Tools for Testing Object-Oriented Programs [1]

Phyllis G. Frankl

Roong-Ko Doong

Department of Computer Science
Polytechnic University
333 Jay Street
Brooklyn, NY 11201
(718-260-3870)
e-mail: pfrankl@polyof.poly.edu

Keywords: software testing, object-oriented, abstract data type

[1]This research was supported in part by NSF grant CCR-8810287 and by the New York State Science and Technology Foundation Center for Advanced Technology program.

1 Introduction

In recent years, object-oriented programming has emerged as a new software development paradigm which has many attractive software engineering features. While a considerable amount of research has focussed on how to design, implement and re-use object-oriented software, relatively little work has addressed the question of, "how should object-oriented software be tested?" This paper presents a new approach to testing object-oriented programs and describes experimental prototypes of two tools based on this approach, a test generation tool and a test execution tool.

Most research on software testing has implicitly assumed that the units under test are control abstractions, such as procedures and functions. In object-oriented programs, the emphasis is shifted from control abstractions to data abstractions. The program units one wishes to test are *classes* which implement abstract data types, rather than individual procedures or functions. Rather than focussing on input and output, we would like to test whether particular sequences of messages put an object into the right state.

In section 2 of this paper, we describe a new model of the testing process which makes this shift of emphasis. This model leads in a natural way to development of tools for test generation and test execution. In section 3 we describe experimental prototypes of such tools, and illustrate their use with examples. Our model is inspired by the theory of algebraic specifications of abstract data types, and our test generation tool uses an algebraic specification as input; however the test execution tool does not require the availability of a formal specification. Section 4 discusses some of the issues raised by inheritance, and section 5 compares our approach to related work, and points out several directions for future work. In the remainder of this introduction, we review relevant background material on software testing, object-oriented programming languages, and algebraic specifications of abstract data types.

1.1 Background on Software Testing

It is well known that testing is one of the most time-consuming parts of the software development process. Increased automation of the testing process could lead to significant saving of time, thus allowing for more thorough testing. Three aspects of the testing process which could potentially be at least partially automated are test data generation, test execution, test checking. Our approach to testing object-oriented programs involves all three of these areas.

One area for potential automation is in the construction of test drivers. Many testing methods can be applied to individual subprograms. When the program unit being tested is a whole program, the inputs and outputs are usually sets of files. When the unit being tested is a procedure or function the inputs and outputs may include values of parameters and of global variables, as well as values read from and written to files. In order to test a procedure, it is necessary to build a *driver* program which initializes global variables and actual parameters to the appropriate values, then calls the procedure, then outputs final values of relevant globals and parameters. It can be quite cumbersome to initialize the inputs and check the values of the outputs. It is particularly unwieldy if, as is often the case in object-oriented programming, the parameters have complicated types. The model described below for testing object-oriented programs circumvents this problem.

Another problem which arises in testing software is the *oracle problem* – after running P on a test case, it is necessary to check whether the result agrees with the specification. This is often a non-trivial problem, for example if there is a great deal of output, or if it is difficult to calculate the correct value [WEY82]. Our testing method uses a novel approach which allows the correctness of test cases to be checked automatically by the test execution system.

1.2 Overview of Object-Oriented Programming

Object-oriented languages support *abstract data types* and *inheritance*. An abstract data type is an entity which encapsulates data and the operations for manipulating that data. In object-oriented programming, the programmer writes *class* definitions, which are implementations of abstract data types. An *object* is an instance of a class; it can be created dynamically by the instantiation operation, usually called "new" or "create". A language supports *inheritance* if classes are organized into a directed acyclic graph in which definitions are shared, reflecting common behavior of objects of related classes.

A class consists of an *interface* which lists the operations which can be performed on objects of that class and a *body* which implements those operations. The state of an object is stored in *instance variables* (sometimes called *attributes*), which are static variables, local to the object. An object's operations are sometimes called *methods*.

In object-oriented programs, computation is performed by sending *messages* to objects. A message invokes one of the object's methods, perhaps with some arguments. The invoked method may then modify the state of its object and/or send messages to other objects. When a method completes execution, it returns control (and in some cases returns a result) to the sender of the message.

The inheritance mechanism of object-oriented languages facilitates the development of new classes which share some aspects of the behavior of old ones. A descendent (subclass) C_d of a class C inherits the instance variables and methods of C. C_d may extend the behavior of C by adding additional instance variables and methods, or specialize C by redefining some of C's methods to provide alternative implementations (or some combination thereof).

A dynamic binding mechanism is used to associate methods with objects. It is legal to assign an object of class C_d to a variable of class C (but not vice-versa). After doing so, a message sent to this object will invoke C_d's method. For example, consider a class polygon with subclasses triangle and square, each of which redefines polygon's perimeter method. Assigning an object of class square to a variable of class polygon, then sending the perimeter message will invoke square's perimeter method. This provides for polymorphism.

Some examples of object-oriented languages include Smalltalk, C++, and Eiffel [GOL83, STR86, MEY88]. While Ada and Modula-2 are not, strictly speaking, object-oriented languages, they do provide support for data abstraction; thus, some of the ideas discussed here are relevant to them. See [MEY88] for a good overview of the object-oriented approach and cogent arguments that object-oriented programs are easier to design, implement, maintain, and re-use than traditional programs.

