The ASTOOT Approach to Testing Object-Oriented Programs *

Roong-Ko Doong
Sun Microsystems Laboratories
Mountainview California

Phyllis G. Frankl
Polytechnic University
Brooklyn, NY

ABSTRACT

This paper describes a new approach to unit testing object-oriented programs, a set of tools based on this approach, and two case studies. In this approach, each test case consists of a tuple of sequences of messages, along with tags indicating whether these sequences should put objects of the class-under-test into equivalent states and/or return objects that are in equivalent states. Tests are executed by sending the sequences to objects of the class-under-test, then invoking a user-supplied equivalence checking mechanism. This approach allows for substantial automation of many aspects of testing, including test-case generation, test driver generation, test execution, and test checking. Experimental prototypes of tools for test generation and test execution are described. The test generation tool requires the availability of an algebraic specification of the abstract data type being tested, but the test execution tool can be used when no formal specification is available. Using the test execution tools, case studies involving execution of tens of thousands of test cases, with various sequence lengths, parameters, and combinations of operations were performed. The relationships between likelihood of detecting an error and sequence length, range of parameters, and relative frequency of various operations were investigated for priority queue and sorted list implementationgs having subtle errors. In each case, long sequences tended to be more likely to detect the error, provided that the range of parameters was sufficiently large and likelihood of detecting an error tended to increase up to a threshold value as the parameter range increased.

Key Words: Software testing, Object-oriented programming, Abstract Data Types, Algebraic Specification

1 Introduction

Object-oriented programming, based on the concepts of data abstraction, inheritance, and dynamic binding, is becoming an increasingly popular software development methodology. Much research has been done on developing object-oriented analysis and design techniques, developing object-oriented programming languages, and exploring how the methodology

*This research was supported in part by NSF grants CCR-8810287 and CCR-9003006 and by the New York State Science and Technology Foundation and was performed while the first author was at Polytechnic University. Authors' address: Department of Computer Science, Polytechnic University, 6 Metrotech Center, Brooklyn, NY 11201. e-mail: phyllis@morph.poly.edu.

changes the software development process. Yet relatively little research has addressed the question of how object-oriented programs should be tested.

We have developed a new approach to unit testing object-oriented programs, which is based on the ideas that the natural units to test are classes, and that in testing classes, one should focus on the question of whether a sequence of messages puts an object of the class under test into the "correct" state. In this approach, roughly speaking, each test case consists of a pair of sequences of messages, along with a tag indicating whether these sequences should result in objects that are in the same "abstract state". A test case is executed by sending each sequence of messages to an object of the class under test, invoking a user-supplied equivalence checking routine to check whether the objects are in the same abstract state, then comparing the result of this check to the tag. This testing scheme has several nice properties:

- Expected results of tests are included in test cases in a concise format (one Boolean) which is independent of the class being tested. This facilitates automatic checking of test results.
- Test drivers for different classes are very similar to one another, hence can be automatically generated from class interfaces.
- If an algebraic specification for the class under test is available, term rewriting can be used to generate test cases automatically. If no algebraic specification is available, a person can develop test cases by reasoning about an informal specification.

This approach is embodied in the prototype testing system ASTOOT, **A S**et of **T**ools for **O**bject-**O**riented **T**esting, which includes an interactive specification-based test case generation tool and a tool that automatically generates test drivers. For any class C, ASTOOT can automatically generate a test driver, which in turn automatically executes test cases and checks their results. In addition, when an algebraic specification for C is available, ASTOOT can partially automate test generation. Thus the system allows for substantial automation of the entire testing process.

The current version of ASTOOT is targeted to testing programs written in Eiffel[1]. Throughout this paper we assume that the classes being tested are written in Eiffel. However, the underlying ideas and tools can be adapted relatively easily to other object-oriented languages.

In Section 2 of this paper, we review relevant background material on software testing, object-oriented programming, and algebraic specification of abstract data types. Section 3 describes the ideas underlying ASTOOT – correctness of a class that implements an abstract data type, test case format, and test result checking. The tools are described in Section 4. Section 5 describes two case studies performed in order to gain more insight into how to generate good test cases. We compare our approach to related work in Section 6 and note directions for future work in Section 7.

[1]Eiffel is a trademark of the Nonprofit International Consortium for Eiffel (NICE).

2 Background

2.1 Background on Software Testing

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.

