incremental" techniques to be used when developing and testing individual classes. Such techniques allow the developer to build and test subsets of class operations, rather than waiting until after the class is written to begin testing (as is the case with most existing class testing methods [Doong and Frankl 1991; Gannon *et al.* 1981; Hoffman and Brealey 1989; Jalote 1989; Zweben *et al.* 1992]).

In this paper, we present an environment to support micro-incremental class development. Our environment utilizes formal specifications, requiring that the class under test be algebraically specified [Guttag *et al.* 1985; Liskov and Guttag 1986]. The environment uses a formal specification language called LOBAS [Doong and Frankl 1991]. Our environment synthesizes three major components:

- a class generator,
- a development ordering generator, and
- a test oracle generator.

The *class generator* produces a partial C++ class from a LOBAS specification. While the entire implementation is not produced, a structural template for the class is generated. The *development ordering generator* produces a recommended ordering of the class's methods, presenting the order in which the methods should be developed in order to maximize the amount of testing which can be performed during development. Finally, the *test oracle generator* produces a driver program which executes a series of test cases on the class, providing results in terms of whether a given test case "passes" or "fails."

The remainder of this paper is organized as follows. Section 2 provides an overview of the basic framework used in class testing with algebraic specifications. Section 3 introduces the concept of micro-incremental development and presents some micro-incremental development techniques. Section 4 then introduces our environment and describes how the environment can be used to construct a debugged implementation for a formally specified class.

2. A basic framework

2.1. *Object-oriented programming and algebraic specifications*

Our model of object-oriented programming is similar to the model used in the object-oriented testing discussion in [Doong and Frankl 1991]. We use the term *class* to refer to the implementation of an abstract data type within an object-oriented language. An *object* is an instance of a class. A class implementation is defined in two parts: an *interface* consisting of a list of operations that can be performed on instances of the class, and a *body* consisting of the implementation of the operations. The implementation of an operation is sometimes called a *method*, and invoking an operation with respect to a given object is sometimes referred to as "sending a message" to the object, which responds to the message by executing its method. Additionally, every object has a *state*, which may be characterized by its history of method invocations.

Numerous class testing techniques have been developed for classes that have been formally specified using algebraic specification techniques. The ordered list class in Table 1 is specified using a language called LOBAS [Doong and Frankl 1991].

The specification requires that all methods be categorized as either a *constructor*, *transformer*, or *observer*. Observer methods return a type other than that of the class itself; the purpose of an observer method is to query the contents of an object (e.g., `IsEmpty`, `Length`, `Top`). Constructor and transformer methods both return objects of the class itself. The distinction between constructors and transformers is somewhat subtle and is explained below.

Table 1
Ordered integer list specification.

```
class OrderedList
  export
    Create, Add, Delete, Head, Tail, Member, Length
  constructor
    Create;
    Add(v: Integer);
  transformer
    Delete(v: Integer);
    Head;
    Tail;
  observer
    Member(v: Integer): Boolean;
    Length: Integer;
  var
    L: OrderedList;
    v, v1, v2: Integer;
  axiom
    (1) Create.Add(v).Head = v
    (2) Create.Member(v) = FALSE
    (3) Create.Add(v).Tail = Create
    (4) L.Add(v).Head = if v > L.Head then v else L.Head
    (5) L.Add(v).Tail = if v > L.Head then L else L.Tail.Add(v)
    (6) L.Add(v1).Member(v2) = (v1 = v2) or L.Member(v2)
    (7) L.Add(v).Length = L.Length + 1
    (8) Create.Delete(v) = Create
    (9) L.Add(v1).Delete(v2) = if v1 = v2 then L else L.Delete(v2).Add(v1)
  end
```

The axioms define the semantics of the class. Sequences of methods separated by dots are read left to right. For example, the sequence "`Create.Add(10).Tail`" means that `Create` is first executed; `Add(10)` is then executed on the result returned by `Create`, and `Tail` is executed on the result returned by `Add(10)`. (This interpretation is very much like the dot notation used in C++.) As an example, Axiom 3 states that executing the method `Create`, followed by `Add(v)`, followed by `Tail` produces an object that should be (in a correct implementation) equivalent to the list object produced simply by executing `Create`. Similarly, Axiom 7 says that given a list object `L`, when executing `Add(v)` on `L` followed by `Length`, the result returned by `Length` should be the same as the result returned by executing `Length` on `L` and adding 1 to the result.

Axioms may be used to derive one sequence from another. For example, the sequence "`Create. Add(10).Add(20).Add(30).Delete(20)`" may be reduced to `Create.Add(10).Add(30)` by repeated applications of Axiom 9 using the following sequence of derivations:

```
Create.Add(10).Add(20).Add(30).Delete(20)  ⇒  Create.Add(10).Add(20).
                                               Delete(20).Add(30)
                                           ⇒  Create.Add(10).Add(30).
```

By understanding derivations such as this one, it is possible to understand the difference between a constructor and transformer method. Constructors represent the "most primitive" sequences of methods that may be used to produce a particular object state. The axioms should be written in such a way that it is possible to rewrite sequences of constructors and transformers into sequences consisting of constructors only. In a correct implementation, it should be possible to produce states equivalent to every possible object state using only constructors.

We now consider the notion of correctness of a class implementation (e.g., in C++) with respect to a given LOBAS specification. We say that two method sequences are *specification equivalent* if the axioms may be used to derive one sequence from the other. For example, the sequences "`Create.Add(10).Add(20).Add(30).Delete(20)`" and "`Create.Add(10).Add(30)`" are specification equivalent as a result of the derivation shown above.

Given the above notion of specification equivalence, we now are able to provide an intuitive definition of the correctness of a class implementation. A class implementation is said to be *correct* with respect to a given algebraic specification *iff* for every pair $(S1, S2)$ of method sequences, $S1$ is specification equivalent to $S2$ *iff* the executions of $S1$ and $S2$ within the implementation return "equivalent" values. Two possibilities exist for the types of the values returned by those sequences:

- the values could be from a built-in type (e.g., integer, character, boolean, float), or
- the values could be objects of a user-defined class.

The notion of equivalence between values of built-in types is simple, and may be determined simply by using the built-in equality operator within the implementation language. The notion of equivalence between objects of user-defined classes is much more complicated. In [Doong and Frankl 1991], a formal, theoretical definition of object equivalence is given, known as *observational equivalence*. The basic idea is that two objects are observationally equivalent if those objects cannot be distinguished in client code. One possible heuristic for simulating observational equivalence is to implement an `Equal` method for the class, whose behavior is similar to that of an equality operator for built-in types. We discuss the use of such a heuristic in the next section.