1.3 Algebraic Specifications of Abstract Data Types

Before we can talk about testing a class, we must have some concept of what it means for the class to be correct. Thus, we must have some means, formal or informal, of specifying the abstract data type which the class is intended to implement. We now give an overview of a formal specification technique known as algebraic specification, upon which some of our testing techniques are based.

Algebraic specification techniques [GOG78, GUT77, LIS75] describe a data abstraction by describing the interaction between the operations, without reference to the underlying representation. An algebraic specification of an ADT has a syntactic part and a semantic part. The syntactic part is a list of function names, and their signatures (the types which they take as input and produce as output). The syntactic specification is similar to the interface specification part of a class definition in an object oriented language.

The semantic part of the specification consists of a list of axioms describing the relation between the functions. Some specification techniques allow for a list of preconditions describing the domains of the functions, while others allow functions to return error values indicating that a function has been applied to an element outside of its domain.

In the specification of a type T, a function `f` is called an *observer* if it's output is a type other than T, and either a *constructor* or *transformer* if T is an output of `f`. Constructors and transformers change the state of the ADT, while observers allow the "outside world" to query the ADT about its current state. The distinction between constructors and transformers depends on the semantic part of the specification: `f` is a transformer if it is possible to eliminate `f` from any sequence by applying rewrite rules, and is a constructor otherwise.

```
CLASS SET_OF_INT EXPORT
  Create, Insert, Delete, Empty, Member
CONSTRUCTOR
  Create;
    -- -- Create an empty set
  Insert(x:  INTEGER)
    -- -- Insert x into set
TRANSFORMER
  Delete(x:  INTEGER)
    -- -- Delete x from set
OBSERVER
  Member (x:  INTEGER): BOOLEAN;
    -- -- Is x in the set?
  Empty:  BOOLEAN
    -- -- Is set empty?
VAR
  old:  SET_OF_INT;
  a, b:  INTEGER
AXIOM
    -- -- 1
  Create.Empty → TRUE;
    -- -- 2
  old.Insert(a).Empty → FALSE;
    -- -- 3
  Create.Member(a) → FALSE;
    -- -- 4
  old.Insert(a).Member(b) → if a = b then TRUE
            else old.Member(b)
    -- -- 5
  Create.Delete → Create;
    -- -- 6
  old.Insert(a).Delete(b) → if a = b then old.Delete(b)
            else old.Delete(b).Insert(a)
END
```

Figure 1: Specification of SET_OF_INT

For example, the specification in Figure 1 describes a set of integers ADT. The specification is written in LOBAS (Language for Object Based Algebraic Specification) [DOO90], a language which we are developing in tandem with our study of testing object-oriented programs. The syntax of LOBAS is similar to that of object-oriented programming languages: the type being specified is an implicit input to each operation except `Create`; sequences of operations read from left to right with dots as separators. This specification says that type SET_OF_INT has a constant `Create` of type SET_OF_INT (the empty set), and operations

`Insert`: SET_OF_INT $\times$ Integer $\longrightarrow$ SET_OF_INT

`Delete`: SET_OF_INT $\longrightarrow$ SET_OF_INT

`Member`: SET_OF_INT $\times$ Integer $\longrightarrow$ Boolean

`Empty`: SET_OF_INT $\longrightarrow$ Boolean.

Axioms 1,2,3, and 5 say that `Create` is empty, the set obtained by inserting any item in any set is not empty, no element is a member of `Create`, and the result of deleting any element from `Create` is `Create`, respectively. Axiom 4 says that if an element `a` is inserted into a set `old`, then `b` is a member of the resulting set if and only if either `b` = `a` or `b` is a member of `old`. Axiom 6 says that the result of inserting `a` then deleting `b` from a set `old` is the same as the result of deleting `b` from `old`, if `a` = `b`; if `a` $\neq$ `b` it is the same as deleting `b` from `old` then inserting `a`. Note that axiom 6 says that if duplicate copies of element `b` have been inserted into `old`, they are all deleted. Thus, the fact that duplicates are present can never affect the result of a sequence of operations ending in an observer, so this ADT behaves like a set, rather than a bag.

Our notion of test cases for data abstractions is inspired by the formal semantics of algebraic specifications developed by [GOG78, GUT77], which involves modelling a specification by a heterogeneous word algebra. Roughly speaking, two sequences `S1` and `S2` of operations returning a value of type T are equivalent if, for any observer `O`, we can use the axioms as rewrite rules, to transform `S1.O` to `S2.O`. For example, `Create.Insert(5).Insert(3).Delete(5)` is equivalent to `Create.Insert(3)` because we can apply axiom 6 twice to give

`Create.Insert(5).`<u>`Insert(3).Delete(5)`</u> $\longrightarrow$

`Create.`<u>`Insert(5).Delete(5)`</u>`.Insert(3)` $\longrightarrow$

`Create.Insert(3)`.

It follows that for any observer `O` we can rewrite

`Create.Insert(5).Insert(3).Delete(5).O` to

`Create.Insert(3).O`.

Intuitively, each equivalence class of sequences returning type T corresponds to a different state of the ADT. For example, the two above sequences represent the state in which the set has a single element, 3.

A specification of type T is sufficiently complete if any sequence of operations ending in an observer of type T' can be simplified into a constant of type T'. A specification is inconsistent if there is a sequence of operations ending in an observer which can be simplified into two different elements, by applying rewrites in different orders; it is consistent otherwise. Throughout this paper, we will only consider specifications which are consistent and sufficiently complete.