Perhaps the most obvious opportunity for partially automating testing is the generation of test cases. In order to automate test generation it is necessary to analyze some formal object, such as source code or a formal specification. Most research on automated test generation has involved *program-based* or *white-box* techniques, i.e., techniques based on analysis of the source code of the program being tested. However, white-box testing suffers from certain limitations, such as its inability to generate test cases intended to exercise aspects of the specification that have inadvertently been omitted from the program. *Black-box* or *specification-based* techniques, based on analysis of the program's specification, overcome some of these limitations, but cannot be automated unless some kind of formal specification is available. Manual black-box test generation techniques, based on informal specifications, are widely used in practice. The testing scheme described in this paper is a black-box approach which is automatable when a formal algebraic specification is available and which can be applied manually, otherwise.

Another 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, 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 a program P on a test case, it is necessary to check whether the result agrees with the specification of P. 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 [30]. Our testing method uses a novel approach which allows the correctness of test cases to be checked automatically by the test execution system.

2.2 Overview of Object-Oriented Programming

Object-oriented languages support *abstract data types*, *inheritance*, and *dynamic binding*. An abstract data type is an entity that 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, often 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 that 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. A class'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, and/or specialize C by redefining some of C's methods to provide alternative implementations.

A dynamic binding mechanism is used to associate methods with objects. In strongly typed object-oriented languages, 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 the method associated with class C_d. 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 allows construction of *polymorphic* data types.

Some examples of object-oriented languages include Smalltalk, C++, and Eiffel [13, 29, 27]. 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 [27] for an overview of the object-oriented approach.

2.3 Algebraic Specification of Abstract Data Types

Before we can talk about how to test a class C, we must have some concept of what it means for C to be correct. Thus, we must have some means, formal or informal, of specifying the entity that C is intended to implement and of stating the conditions under which the implementation conforms to the specification. In the case where C is intended to implement an abstract data type, algebraic specifications provide a formal means of doing this.

An algebraic specification has a syntactic part and a semantic part. The syntactic part consists of function names, and their signatures (the types they take as input and produce as output). In an algebraic specification of type T, functions which return values of types other than T are called *observers*, because they provide the only ways for us to query the contents of T. Functions which return values of type T are called *constructors* or *transformers*[2]. The

[2]Transformers are called *extensions* in [16].

distinction between constructors and transformers is clarified below.

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.

Term rewriting [24] has been used to define a formal semantics for algebraic specifications [28, 12]. Two sequences S_1 and S_2 of operations of ADT T are *equivalent* if we can use the axioms as rewrite rules to transform S_1 to S_2[3]. A specification can then be modeled by a *heterogeneous word algebra*, in which the elements are equivalence classes of sequences of operations.

For a specification $\mathcal{S}$ to be useful, it must be *consistent* and *sufficiently complete* [16]. A consistent specification must not contain contradictory axioms, i.e., no contradiction should be derivable from any operation sequences of the specification. Let W be the set containing all the operation sequences consisting of constructors or transformers of $\mathcal{S}$. $\mathcal{S}$ is sufficiently complete, if for every sequence w in W, the result of applying each observer of $\mathcal{S}$ to w is defined. Discussion of how to construct useful algebraic specifications can be found in [14, 15, 1].

Most algebraic specification languages use a functional notation. For convenience, we have designed a specification language, LOBAS, whose syntax is similar to OO programming language syntax [6]. The syntactic part of a LOBAS specification includes an *export* section which lists operations available to the users of the ADT. In LOBAS, the designer of a specification has to classify the operations into three categories — constructors, transformers, and observers. This classification process helps the designer in producing a sufficiently complete specification and facilitates the test generation scheme described in Section 4.2, below. An additional advantage of this notation is pointed out in Section 3.1.

Algebraic specifications of a priority queue in LOBAS and in functional notation are shown in Figure 1(a) and Figure 1(b). Sequences of operations (separated by dots) are to be read left to right, so that, for example, `create.add(5).add(3)` represents the result of creating a priority queue, then adding items 5 and 3 to it, in that order. According to the specification, `create.add(5).add(3).delete` is equivalent to `create.add(3)` because we can apply axiom 6 twice to give

$$
\begin{array}{ll}
 & \texttt{create.add(5).}\underline{\texttt{add(3).delete}} \\
\Longrightarrow & \texttt{create.}\underline{\texttt{add(5).delete}}\texttt{.add(3)} \\
\Longrightarrow & \texttt{create.add(3).}
\end{array}
$$

The difference between constructors and transformers becomes clear at this point. After the simplification is complete, only constructors are left in the operation sequence. The role of transformers is to transform a sequence of constructors into another sequence of constructors.