2.2. *An algebraic specification-based test execution model*

In this section, we consider a model for test execution based on algebraic specifications. We assume a model similar to that of the ASTOOT system [Doong and Frankl 1991]. In this model, a test case is a triple of the form ($S1$, $S2$, tag), where

$S1$ and $S2$ are method sequences and the tag is either "equivalent" or "inequivalent," depending on whether or not the sequences are specification equivalent.

For the moment, we will assume that specific test cases have already been generated. (The problem of generating test cases will be addressed later in the paper.) Given that test cases have already been created, we identify the following basic testing process:

1. For each test case, execute $S1$ and $S2$ and determine whether or not they return either "equivalent" values (the mechanism for assessing this equivalence is discussed below).
2. If the test case is to be considered "OK" (unrevealing of a defect), then either (a) the test case tag is "equivalent," and values returned by $S1$ and $S2$ are actually "equivalent" or (b) the test case tag is "inequivalent" and the values returned by $S1$ and $S2$ are actually "inequivalent." Otherwise, the test case reveals a defect.

We now discuss the mechanism for assessing the equivalence of two sequences when executed as part of a given implementation. Provided that $S1$ and $S2$ return values of a built-in type, a test driver can be constructed that automatically compares return values and prints a message as to whether or not the test case reveals a defect. In particular, consider the test case *(*$B1$, $B2$, *equivalent)*, where $B1$ and $B2$ are as follows:

($B1$) `Create.Add(10).Add(20).Length`
($B2$) `Create.Add(10).Add(20).Add(30).Tail.Length`

It is easy to verify that $B1$ and $B2$ are specification equivalent, which implies that the "equivalent" tag is valid for this (hypothetical) test case. Since `Length` returns an integer (which is a built-in type), a test for correctness is simply a test to determine whether the integers returned in both sequences are identical. Thus, we have the following test driver psuedocode:

```
List a, b;
a.Create(); b.Create();
a.Add(10); a.Add(20);
b.Add(10); b.Add(20); b.Add(30); b.Tail();
if (a.length() == b.length())
      output "Test case correct";
else
      output "Test case error";
```

The condition `a.length() == b.length()` uses the built-in integer equality operator to determine whether or not the integer results of the two sequences are identical.

On the other hand, if the two sequences both return objects, then a user-defined `Equal` method must be used to measure equivalence, as was discussed in the previous

section. Unlike the built-in equality operator (which should not contain defects), this approach is not fool-proof, in that the user-defined `Equal` method may itself contain errors. However, the testing process will (ideally) reveal any such errors, e.g., if a test case fails and there is no defect in the methods that are explicitly a part of the test, the defect must be in `Equal`.

The test driver constructed in a situation where the sequences return objects is conceptually the same as the test driver above. In particular, consider the test case *(C1, C2, equivalent)*, where $C1$ and $C2$ are simple modifications of $B1$ and $B2$ above (`length` has been removed):

(C1) `Create.Add(10).Add(20)`
(C2) `Create.Add(10).Add(20).Add(30).Tail`

$C1$ and $C2$ are indeed specification equivalent, given that both lists are ordered. Thus, the "`equivalent`" tag in this test case is valid. Unlike $B1$ and $B2$, both $C1$ and $C2$ return lists, i.e., objects of a user-defined class. The driver will need to utilize a user-defined list equality method to compare the stack results. In C++, the == operator may be overloaded for the list class. After performing such overloading for class `List`, the resulting test driver appears below:

```
List a, b;
a.Create(); b.Create();
a.Add(10); a.Add(20);
b.Add(10); b.Add(20); b.Add(30); b.Tail();
if (a == b)        //  == is defined for class List
      output "Test case correct";
else
      output "Test case error";
```

Currently, our method for generating test cases (discussed in section 3) only generates test cases with `equivalent` tags. Because of this, we will drop the tag notation in the remainder of the paper, and express test cases as simple pairs of method sequences. As is noted in [Doong and Frankl 1991], this eliminates some test cases that may be useful in revealing defects. (The resulting model is actually similar in power to the DAISTS model [Gannon *et al.* 1981].) However, there are a number of useful test cases still available, and our method is not intended to be the final testing method used when validating a class, but is simply a tool to use during development to assist in defect elimination.

3. Micro-incremental development

Existing class testing techniques assume that the class under test has already been constructed. A completed class normally involves a plurality of methods. In a

good design, we would not expect individual methods to be complex [McGregor and Sykes 1992]. However, many methods are non-trivial, and methods (whether simple or complex) may interact to produce objects whose states are relatively complex. Moreover, under a typical development model, the number of failures that occur initially during testing is relatively high. Tracking down the sources of these failures among a large number of methods that are interacting in potentially subtle ways can prove to be extremely difficult.

We believe that one of the keys to effective debugging support is in the idea of *state simplification*. When producing class objects for testing and debugging purposes, objects with simple states are easier to reason about than are objects with complex states. Not all defects are revealed by simple state objects; however, many defects revealed by complex state objects are also revealed by simple state objects. Thus, we are advocating an organization of the class testing process, where testing progresses from (initially) examining simple state objects to (eventually) examining objects with states that are more complex.

We identify two possible "state complexity" dimensions around which to organize an incremental class testing process:

- *Structural complexity:* The number of *different methods* that are invoked to produce a given state.
- *Behavioral complexity:* The number of *method invocations* producing a given state.

For example, consider the state produced by invoking "`Create.Add(10).Add(20).Delete`." The structural complexity of this state is 3, since there are three methods invoked to produce this state (`Create`, `Add`, and `Delete`). The behavioral complexity of this state is 4 (the length of the sequence).

The goal of our incremental testing process can now be more precisely stated: *To locate each class defect using the simplest possible state in terms of both its structural and behavioral complexity*. Our process may be characterized as follows:

1. Execute a series of test cases, where the objects produced by the test are of progressively increasing complexity.
2. When a test case in the series fails, cease testing until the defect is repaired.
3. Repairing the defect involves modifying one or more methods. Repeat any previously executed test cases that involve modified methods.
4. Continue testing in this manner until the next defect is revealed (at which point steps 2 and 3 are repeated) or testing is completed without revealing more defects.

This process results in two tangible benefits:

- Reasoning about the source of a defect may be conducted in the simplest possible context.

- Regression testing resulting from defect repair is simplified. In particular, if a defect is located and repaired in the simplest possible state, then the number of methods that have to be re-executed when retesting simpler states is correspondingly reduced.