We next consider what it means for a class C to be a correct implementation of an ADT T. The idea is that C is correct if and only if we can map operations of T to operations in C in a way which preserves signatures and preserves the way operations transform states. We will say that objects O_1 and O_2 of class C are *observationally equivalent* if and only if for any sequence S of operations of C ending in an observer, $O_1.S$ is equivalent to $O_2.S$ (in the output type of S). Thus, O_1 is observationally equivalent to O_2 if and only if it is impossible to distinguish O_1 from O_2 using the operations of C. Two observationally equivalent objects are in the same "abstract state", even though the details of their representations may be different.

DEFINITION:
A class C is a correct implementation of an abstract data type T if and only if for every pair (S_1, S_2) of sequences of operations in T,

> S_1 is equivalent to S_2 if and only if the corresponding sequences of operations in C put objects of class C into observationally equivalent states.

2 An Object-Oriented Notion of Test Case

Traditional testing strategies are based on viewing a specification and a program as functions mapping an input space to an output space. Let $\mathcal{D}$ be the domain of a specification $\mathcal{S}$. Program $\mathcal{P}$ is a *correct* implementation of $\mathcal{S}$ if and only if $\mathcal{P}$ and $\mathcal{S}$ compute the same function on $\mathcal{D}$, i.e. iff $(\forall d \in \mathcal{D})(\mathcal{P}(d) = \mathcal{S}(d))$. In order to test $\mathcal{P}$, one chooses a subset $\mathcal{T} \subseteq \mathcal{D}$, runs program $\mathcal{P}$ on each element of $\mathcal{T}$, and compares the output to the specified value. If $\mathcal{P}(t) \neq \mathcal{S}(t)$ for some $t \in \mathcal{T}$, then an error has been found. On the other hand, if the program agrees with the specification on each element of a well-chosen test set $\mathcal{T}$, our confidence that the program is correct should increase. Of course nothing less than exhaustive testing ($\mathcal{T} = \mathcal{D}$) can *guarantee* correctness.

In one of the early papers on specification of data abstractions [LIS75], Liskov and Zilles pointed out that it is possible to specify a data abstraction by specifying the intended input-output behavior for each of its operations individually, but doing so is usually cumbersome and leads to overspecification of the underlying representation of the data. Similarly, it is possible to test a class by testing each of it's methods individually, but doing so may necessitate the construction of complicated drivers and output checking mechanisms. For example, a test case for the push operation in a stack would consist of stack and an item and the output would be another stack. Thus the driver would have to initialize the input stack and checking the output would entail examining the output stack to see if it is the correct result. Moreover, testing individual operations separately shifts the focus of testing away from the essence of the data abstraction: the interaction between operations resulting in the equivalence of various sequences of messages.

Our approach to testing classes is based on a different kind of test case, which, we believe, is more appropriate for testing data abstractions. It is inspired by the definition of *correctness* given above. A test case for C will consist of a pair of sequences of operations (messages), along with a tag indicating whether or not they should be equivalent, according to the specification. For example,

`(Create.Insert(5).Insert(3).Delete(5), Create.Insert(3), equivalent)`, and

`(Create.Insert(5).Insert(3).Delete(5), Create.Insert(5), unequivalent)`

are test cases for SET_OF_INT.

To execute test cases, we build a driver class which instantiates two objects O_1 and O_2 of class C. For each test case, the driver sends the first sequence of messages to O_1 and the second sequence to O_2, invokes a user-supplied operation, EQN, to check whether O_1 and O_2 are in the same state, and compares the result of this with the tag. We will discuss the EQN operation shortly. If, according to the specification, the two sequences should be equivalent, but EQN returns false, then an error is reported. Similarly, If the specification says the sequences should be unequivalent, but EQN returns true, an error is reported.

Notice that in this model, exhaustive testing corresponds to sending all possible pairs of sequences (perhaps using transitivity of equivalence to eliminate redundancies). If a class is exhaustively tested without exposing an error, then the class is guaranteed to be correct, assuming that the EQN operation is correct. This is the analog of the fact that exhaustive testing of a traditional program guarantees correctness, provided that no mistakes are made in checking whether the outputs meet the specification. Of course, as in the case of traditional program testing, exhaustive testing is completely impractical in all but the most trivial situations.

We now discuss the EQN operation. Ideally, the EQN operation in class C should check whether two objects O_1 and O_2 of class C are observationally equivalent, that is, it should check whether any sequence of messages ending in an observer yields the same result when sent to O_1 as when sent to O_2. Since it is clearly impossible to send every such message to the objects, in practice EQN will approximate a check for observational equivalence.

It is often quite easy to produce a recursive version of EQN from the specification of the ADT which C is intended to implement. For example a specification of a queue ADT gives rise to the EQN function,

```
EQN(q1,q2):
    if Empty(q1) then
        if Empty(q2) then return(TRUE)
        else return (FALSE)
    else if Empty(q2) then return(FALSE)
    else return(EQN(q1.Remove, q2.Remove))
```

which can easily be translated into the implementation language. Note that this is actually only an approximation of true observational equivalence because it neglects the possible effects of "building up" `q1` and `q2`, then removing elements. Thus, it might say that two objects are equivalent when they are not. Also, since EQN calls Empty and Remove, an error in one of these operations may propagate to EQN, causing it to "mask out" detection of the error.