We now address the actual mechanisms used to achieve our goal of producing test cases in increasing order of structural and behavioral complexity. First, to ensure a progression of increasing structural complexity during test case generation, we need to be able to test iteratively during the development process. That is, we want to be able to test subsets of methods as those methods are developed. With our (algebraic) specification-based testing model, the order in which methods are developed directly impacts the amount of testing that can take place prior to class completion. Thus, a technique is needed to generate the optimal ordering in which methods should be developed for a given class, if we wish to maximize testing during development.

Second, to ensure test case generation is performed in increasing order of behavioral complexity, we need to be able to progressively increase the number of method invocations involved in test cases. Our method for generating test cases directly addresses this.

The remainder of this section is organized as follows. Section 3.1 discusses a method for generating an optimal ordering in which methods should be developed, thus addressing the structural complexity dimension. Section 3.2 discusses a method for generating test cases in increasing order of behavioral complexity, thus addressing the behavioral complexity dimension. Finally, section 3.3 discusses the integrating these techniques to define an overall testing methodology.

3.1. Selecting a development ordering

A *development ordering* is an arrangement of class methods that identifies the order in which the methods should be developed. As discussed above, the selection of such an ordering is determined using the goal of permitting as much testing as possible as early as possible. This allows testing to take place as (small) subsets of methods are written, as opposed to deferring testing until after all methods have been coded.

Not all development orderings are equally good in terms of permitting periodic testing. To illustrate, consider the specification for a simple stack class in Table 2.

Now consider the development ordering `Create` → `Pop` → `Top` → `Push`. In this case, *no* testing is permitted until all methods are complete. In order to construct test cases using a given axiom, all of the methods in that axiom must be present; however, `Push` is present in every axiom. Thus, it is impossible to construct any executable test cases for the class until after all methods are completed (since `Push` is constructed last). `Create` → `Push` → `Pop` → `Top` would be a much better development ordering, in that there are opportunities for testing earlier (and more often) in the development process. With this ordering, the developer could:

1. Develop `Create` and `Push` and then construct test cases using Axiom 1.

Table 2
Integer stack specification.

```
class Stack
  export
    Create, Push, Pop, Top
  constructor
    Create;
    Push(v: Integer);
  transformer
    Pop;
  observer
    Top: Integer;
  var
    s: Stack;
    v: Integer;
  axiom
    (1) not (s.Push(v) = Create)
    (2) s.Push(v).Pop = s
    (3) s.Push(v).Top = v
```

2. Test the methods Create and Push.
3. Develop Pop and then construct test cases for Create, Push and Pop using Axioms 1 and 2.
4. Test the method Pop, and perform additional tests on Create and Push. (Additional testing on Create and Push might be necessary because of the new states generated by including Pop in the test cases.)
5. Develop Top and then construct test cases for all four methods using all three axioms.
6. Test all methods in the class.

More generally, given a development ordering $M_1, M_2, M_3, \ldots, M_n$, we attempt to maximize:

1. the number of M_is for which new testing is possible (i.e., testing that was not possible to perform after constructing M_{i-1}), and
2. for each M_i after which testing is permitted, the number of axioms available for constructing test cases.

Item (1) deals with maximizing the number of opportunities in which testing may occur, while item (2) deals with maximizing the amount of testing that can take place at a given opportunity. We call an opportunity to test a *test point*. A *test point at position* i in a development ordering means that it is possible to construct and execute test cases after developing the ith method that were not executable after the $(i-1)$th method in the ordering. In the stack example, with the ordering Create $\rightarrow$ Push $\rightarrow$ Pop $\rightarrow$ Top, there are test points after Push, Pop and Top; each method introduces

new testing opportunities, as additional axioms may be used in the production of test cases. There is no test point after `Create` because no test cases can be produced from the axioms with just the `Create` method.

By developing methods in an order which maximizes both the number of test points and the amount of testing which may occur at a given test point, we are able to design the testing process so that testing can take place on subsets of the methods. This allows us to test in the context of reduced structural complexity. As methods are added, we increase the structural complexity of our test cases in a gradual, orderly fashion.

We now provide a simple greedy-style algorithm to select development orderings in this fashion. To define this algorithm, we first have the notion of an *axiom-method table*. This table is simply a dynamic table of those methods that appear in each axiom that have not yet been selected for the recommended development ordering. That is, position (1) in the table initially contains those methods that appear in Axiom 1; position (2) initially contains the methods that appear in Axiom 2, etc.

In order to conduct any testing at all using our model, it is necessary to have two special methods: a method that returns objects "from scratch" and an equality operator. We call methods that return objects without requiring an input object a *base constructor*.[1] We assume that a base constructor and an equality operator are both developed first, and thus omit them from the axiom-method table. To avoid confusion, we call the base constructor `Create` and the equality operator `Equal`, although in our target implementation language, it is possible to implement both as overloaded operators (`EQUAL` as ==, and `Create` as a C++ constructor). We consider this issue further in section 4.

To illustrate the axiom-method table, we reconsider the ordered list specification from Table 1. In this case, the only base constructor is `Create`. Although no equality operator is specified, one must still be developed to test this class using our test execution model. However, neither `Create` nor the equality operator will appear in the axiom-method table for this class. Each position in the table contains the methods that appear in the the axiom corresponding to that position (minus `Create`). The initial value of the axiom-method table appears in Table 3(a).

The second important concept that we must introduce is the notion of a *method table*. The method table contains a record for each method with three attributes: (1) the method name; (2) the method's *test point contribution* and; (3) the method's *axiom contribution*. A method's test point contribution is simply "yes" or "no," depending on whether a test point is created if that method is chosen. A method's axiom contribution is the sum of the contributions that the method makes to each axiom in which it is used. The notion of "contribution to an axiom" is explained below. The method table is also dynamic, in that once a method is selected for a development ordering,

[1] The term *base* constructor is used to distinguish such methods from the LOBAS notion of a constructor, which is slightly different. Base constructors are very much like the C++ notion of a constructor.

Table 3(a)
Ordered list axiom-method table.

Axiom	Methods
(1)	`Add, Head`
(2)	`Member`
(3)	`Add, Tail`
(4)	`Add, Head`
(5)	`Add, Head, Tail`
(6)	`Add, Member`
(7)	`Add, Length`
(8)	`Delete`
(9)	`Add, Delete`

Table 3(b)
Ordered list method table.

Method	TC	AC
`Add`	no	$3.3\overline{3}$
`Delete`	yes	1.50
`Head`	no	$1.3\overline{3}$
`Tail`	no	$0.8\overline{3}$
`Member`	yes	1.50
`Length`	no	0.50

the method table is updated to reflect changes that have occurred with respect to the contribution of the other methods.