Another approach to developing the EQN function is to write it at the "implementation level". In this approach, EQN is based on detailed knowledge of how data is represented and manipulated in the class body. For example, knowing that the queue is represented as a linked list, one can traverse the two lists comparing the elements. If sufficient attention is paid to the details of the representation, EQN can implement observational equivalence exactly. On the other hand, it is possible that the same misconceptions which lead to implementation errors in C's other methods may lead to errors in EQN.

It is also sometimes possible to use a very coarse approximation of observational equivalence as the EQN function. For example, we might consider two queues to be equivalent if they have the same number of elements, or if they have the same front element. As in the case of our specification-level EQN function, this version of EQN may consider two unequivalent objects to be equivalent.

Naturally, using a coarser approximation of observational equivalence will lead to less accuracy in the test results. It is interesting to note that deficiencies in the EQN function manifest themselves differently for our two different kinds of test cases. Consider a test case of the form `(S1, S2, equivalent)`. Suppose that the resulting objects O_1 and O_2 are not observationally equivalent, but EQN returns true. Then an error has gone undetected. This is unfortunate, but it is in the nature of testing to live with the possibility that errors will be missed. On the other hand consider `(S1, S2, unequivalent)`. Suppose again the resulting objects O_1 and O_2 are not observationally equivalent, but EQN returns true. In this case, a spurious error report will be generated.

2.1 Example: Testing An Incorrect Implementation of SET_OF_INT

Consider again the SET_OF_INT ADT whose specification is shown in Figure 1. Figure 2 shows a buggy implementation of SET_OF_INT, in which a set is represented by a linked list, the insert operation inserts the new element at the front of the list, and the `Delete(x)` operation removes the first occurrence of `x`. The EQN(O_2) message, when sent to O_1, traverses the O_1's list checking that each element is a member of O_2, then traverses O_2's list checking that each element is a member of O_1.

This implementation is incorrect because either the `Insert(x)` operation should check that `x` is not already in the list, or the `Delete(x)` operation should remove all occurrences of `x`.

By reasoning about an informal specification, or by using the test generation technique described in section 3 below, we can generate the test case

```
(Create.Insert(1).Insert(1).Delete(1), Create, equivalent),
```

which says that the object obtained by creating a new set, inserting `1` twice, then deleting `1` should be in the same state as an object obtained by simply creating a new set. To execute the test case, we could use a driver like the one shown in Figure 3 which declares two variables O_1 and O_2 of class SET_OF_INT, sends the first sequence of messages to O_1, the second sequence to O_2, then sends EQN(O_2) to O_1. As illustrated in Figure 4, EQN returns false. Comparing this with the "equivalent" tag on the test case, we see that the error is exposed.

Of course, in practice we don't want to "hard-wire" a specific test case into the driver, as we have done in the above example. Instead, we can build a more sophisticated driver which reads in the test case from a file, sends the given sequences of messages to objects of the class under test, invokes EQN, and compares the result with the tag. Such a driver has very limited dependence on the specifics of the class under test. It is thus possible to automatically generate a test driver for class C from the interface of class C. In section 3 we describe a tool which generates such drivers, and an interactive tool for generating test cases from an algebraic specification.

3 Description of the Tools

In this section, we describe ASTOOT – **A** **S**et of **T**ools for **O**bject-**O**riented **T**esting, a system for testing classes based on the concepts explained in section 2. ASTOOT contains three components: the driver-

```
      -- -- Buggy SET: ‘‘delelete" contains a bug
      -- -- EQN uses ‘‘inclusion" of sets to test equality
class SET export
   empty, insert, delete, member, eqn
inherit
   LINKED_LIST [INTEGER]
      rename
         present as member, delete as ll_delete
feature
      -- -- Insert v into current set
   insert (v:  INTEGER) is
      do
         start; insert_left(v)
      end; -- -- insert

      -- -- Delete only the first v in the current set
   delete (v:  INTEGER) is
      do
         search(v, 1);
         if not offright then ll_delete end;
      end; -- -- delete

      -- -- Two sets are equivalent if
      -- -- (Current includes s) and (s includes Current)
   eqn (s:  like current):  BOOLEAN is
      local stop:  BOOLEAN
      do
            -- -- Test whether s includes Current
         from start until offright or stop
         loop
            if s.member(value) then forth
            else stop := true
            end; -- -- if
         end; -- -- loop

            -- -- Test whether Current includes s
         from s.start until s.offright or stop
         loop
            if member(s.value) then s.forth
            else stop := true
            end; -- -- if
         end; -- -- loop

         Result := not stop
      end; -- -- eqn
end; -- -- Class SET
```

Figure 2: Buggy implementation of SET_OF_INT in Eiffel

```
var O1, O2: SET_OF_INT;

begin
   O1.Create;
   O1.Insert(1);
   O1.Insert(1);
   O1.Delete(1);

   O2.Create;

   equiv := O1.EQN(O2);
   if not equiv then
      Generate_Error_Report
end.
```

Figure 3: Hard-wired driver

O1
Create
empty list
Insert(1)
$\rightarrow$ [1]
Insert(1)
$\rightarrow$ [1] $\rightarrow$ [1]
Delete(1)
$\rightarrow$ [1]