The initial value of the method table for the ordered list class is shown in Table 3(b).

As discussed above, test point contributions (TC) are obtained by noting which methods would result in test points, if chosen as the next method in the development ordering. `Member` and `Delete` both result in a test point so the test point contribution for both of these methods is "yes". On the other hand, `Add`, `Head`, `Tail` and `Length` would not result in a test point if chosen first, so their test point contributions are "no." Axiom contributions (AC) are based on the cumulative contributions of that method to each axiom. For example, `Head` (which appears in 3 axioms) receives a $1.3\overline{3}$, based on the sum of:

- 0.5 from Axiom 1, because it is one of two methods present (and represents one-half of the methods remaining to be implemented in order to test using Axiom 1),
- 0.5 from Axiom 4, because it is one of two methods present (and represents one-half of the methods remaining to be implemented in order to test using Axiom 4),
- $0.3\overline{3}$ from Axiom 5, because it is one of three methods present (and represents one-third of the methods remaining to be implemented in order to test using Axiom 5).

We then have two selection criteria for the next method:

- First, identify methods that make test point contributions, thus permitting an opportunity to test after that method is constructed. For the above method table, this rule implies we should only consider `Delete` and `Member` as the next method.
- Second, for the methods identified in the previous step, choose the next method based on the highest axiom contribution, thus maximizing the amount of testing that can be conducted at a given opportunity. If two methods both define a new test point and both contribute equally, an arbitrary choice can be made. In this

example, both `Delete` and `Member` have the same axiom contribution (1.5), and so we can make an arbitrary choice between the two.

We choose `Delete` for the first method (arbitrarily). We then revise the axiom-method table and the method table, as is shown below. The method chosen (in this case `Delete`) is removed from the axiom-method table. In addition, the method table is revised as follows: (a) `Delete` is removed from the table and; (b) the axiom and test point contributions are revised accordingly. The new axiom-method and method tables are then shown in Table 4(a).

Of the entries in the revised method table that make a test point contribution, `Add` has the highest axiom contribution. Consequently, we take `Add` to be the next method; the revised tables are illustrated in Table 4(b).

At this point, `Head` has the highest axiom contribution of those methods making a test point contribution, so it is chosen next. Repeatedly applying this same procedure to the remaining methods, the final ordering (including `Create` and `Equal`) is `Create` → `Equal` → `Delete` → `Add` → `Head` → `Tail` → `Member` → `Length`.

This process focuses primarily on subsets of the methods in which it is still possible to conduct testing, thus allowing us to initially test in the context of structurally simple states involving a limited number of methods. This testing using simple states may then gradually transition to testing that involves more complex states.

Table 4

(a) Tables after selecting method `Delete`.

Axiom-Method Table		Method Table		
Axiom	Methods	Method	TC	AC
(1)	`Add, Head`	`Add`	yes	$3.8\overline{3}$
(2)	`Member`	`Head`	no	$0.8\overline{3}$
(3)	`Add, Tail`	`Tail`	no	$0.8\overline{3}$
(4)	`Add, Head`	`Member`	yes	1.50
(5)	`Add, Head, Tail`	`Length`	no	0.50
(6)	`Add, Member`			
(7)	`Add, Length`			
(8)				
(9)	`Add`			

(b) Tables after selecting method `Add`.

Axiom-Method Table		Method Table		
Axiom	Methods	Method	TC	AC
(1)	`Head`	`Head`	yes	2.50
(2)	`Member`	`Tail`	yes	1.50
(3)	`Tail`	`Member`	yes	2.00
(4)	`Head`	`Length`	yes	1.00
(5)	`Head, Tail`			
(6)	`Member`			
(7)	`Length`			
(8)				
(9)				

3.2. *Generating test cases*

Our method for generating test cases involves generating "object states", and then using those states as inputs to axioms from the specification. Thus, we divide the test case generation problem into two parts: the problem of generating object states and then the problem of actual test case construction, using the object states as inputs.

In [Parrish *et al.* 1994] we first described a relatively simple scheme for generating object states. The basic idea is to identify those methods that have an effect on the object state. In LOBAS terms, these are the constructor and transformer methods. In the case of the ordered list class, those methods are `Create` and `Add` (constructors) and `Delete`, `Head` and `Tail` (transformers). It is useful to further subdivide this group of methods in a slightly different way. Recall from section 3.1 that a method is a *base constructor* if it produces an object from scratch, i.e., without taking an object as input. To develop our state generation approach, we will need to segregate base constructors from other constructor/transformer methods. We call the other constructor/transformer methods *modifiers*, in that they take an object as input and modify the object to produce a result.

It is then possible to systematically generate a series of increasingly complex object states. These states may be used as input to the axioms during testing, as is described later in this section. To organize the state generation process, we utilize what we call a *state tree*. Each node in the tree represents a particular state. The character strings at each node represent the method sequences used to generate the state (e.g., `CAD` refers to the state generated by executing `Create`, followed by `Add`, followed by `Delete`). A partial state tree for our list class is shown in Figure 1.

The states in the tree are generated as follows. First, the base constructor method is executed to generate the initial (root) state. Next, the states at Level 1 are generated, from left to right, by invoking the methods at each node. Following the generation of Level 1 states, the states at each successive level are generated in similar fashion.

It should be noted that other information is needed to generate states than simply the order of method invocations as defined within the state tree. In particular, the sequence `CAD` indicates that `Create`, `Add` and `Delete` are to be executed on some defined list object. In C++ syntax, the object is written in front of the method in an invocation (e.g., the `L` in `L.Add(v)`). However, each of these methods (`Create`,

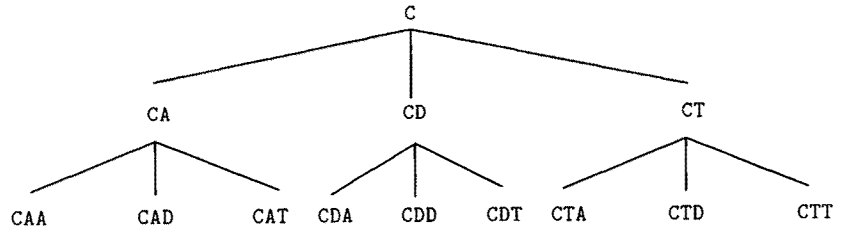

Figure 1. State tree for ordered list class.

`Add`, `Delete`, and `Tail`) may take other "secondary" parameters. For example, `Add` and `Delete` take a specific element value that is to be added or deleted (e.g., the `v` in `L.Add(v)`). In general, there are three types of secondary parameters that can be expected in a given implementation:

- Parameters of built-in types, such as integer, character, real, boolean, *etc.*
- Parameters of a class other than the class under test.
- Parameters of the class under test. In the case of a "binary" method (e.g., concatenate, which takes two string objects), one of the two parameters may be viewed as the primary parameter and the other one as the secondary parameter (e.g., `s1.concat(s2)`, where `s2` is the secondary parameter). Note that in C++, operator overloading may eliminate the syntactic distinction between primary and secondary parameters (e.g., `s1 + s2`).

Our current implementation builds a state tree for the primary object parameter, and randomly produces values for secondary parameters. For built-in types, this essentially involves using a random number generator; for class objects, this involves generating random constructor and transformer method sequences. Although this introduces an element of randomness to an otherwise systematic process, we can at least ensure that at least one parameter of the class under test is being systematically manipulated by method invocation sequences of steadily increasing complexity.

States produced by this process are used as parameters to the axioms to generate test cases. Specifically, if there is a variable in an axiom of the type of the class under test, then states from the state tree are used to produce values for the variable in constructing test cases. For example, consider Axiom 7 from the ordered list specification in Table 1:

```
L.Add(v).Length = L.Length + 1
```

Producing test cases from this axiom involves generating a succession of values for `L` and `v`. Using our state tree generation strategy, we can have the following sequence of values that may be substituted for `L` (the first three levels of the state tree):

```
L1   Create
L2   Create.Add(10)
L3   Create.Delete(6540)
L4   Create.Tail
L5   Create.Add(10).Add(564)
L6   Create.Add(10).Delete(430)
L7   Create.Add(10).Tail
L8   Create.Delete(6540).Add(543)
L9   Create.Delete(6540).Delete(54987)
L10  Create.Delete(6540).Tail
```

```
L11  Create.Tail.Add(237)
L12  Create.Tail.Delete(10)
L13  Create.Tail.Tail
```

Note that once a value is randomly generated for a parameter, that value is retained for the parameter within a given subtree. For example, the first invocation of `Add` in every node of the subtree rooted at `CA` receives the parameter value "10" (i.e., in L2, L5, L6 and L7 in the table above).

The object values from the above table may then be substituted back into the axiom to create a series of test cases:

```
(L1.Add(103).Length, L1.Length + 1)
(L2.Add(653).Length, L2.Length + 1)
(L3.Add(9403).Length,L3.Length + 1)
(L4.Add(120).Length, L4.Length + 1)
(L5.Add(457).Length, L5.Length + 1)
etc.
```

The progression of states from the state tree is substituted for the leftmost list variable to appear in each axiom. Substitutions for all other variables are made at random. However, by using the progression of states generated by our state tree to make at least one substitution, we normally meet our goal of ensuring that test cases steadily increase in complexity. We acknowledge, however, that due to the randomness of the remaining substitutions, there will be some exceptions.

The same substitution process takes place for the remaining axioms to generate additional test cases. We note that, for branching axioms (i.e., axioms that contain an `if` or some type of conjunction with multiple exclusive parts), there is no guarantee of testing all branches of the axiom; testing both branches will normally require that the random values generated for various variables be appropriate to ensure such coverage. For example, consider the axiom:

```
L.Add(v).Head = if v > L.Head then v else L.Head
```

Testing both branches for the right hand side requires that values be generated for `v` and `L` such that `v > L.Head` and `v <= L.Head`. It is possible that (eventually) such values will be generated simply by generating a large number of test cases. However, since this cannot be guaranteed, our environment (as discussed in section 4) produces information regarding which branches of the branching axioms have been tested. In this way, the developer can monitor whether additional testing is needed.

A final question which surfaces involves determining when to stop testing. As we discuss in section 4.1, the developer must specify two things when using our environment to generate test cases: (a) the number of levels in the state tree and (b) which base constructors to use in generating state trees (as it is possible to generate a distinct state tree for every base constructor). However, the testing technique does

not provide any guidance regarding how to make these two choices; effectively, there is no stopping criterion built into the testing technique. We would expect that a necessary condition for stopping would be coverage of all branching axioms, as discussed above. This is consistent with the approach used by DAISTS [Gannon *et al.* 1981]. However, we certainly do not feel that this is a sufficient condition.

On the other hand, we are not suggesting that this method should be the only testing method applied to a class, where a decision would have to be made based solely on our results regarding whether or not the class had been tested adequately. Instead, this method is a development and debugging-oriented approach that makes an initial pass at testing classes in an organized fashion during development. Once the class is completely developed, a more rigorous testing method with a definite completion point could still be applied to the completed class (such as the technique from [Doong and Frankl 1991]). Further investigation is needed to determine how extensively testing should be conducted during development, as well as to determine ways of refining or modifying our test generation strategy. One of the advantages of our overall framework is that it is possible to modify the test generation strategy (or the development ordering generation strategy) and still retain the overall benefit of incremental development within a general framework.

3.3. An overall development and testing methodology

Our method for generating test cases (in section 3.2) may be combined with our method for generating development orderings (in section 3.1) to create an overall testing approach in the context of development. In particular, suppose that you have the following development ordering for the ordered list class as defined in section 3.1 (`Equal` → `Create` → `Delete` → `Add` → `Head` → `Tail` → `Member` → `Length`). A general development and testing process would be as follows:

1. Develop `Equal`, `Create` and `Delete` and generate test cases using Axiom 8 and state tree generation restricted to these two methods. In particular, the state tree would be linear, with `Delete` the only modifier method in the tree (the generated states would be `C`, `CD`, `CDD`, `CDDD`, `CDDDD`, etc.).

2. Develop `Add` and generate test cases using Axioms 8 and 9. In this case, the state tree would be binary, with `Add` and `Delete` as modifiers and `Create` as the base constructor. States are produced such as `C`, `CA`, `CD`, `CAA`, `CAD`, `CDA`, and `CDD` (the first three levels in the state tree in breadth-first order).

3. Develop `Head` and generate test cases using Axioms 1, 4, 8 and 9. At this point, we have three modifier methods (`Add`, `Delete` and `Head`), and so the state tree is ternary. States produced include `C`, `CA`, `CD`, `CH`, `CAA`, `CAD`, `CAH`, `CDA`, `CDD`, `CDH`, `CHA`, `CHD`, `CHH` (again, the first three levels in breadth-first order).

4. Develop `Tail` and generate test cases using Axioms 1, 3, 4, 5, 8 and 9. Now we have four modifier methods (`Add`, `Delete`, `Head`, and `Tail`), and the state tree is extended appropriately.