O2
Create
empty list

EQN returns false

Figure 4: State diagram

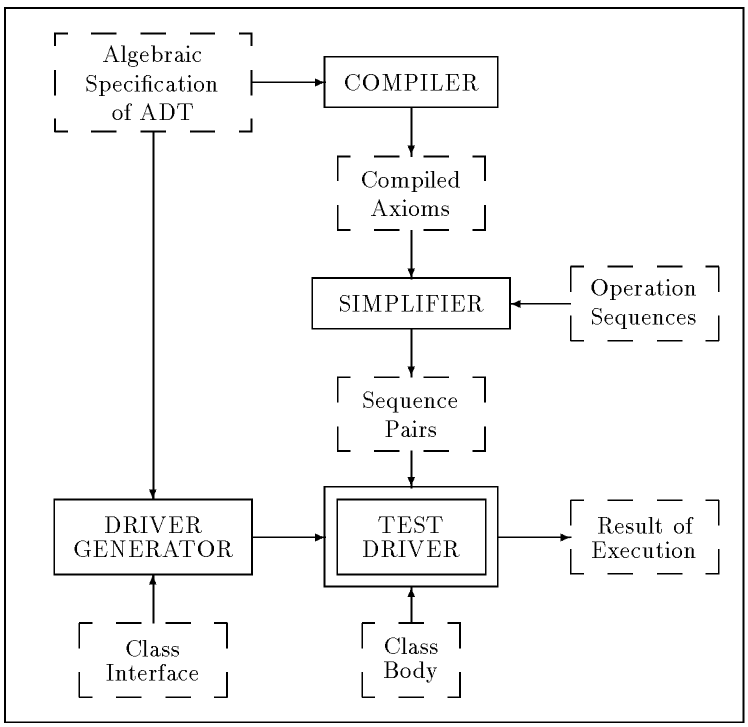

Figure 5: Components of ASTOOT

generator, the compiler, and the simplifier. The compiler and simplifier together form an interactive test generation tool, which generates test cases from an algebraic specification. The driver-generator automatically generates a test driver for a class from the class interface. This test driver can be used to execute automatically generated and/or manually generated test cases. The relationships between these components are illustrated in Figure 5.

3.1 The Driver Generator

When using the concept of sequence pairs to test a class, the test drivers become simpler. Furthermore, drivers for different classes are very similar to one another. This uniformity makes it possible to automate the process of creating test drivers. The driver generator takes the class interface and syntactic part of the specification as input, checks whether both of them have the same exported operations, checks whether the implementation has an EQN operation with the correct declaration. For testing purposes, the operation EQN has to be exported by the implementation. If the implementation language has the facility of selective export (like Eiffel), then we can let EQN be exported only to the test driver, so the integrity of the implementation can be preserved. The driver generator then builds a test driver, which is a program in the same language as the class being tested. The current version of the driver generator is targetted to Eiffel.

The resulting test driver reads in a set of test cases, one at a time from a file. Each test case consists of two sequences of messages followed by a tag indicating whether they are supposed to be equivalent or unequivalent. Since the test cases may have been generated manually, the test driver contains a parser that parses the input operation sequences to make sure they are syntactically correct to decrease the possibility of human errors. For each operation sequence pair, the test driver declares two objects of the class under test, one for the *original sequence* (called *original object*), the other for the *simplified sequence* (called *simplified object*). The test driver then, reads in the first operation in the original sequence, makes sure it is a "create" operation, reads in the arguments of the operation as character strings, makes sure the arguments are in the proper forms for the corresponding types, converts the arguments into proper types, sets up necessary assignments for these arguments, and applies the operation with its arguments to the original object. This process is repeated for every operation in the original sequence. Then, the same process is applied for the simplified object. Next, the test driver reads in the tag for the sequence pair, uses EQN to compare the final states of these two objects, examines the tag, and reports an error if the result of EQN does not agree with the tag.

3.2 Test generation tools

The compiler and simplifier are used to generate test cases from an algebraic specification of an ADT. The compiler translates the axioms from LOBAS into a language-independent internal representation. The simplifier takes an initial sequence of operations, supplied by the user, translates it into an internal representation called an ADT tree, then uses the axioms as re-write rules to obtain equivalent sequences of operations. This process also suggests "interesting" unequivalent sequences. We now describe the compiler and simplifier in more detail.

3.2.1 Compiler

The compiler first reads in a specification written in LOBAS and does syntactic and semantic checking on the specification. Since LOBAS is a language with multiple inheritance and genericity, the semantic checking of the compiler is more complex than conventional compilers. In addition to conventional semantic checking, the compiler has to resolve inheritance and genericity constraints.

Under multiple inheritance, if a class has two ancestor classes having operations with the same name, the programmer must rename one of them. The compiler checks to make sure there is no name clashing between ancestors of the class. Also, the compiler has to check for circular inheritance, that is, a class being an ancestor of itself.

In LOBAS, a generic parameter may be put under constraint, as in the declaration of SORTED_LIST [T $->$ COMPARABLE]. When the compiler encounters a variable of type SORTED_LIST[NAME], it has to make sure the class NAME is a descendant class of COMPARABLE.

After syntactic and semantic checking, the compiler outputs a list of axioms, in the form of trees, which will be used by the simplifier as rewriting rules. For instance, the left-hand-side and right-hand side of axiom 6 in SET_OF_INT are shown in Figure 6.

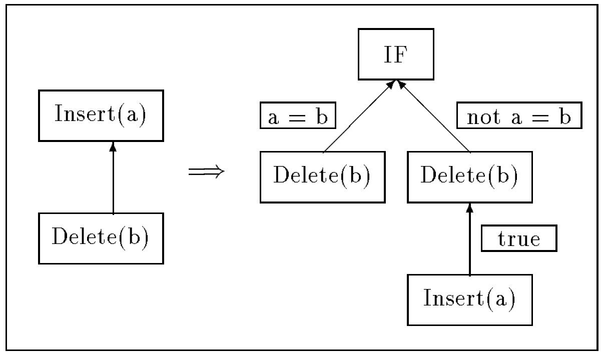

Figure 6: Axiom 6 of SET_OF_INT in ADT tree form

3.2.2 Simplifier

The simplifier takes a user-supplied sequence of operations and uses the rewriting rules generated by the compiler to obtain an equivalent sequence (i.e. a sequence which should put an object into the same state as the original sequence does).

Internally, the simplifier uses a tree called *ADT tree* to represent the states of an object. The branching of ADT tree arises from axioms having *IF_THEN_ELSE* expressions on the right hand side. Each node of the ADT tree is an operation with its arguments, which might also be ADT trees. Each edge of the ADT tree has a boolean expression, called the *edge condition*, attached to it. Each path from the root to a leaf of the ADT tree represents a possible state of the ADT. A boolean expression, called *path condition*, is the conjuction of all the edge conditions on that particular path. The path conditions in a given tree are mutually exclusive. The size of the tree may be exponential in the size of the initial sequence. To deal with this complexity, the simplifier can operate either in batch mode, which builds the entire tree, or in interactive mode, which allows the user to selectively guide the construction of a subtree.

In batch mode, the process of simplification is as follows:

- Search through the axioms to find an axiom with a left-hand-side that matches some partial path of the ADT tree,
- If an axiom is found, the simplifier will bind all the variables in the axiom to the proper arguments in the partial path of the ADT tree, and simplify the arguments. It then replaces the partial branch with the right-hand-side of the axiom.
- The process of matching-and-replacing stops when there is no matching axiom.

In order for the simplifier to work properly, the set of axioms in the specification must be *convergent*, i.e., the axioms must have the properties of *finite* and *unique termination* [MUS80]. The property of finite termination ensures the process of simplification will not go into infinite loop. The property of unique termination makes sure that any two terminating sequences starting from the same operation sequence have the same final results, no matter what choice is made as to which axiom to rewrite or which axiom to apply first.

An example, involving batch-mode simplification of the sequence `Create.Insert(x).Delete(y)` for the SET_OF_INT abstract data type is shown in Figure 7.

The simplifier will generate test cases of the form:

> `(Create.Insert(x).Delete(y), Create, equivalent)` with the path condition "x = y", and
>
> `(Create.Insert(x).Delete(y), Create.insert(y), equivalent)` with the path condition "not x = y" for the ADT tree (3) in Figure 7.

In interactive mode, the user picks out one of the paths of the ADT tree to be followed, attaches one more operation to the branch, waits for the simplifier to simplify the ADT tree, then repeats the process, as illustrated in the appendix.

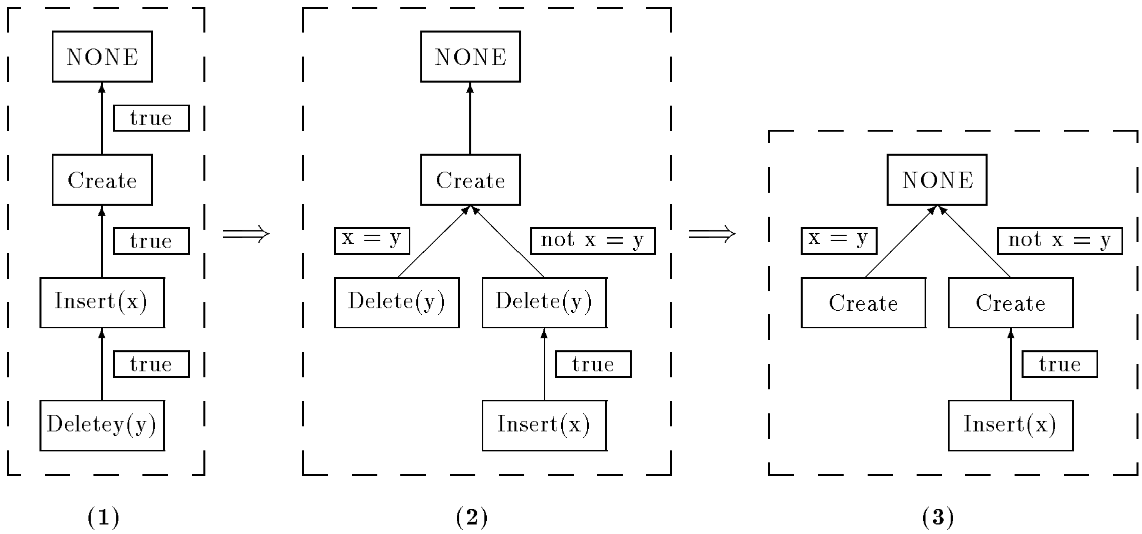

Figure 7: Simplification of create.insert(x).delete(y)

4 Inheritance

In this section, we discuss some of the ways in which inheritance impacts on testing. Suppose class C has been thoroughly tested and class C_d is a descendant of C. Under some circumstances, it is possible to "reuse" some of the results of testing C to reduce the effort of testing C_d.