5. Develop `Member` and generate test cases using Axioms 1, 2, 3, 4, 5, 6, 8 and 9. Although we are able to conduct testing using additional axioms beyond the previous step, the state tree does not change from before, since `Member` is not a modifier method.
6. Develop `Length` and generate test cases using all of the axioms. Again, the state tree remains as before; however, we are able to generate additional test cases via the use of Axiom 7.

Thus, we achieve our first goal of incremental structural complexity by "testing as we go," before all of the methods are written, limiting the number of methods that can appear in test cases to just those which have been written. We achieve our second goal of incremental behavioral complexity by utilizing a progression of states from a state tree as object inputs to the axioms in constructing test cases.

4. An integrated development environment

4.1. Environment functionality and architecture

An integrated environment which supports the class development paradigm described in section 3 is presently under construction; we currently have a working prototype that we are using for experimental purposes. In our environment, LOBAS specifications are used as input to generate three separate items:

- a development ordering for the class,
- a "template" for a (partial) class (using C++), and
- a test driver program for a (partial) class.

The notion of a development ordering was discussed extensively in section 3. A *test driver program* is just a program that executes the test cases generated for the class using the techniques of section 3.2. The notion of a *class template* is discussed below.

Let us first consider the sketch of the environment's X-Windows interface found in Figure 2. Figure 2 assumes that we have loaded the stack specification from Table 2 (section 3.1). We can generate the development ordering by clicking on the `Ordering` button, thus resulting in the configuration shown in Figure 3.

Now we are in a position to begin development. We begin by clicking the buttons containing the names of the methods that we wish to develop from the top of the screen. Clicking on `Create` and `Push` indicates that we wish to develop these two methods first (as the ordering suggests). Once we have clicked on both methods, Axiom 1 shifts from the "Inactive" to "Active" windows. Figure 4 shows the configuration of the development manager at this point. This configuration indicates visually that we have a "partial specification" consisting of Axiom 1 now available for test generation purposes. Thus, once `Create` and `Push` are actually written, we can conduct testing using Axiom 1 to generate test cases.

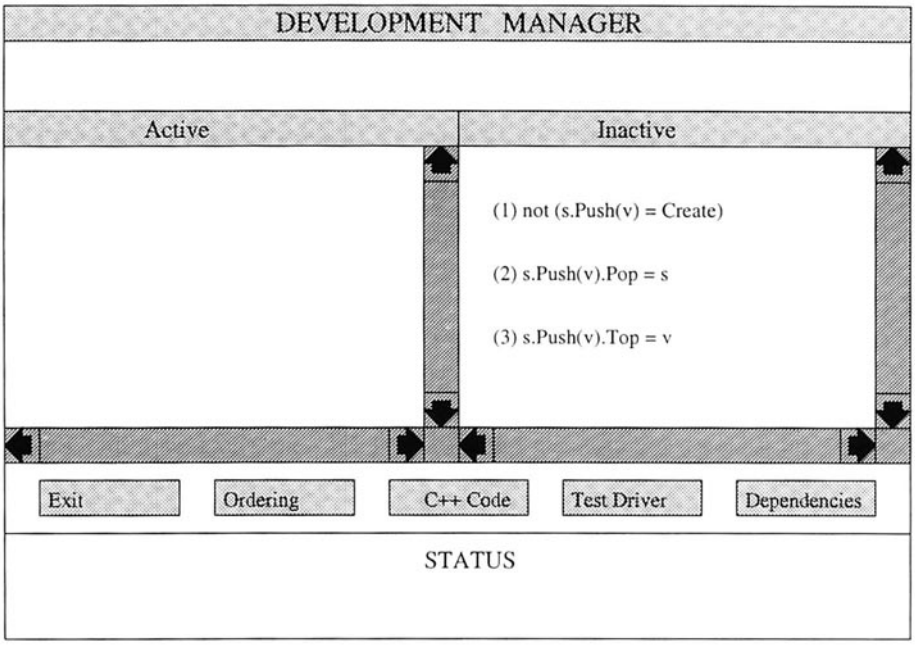

Figure 2. Initial configuration.

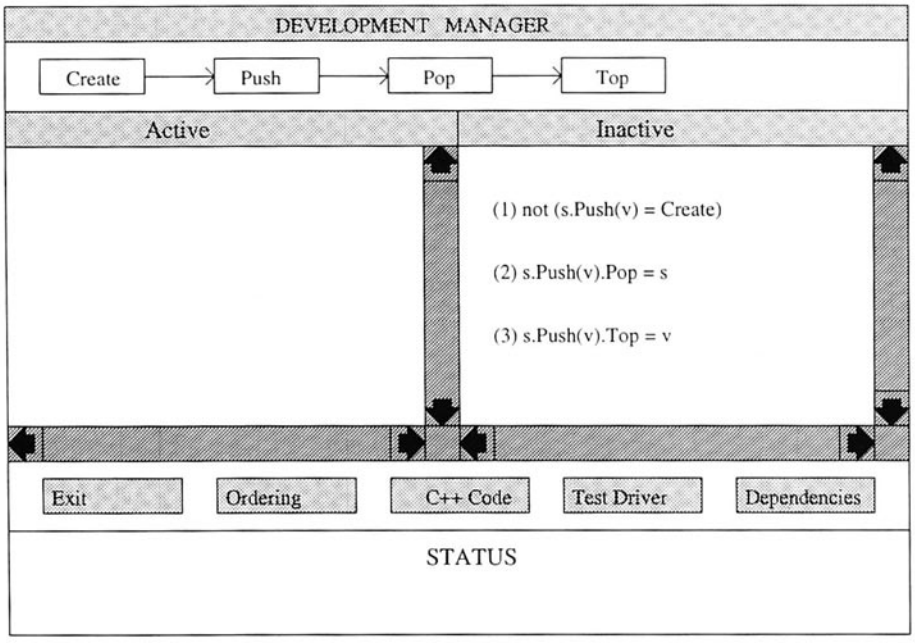

Figure 3. After generating development ordering.