We begin by noting that we can not expect the correctness of C_d to have any relation at all to the correctness of C unless there is a relationship between the specifications of the two classes. It is often, but not always, the case that the specification of C_d is a specialization or extension of the specification of C. The LOBAS specification language addresses this issue by incorporating a mechanism for inheritance of specifications. A descendent T_d of an ADT T contains all T's functions and may contain additional functions. Those functions which have been inherited from T obey the axioms of T. Additional axioms may further constrain the behavior of T_d.

Let T and T_d be the specifications of C and C_d respectively, where C_d is a descendent of C. Assume that C has already been thoroughly tested. In addition to testing C_d against it's own specification T_d, it is usually necessary to test C_d against T. This is because, given the dynamic binding mechanism, a variable of class C may be bound to an object of class C_d which uses the methods of C_d to respond to messages. The sender of such a message will expect the object to react to the message in accordance to the specification of C. In testing C_d against T, it may be unnecessary to perform a complete regression test (i.e. to re-execute all the tests used in testing C). This is because some of C_d's code has been inherited from C without modification. Thus, to check C_d against T it is only necessary to execute C_d those of C's test cases which involve methods which have been redefined.

Note that it is sometimes unnecessary for C_d to satisfy T. Programmers sometimes define a class C_d as a descendent of C strictly for convenience, in order to re-use C's code. For example, the implementation of SET_OF_INT shown in Figure 2 inherits from class LIST. A client of class SET_OF_INT (i.e. another class which sends messages to SET_OF_INT) may not care, or even be aware, that SET_OF_INT is implemented as a LIST, and thus may not care whether SET_OF_INT respects the specification of LIST. As long as none of the clients of C_d have the opportunity to assign an object of class C_d to a variable of class C, it is not necessary to check C_d against T. Of course, it is dangerous to omit testing of C_d against T unless the entire system has been carefully checked to assure that this is the case.

We next consider testing C_d against T_d. If T_d is a descendant of T some effort can be saved. Those test cases of T_d consisting of sequences of operations inherited from T will have already been considered in checking C_d against T. (As pointed out above, C_d may have been executed directly on them or the test results may have been "inherited" from the testing of C, depending on whether they involved redefined methods). Thus testing C_d against T_d can focus on sequences involving operations which are unique to T_d and their interaction with operations of T.

Finally, we mention another way in which inheritance can affect test data selection. When testing C_d against T_d, it may be useful to consider "high-level" information about an ancestor C of C_d to help select instantiations of parameters in the operation sequences. Consider a class SET_OF_INT which is implemented as a descendent of the BINARY_SEARCH_TREE class. We would expect the order of insertions to be important, and might, for example want to include test cases such as

```
Create.Insert(w).Insert(x).Insert(y).Delete(z),
```

in which (w,x,y) represented different permutations of (1,2,3). On the other hand, if SET_OF_INT were implemented as a descendent of a LIST class, we would not expect the relative order in which items are inserted into a set to effect the results.

5 Conclusions

5.1 Related Work

We now compare our approach to related work on testing data abstractions. These systems generally fall into two categories: test execution tools and test generation tools, and are based on the traditional model of test cases. In contrast, our approach involves a new model of test cases, which we believe to be more natural for testing data abstractions, and which gives rise to test generation and test execution tools.

5.1.1 Test Execution Tools

One of the first systems to address the question of testing data abstractions was DAISTS (Data Abstraction Implementation Specification and Test System) [GAN81]. DAISTS uses the axioms of an algebraic specification to provide an oracle for testing implementations of the ADT. A test case is a tuple of arguments to the left-hand-side of an axiom. DAISTS executes a test case by giving it as input to the left-hand side and right-hand side of an axiom, then checks the output by invoking a user-supplied equality function (similar to our EQN).

Our test execution tool can be considered to be a generalization of DAISTS. For example, executing the DAISTS test case

```
(old = Create.Insert(1).Insert(2), a = 3, b = 2)
```

on the SET_OF_INT axiom 6 is equivalent to our test case

```
(Create.Insert(1).Insert(2).Insert(3).Delete(2), Create.Insert(1).Insert(3),
equivalent),
```

in which the second sequence is obtained by using axiom 6 to rewrite the first sequence.

However, DAISTS has no analog of our test cases of the form (S_1, S_2, `unequivalent`). This has significant ramifications: Even exhaustive testing with DAISTS may fail to detect an error which results in two states being erroneously "melded" into a single state. As an extreme example, consider an erroneous implementation in which none of the operations change the state of the object. The two sides of each axiom will return the same state on any input, and thus the error will not be detected.

A second distinction between DAISTS and our approach is that DAISTS requires the availability of a formal specification, while our test execution tool can be used when only an informal specification is available.

Another approach to testing ADT's has been proposed by Hoffman and Brealy [HOF89]. This approach facilitates test execution through the manual preparation of tables listing test cases along with their expected output. Each table entry consists of a trace (a sequence of operations putting the ADT into some state), exceptions raised by that trace, an observer, the type of the observer, and the expected value of applying the observer to that state. Tools to aid in generation and executions of these tables have been built. One distinction between this approach and ours is by using the EQN function to check outputs, in effect, we combine many of Hoffman's test cases into a single test case. It also appears that our approach leads to easier generation of test drivers. On the other hand, Hoffman's handling of exceptions is certainly an important idea, which we would like to try to incorporate into future versions of ASTOOT.