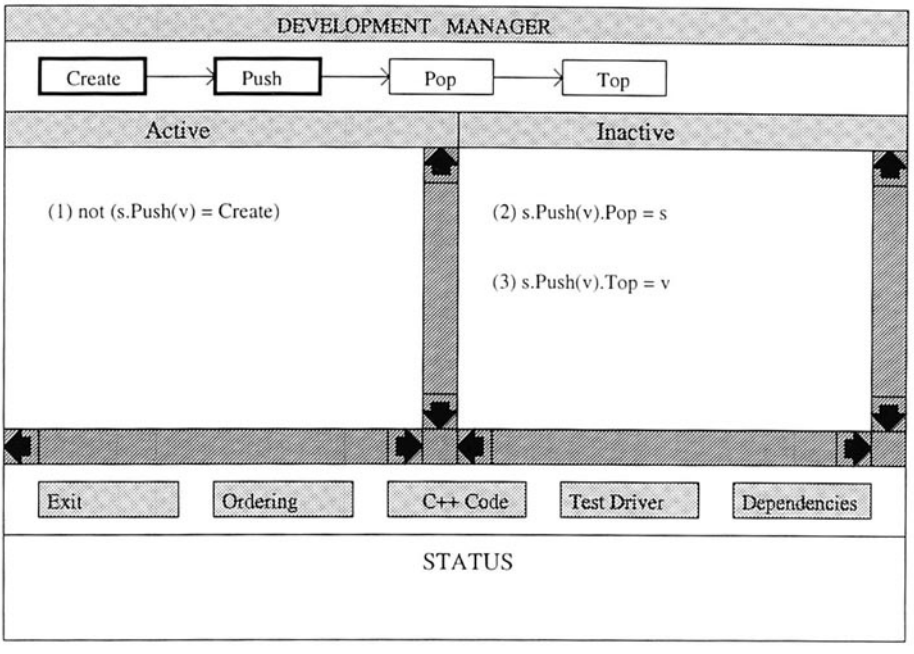

Figure 4. After selecting `Create` and `Push`.

To assist in writing `Create` and `Push`, we first click on the `C++ code` button. This generates a "template" for the C++ class, based on information found in the LOBAS specification, as follows:

```
class Stack{
private:
   // Fill in implementation here
public:
   void Create(void);
   void Push(int v);
};

void Stack::Create(void){
   // Fill in implementation here
}

void Stack::Push(int v){
   // Fill in implementation here
}
```

Our current implementation simply takes the method names from the LOBAS specification and generates the C++ template accordingly. However, it should also be possible to specify mappings from LOBAS names to C++ names. For example, it would be appropriate to implement `Create` as a C++ constructor rather than as a method called `Create`. We are currently working on an enhancement to the prototype that provides the facility to specify such mappings.

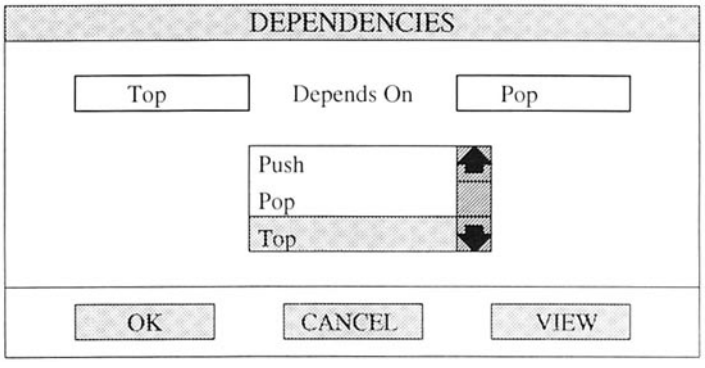

Figure 5. Defining implementation dependencies.

At this point, the developer must provide an implementation for the actual data representation for class `Stack`, and then implement `Create` and `Push`. Once bodies are written for these methods, testing may begin. Clicking on `Test Driver` generates another window, where the user is asked to specify two things:

- the base constructors for which it is desired to generate state trees (effectively, how many different state trees should be generated), and
- the number of levels in each state generation tree that should be generated.

Once this information is provided, a test driver is generated. This driver executes the test cases produced by combining the state trees with the axioms, as described in section 3.2. For every test case, the driver executes the two sequences found in the test case, and then compares the results of those sequences to determine whether or not the test case passes (as discussed in earlier sections). Information regarding any test cases that fail is shown in a separate window. Also, whenever a test case is executed that involves a branching axiom, information regarding which branch of the axiom is actually tested is also output to the separate window.

Once `Create` and `Push` have been debugged, the tester can click on additional methods. Clicking on `Pop` causes Axiom 2 to become active, meaning that more testing can now be performed. Clicking on `C++ Code` extends the existing class with (a) a prototype for the `Pop` method and (b) a function definition for `Pop` with an empty body. The developer may fill in the contents of `Pop`, click on `Test Driver`, and continue testing as before.

The only remaining button not discussed so far is the `Dependencies` button. Clicking on `Dependencies` allows the developer to define implementation dependencies among the methods. For example, suppose the developer wishes to implement `Top` by invoking `Pop` on a local copy of the stack and returning the value that is removed. Consequently, there is a dependency in the implementation (not present within the specification) that needs to be specified. Figure 5 contains the window where implementation dependencies are defined.

By specifying this dependency, the development ordering generator will not allow an ordering to be constructed that contains `Top` before `Pop`. The actual implementation simply will not set `Top`'s test point contribution to "yes" until `Pop` has been selected. It should be pointed out that such an implementation may cause the development ordering algorithm to fail, because there may be a case where the only method making a test point contribution depends on some method that hasn't been developed yet. In such a case, the developer is told of the conflict and the reasons for it, and is given the opportunity to resolve the conflict by either changing the specification or removing the implementation dependency. Failure to resolve the conflict simply means that testing will have to be deferred until the class is completed.

4.2. *Case study: Building a list implementation*

As an example illustrating the utility of this environment, we developed a C++ implementation of the ordered list class. For demonstration purposes, we elected to generate test cases utilizing an eight-level state tree. As already discussed, the development ordering generated by the environment (based on our algorithm from section 3.1) was `Create` → `Equal` → `Delete` → `Add` → `Head` → `Tail` → `Member` → `Length`.

Our development was completed by one of the authors in a relatively non-contrived setting. In particular, the author was handed the specification and was given the development environment, with no preparation regarding expected outcomes. The mistakes made were not deliberately contrived, but were in fact real. A log of the development process appears as follows:

1. The initial step was to build the first methods in the ordering: `Create`, `Equal` (implemented as `==`) and `Delete`. No testing was possible until after `Delete` was completed, as `Delete` constituted the first test point in the ordering. At this point, we were able to generate test cases using Axiom 8 and an eight-level state tree (i.e., `C`, `CD`, `CDD`, `CDDD`, etc., up to eight levels). All test cases passed.

2. The method `Add` was developed, allowing test cases to be generated using Axioms 8 and 9 and an eight-level state tree produced by `Create` (base constructor) and `Delete` and `Add` (modifiers). All test cases passed.

3. The method `Head` was developed, allowing us to generate test cases with Axioms 1, 4, 8 and 9 (and an eight-level state tree). Executing the test driver resulted in a segmentation fault, which upon subsequent examination, was found to be caused by an uninitialized pointer. The defect was repaired.

4. The test cases from step (3) were repeated. The result was that one of the Axiom 4-based test cases, (`Create.Add(27).Add(540).Head, 540`), failed. Because of our incremental approach, it can be inferred that the defect is in one of three methods: `Create`, `Add` or `Head` (the only methods involved in the construction

and execution of the test case).[2] The cause turned out to be the `Add` method, in that it did not insert elements in the proper order, but rather simply maintained the list in random order. This defect was repaired.

5. The test cases from step (3) were again repeated. Again, a similar test case (based on Axiom 4) to the one from step (4) failed, although with different random number integer inputs. Similar inferences can therefore be made regarding the source of the defect. The cause was that `Add` now inserted the elements into the list in the wrong order (ascending, rather than descending).

6. The test cases from step (3) were again repeated, again with a failure on a test case like the one in step (4) above. The problem this time was that the special case of inserting an element at the head of the list was not handled properly. The defect was repaired and the tests repeated with no failures.

7. At this point, the `Tail` method was added, allowing test cases to be generated with Axioms 3 and 5 (in addition to Axioms 1, 4, 8 and 9, which were already available). Running the test cases revealed a failure in an Axiom 5-based test case where the integer parameter to `Add` equaled the item already at the head of the list. (This test case is too complex to specify here, given that generating a parameter to `Add` that was already at the head of the list was a random event and did not occur early in the sequence of test cases.) This test case was created and executed using `Create`, `Add`, `Head`, `Tail` and `Equal`. Thus, `Delete` was not examined for the defect, nor were `Member` and `Length` (which are not even written yet). It was eventually concluded that the defect was the result of an incorrect assumption that if an existing element was to be added again to the list it should not be duplicated. The axioms, however, imply that elements may be duplicated. Thus, `Add` was modified accordingly.

8. Although `Delete` was not examined directly in the previous step, we realized at this point that a change to `Delete` was necessary. The original implementation of `Delete` removed all elements in the list of a given value, while the revised implementation only removes one such element if there are multiple copies. After making these changes and re-running the test cases from the previous step, no failures were observed.

9. The method `Member` was added next, allowing test cases to be produced using all of the axioms except Axiom 7. Test cases were then produced using an eight-level state tree and Axioms 1–6 and 8–9. No failures were observed when executing any of the test cases.

[2] Note that for some test cases, the class equality operator is also involved in executing the test case; however, the built-in integer equality operator was used in this test case because the two sides of the test case had integer results.

10. Finally, the method `Length` was added, allowing test cases using all of the axioms to be generated. Testing was performed using all axioms and all states in an eight-level state tree. No failures were observed.

The important observation to make about this process is that, cumulatively speaking, there were actually six separate defects ultimately observed in this class. Although two of the defects could not have arisen simultaneously (the `Add` defects where the list was first unordered and then ordered in the wrong direction), five of the defects could have co-existed within an initial version of the class. Isolating and eliminating defects (even relatively straightforward defects such as these) would have been extremely difficult in such a context. Our incremental approach provides the advantage of "divide-and-conquer" when locating and removing defects, which is not available if testing is deferred until the class is complete.

To appreciate the difference between incremental and "post-development" class testing, consider the fact that there were test cases involving two different axioms that failed in the above testing scenario (Axioms 4 and 5). This allowed us to restrict our attention to analyzing only those axioms, and only the methods involved in producing test cases with those axioms. With post-development testing, the various defects in the `Add` method would have caused failures to occur in most of the axioms where `Add` is present (possibly Axioms 4, 5, 6, 7, and 9). In particular, the failure to add an element properly to the head of the list would have resulted in definite failures in test cases based on Axioms 6 and 7 (the axioms which specify `Member` and `Length`). Thus, the developer might have wasted time looking for defects in `Member` and `Length` when none existed. Instead, we eliminated the `Add` defect before test cases based on Axioms 6 and 7 were even constructed and executed.

5. Conclusion

In this paper, we have identified an incremental development process for object-oriented class development. We call this type of process "micro-incremental" to reflect the idea that it is incremental with respect to developing the internals of a single class. We believe that this type of process is both natural and important to the development of classes, which may contain methods that interact in complex ways. Identifying an incremental process of this type simplifies regression testing and makes the defect removal process more efficient.

Future work will involve the development of additional development ordering and test case generation strategies as alternatives to our current strategies. We acknowledge that there are limitations to the current strategies, and intend to consider alternatives that fit within this paradigm. Our development environment is constructed to provide an overall framework where other testing strategies may be substituted without sacrificing any of the advantages of incremental development. A primary goal of our future work is to experimentally evaluate a number of different incremental development strategies over a wide variety of classes; our environment provides a framework for such evaluation.

References

Doong, R. and P. Frankl (1991), "Case Studies on Testing Object-Oriented Programs," In *Proceedings of the Fourth Symposium on Software Testing, Analysis and Verification*, pp. 165–177.

Gannon, J., P. McMullin and R. Hamlet (1981), "Data Abstraction, Implementation, Specification and Testing," *ACM Transactions on Programming Languages and Systems 3*, 211–223.

Guttag, J., J. Horning and J. Wing (1985), "The Larch Family of Specification Languages," *IEEE Software 4*, 5, 24–36.

Harrold, M., J. McGregor and K. Fitzpatrick (1992), "Incremental Testing of Object-Oriented Class Structures," In *Proceedings of the International Conference on Software Engineering*.

Hoffman, D. and C. Brealey (1989), "Module Test Case Generation," *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, pp. 97–102.

Jalote, P. (1989), "Testing the Completeness of Specifications," *IEEE Transactions on Software Engineering 15*, 526–531.

Liskov, B. and J. Guttag (1986), *Abstraction and Specification in Program Development*, McGraw-Hill, New York.

McGregor, J. and D. Sykes (1992), *Object-Oriented Software Development: Engineering Software for Reuse*, Van Nostrand Reinhold, 1992.

Parrish, A., D. Cordes and H. Dyal (1995), "Incremental Testing of Algebraically Specified Object-Oriented Software Modules," Department of Computer Science Technical Report, The University of Alabama.

Parrish, A., D. Cordes and M. Govindarajan (1994), "Systematic Defect Removal from Object-Oriented Software Modules," In *Proceedings of the Seventh International Software Quality Week Conference*, San Francisco.

Zweben, S., W. Heym and J. Kimmich (1992), "Systematic Testing of Data Abstractions Based on Software Specifications," *Journal of Software Testing, Verification and Reliability 1*, 4, 39–55.