5.1.2 Test Case Generation

Two previous approaches to generating test cases from algebraic specifications have been reported. Gaudel and her colleagues [CHO86, GAU88] have developed a system in which the specification is written in a dialect of Prolog, a complexity measure is defined for sequences of operations, and the Prolog interpreter is used to generate sequences of operations of given complexities, sometimes subject to additional constraints. This approach might provide a useful means to generate interesting initial sequences for our simplifier.

Jalote et. al. [JAL88, JAL89] suggest that effective test cases can be generated from the syntactic part of an algebraic specification, without reference to the semantics. Experience with our tools indicates that in fact, it is very important to consider the semantic part as well, since different instantiations of arguments in a sequence of operations can lead to profoundly different states.

5.2 Future Work

The current version of ASTOOT is an experimental prototype. We note several possible enhancements. The operations in the test cases produced by the simplifier contain only symbolic arguments. It is left to the user to find an instantiation of the arguments which satisfies the path condition. In the future, we would like to add a constraint solver to substitute these symbolic arguments by values satisfying the path conditions. Then, we can feed the output of the simplifier directly to the test driver.

Although the compiler is written for the language LOBAS (Language for Object-Based Specification), this component can be replaced by a compiler which inputs specifications in another algebraic specification language and outputs ADT tree.

The concepts for driver generator can be applied to any implementation language with object-oriented features. The current version is targetted to implementations written in Eiffel. We would like to develop versions targetted to other object-oriented languages.

At the current time, the initial sequences for the simplifier is generated manually. In the future, we would like to develop an operation sequence generator which will generate operation sequences according to some testing criteria.

5.3 Summary

We have presented a new model of how to test classes which places emphasis on the fact that classes are implementations of data abstractions. In this model a test case is a pair of sequences of operations along with a tag indicating whether they should put objects into the same abstract state. A test case is executed by sending the sequences to two objects of the class under test, checking whether they are in the same state by using a user-supplied equivalence checking function, EQN, then comparing this result with the tag.

We then described ASTOOT, a set of tools based on this model. The driver generator automatically generates test drivers from the interface of the class under test. The compiler and simplifier partially automate the task of test generation, using an algebraic specification of the ADT under test as an input.

When no algebraic specification is available, the drivers produced by the driver generator can be used to execute manually generated tests.

Our initial experience using these tools has been encouraging. We have been able to efficiently generate and execute many more test cases than we would ever wish to generate manually. So far, we have used the tools to try to gain insight into how to generate effective tests.

One of the first things which has become apparent is that in classes whose specifications involve conditional axioms, the particular values used to instantiate parameters are important in test cases. For example we tested a heap implementation of a priority queue in which there was an off-by-one error in the sift operation. We executed 2771 test cases of the form

```
(Create.Insert(a).Insert(b).Insert(c).Insert(d).Insert(e).Remove,
Create.Insert(a).Insert(b).Insert(c).Insert(d).Remove.Insert(e), equivalent)
```

where `e` is not the largest element of `a,b,c,d,e`. Of them, only 280 exposed the error. This indicates that test generation methods which generate only one test case for each sequence of operations, without regard to the values of the parameters will not be very effective.

We plan to use ASTOOT for further experimentation, comparing the effectiveness of various testing strategies for object-oriented programs, and to incorporate our findings into future versions of the tools.

References

[CHO86] N. Choquet, "Test Data Generation Using a Prolog with Constraints", *Proceedings of the Workshop on Software Testing*, Banff, Canada, July, 1986, IEEE Computer Society Press.

[DOO90] R. Doong, "LOBAS Reference Manual," Polytechnic University Computer Science Technical Report, in preparation.

[GAN81] J. Gannon, P. McMullin, R. Hamlet, "Data-Abstraction Implementation, Specification, and Testing", *ACM Transactions on Languages and Systems*, 3, 3, July, 1981.

[GAU88] M.–C. Gaudel, B. Marre, "Generation of Test Data from Algebraic Specifications", *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, July, 1988, IEEE Computer Society Press.

[GOL83] A. Goldberg, D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. 1983.

[GOG78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types", in *Current Trends in Programming Methodology, v. 4*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[GUT77] J.V. Guttag, "Abstract data types and the development of data structures", *CACM* 20, 6, June 1977.

[HOF89] D. Hoffman, and C. Brealey, "Module test case generation," *Proc. of the ACM SIGSOFT '89 3rd Symposium on Software Testing, Analysis, and Verification*, Dec. 1989.

[JAL88] P. Jalote, M.G. Caballero, "Automated Testcase Generation for Data Abstraction," *Proc. COMPSAC 1988*, IEEE Computer Society Press.

[JAL89] P. Jalote, "Testing the completeness of specifications", *IEEE Transactions on Software Engineering*, SE-15, 5, May 1989.

[LIS75] B.H. Liskov, S.N. Zilles, "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering*, SE-1,1, March, 1975.

[MEY88] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[MUS80] D. R. Musser, "Abstract data type specification in the AFFIRM system," *IEEE Transaction on Software Engineering*, SE-6,1, Jan. 1980.

[STR86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.

[WEY82] E.J. Weyuker, "On Testing Non-testable Programs," *The Computer Journal*, 25,4, 1982.