Note that the appearance of operation sequences in LOBAS bears a strong resemblance to *trace specifications* which data abstractions by specifying the legality, equivalence, and values of traces (operation sequences) [3, 20, 21]. Two advantages of trace specifications over

[3]This definition follows the assumption of Goguen [11]; Guttag [18] makes the opposite assumption, i.e., that two sequences *may* be assumed to be the equivalent unless provably inequivalent.

```
class Priority_Queue export
    create, largest, add, delete, empty, eqn
constructor
    create;
    add (x: Integer)
transformer
    delete
observer
    empty: Boolean;
    largest: Integer;
    eqn (B: Priority_Queue): Boolean
var
    A, B: Priority_Queue;
    x, y: Integer
axiom
    1: create.empty -> true;
    2: A.add(x).empty -> false;
    3: create.largest -> -∞;
    4: A.add(x).largest ->
          if x > A.largest then x
          else A.largest;
    5: create.delete -> create;
    6: A.add(x).delete ->
          if x > A.largest then A
          else A.delete.add(x);
    7: A.eqn(B) ->
          if A.empty and B.empty then true
          else if (A.empty and not B.empty) or
                  (not A.empty and B.empty)
          then false
          else if A.largest = B.largest
          then A.delete.eqn(B.delete)
          else false
end
```

(a) Specification in LOBAS

```
type Priority_Queue
syntax
    create: -> Priority_Queue;
    add: Priority_Queue × Integer
          -> Priority_Queue;
    delete: Priority_Queue -> Priority_Queue;
    empty: Priority_Queue -> Boolean;
    largest: Priority_Queue -> Integer;
    eqn: Priority_Queue × Priority_Queue
          -> Boolean;
declare
    A, B: Priority_Queue;
    x, y: Integer;
semantics
    1: empty(create) -> true;
    2: empty(add(A,x)) -> false;
    3: largest(create) -> -∞;
    4: largest(add(A,x)) ->
          if x > largest(A) then x
          else largest(A);
    5: delete(create) -> create;
    6: delete(add(A,x)) ->
          if x > largest(A) then A
          else add(delete(A),x);
    7: eqn(A,B) ->
          if empty(A) and empty(B) then true
          else if (empty(A) and not empty(B)) or
                  (not empty(A) and empty(B))
          then false
          else if largest(A) = largest (B)
          then eqn(delete(A),delete(B))
          else false
end
```

(b) Specification in functional notation

Figure 1: Specifications of Priority Queue

LOBAS are their ability to specify functions (observers) with side effects, and their ability to handle operation sequences with intermingled procedures and functions. However, the axioms of LOBAS (and other algebraic languages) facilitate automatic test case generation, as discussed in Section 4.2, below.

3 Self-checking test cases

In this section we describe the main concepts that underlie ASTOOT. These include a notion of correctness for classes, a model of test cases and their execution, and a test checking mechanism.

In one of the early papers on specification of data abstractions [25], 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 may lead to overspecification of the underlying representation of the data. Instead, they and others [11, 14, 17] proposed algebraic specifications of abstract data types (ADTs), which define the intended behavior of an ADT by giving axioms describing the interaction of operations.

Similarly, it is possible to test a class by testing each of its methods individually, treating each as a function mapping some input space to some output space, selecting elements of that input space, and examining the outputs to see if they are correct. However, doing so shifts the focus of testing away from the essence of the data abstraction – the interaction between operations. Furthermore, testing each method individually necessitates the construction of complicated drivers and output checking mechanisms. For example, a test case for the `add` operation in a priority queue would consist of priority queue and an item and the output would be another priority queue. Thus the driver would have to initialize the input priority queue, and checking the output would entail examining the output priority queue to see if it is the correct result. In contrast, our approach to testing classes focuses on the interaction of operations.

In this section, we restrict attention to classes intended to implement ADTs. We require that,

1. operations have no side effects on their parameters,
2. functions (observers) have no side effects,
3. functions (observers) can only appear as the last operation of a sequence, and
4. when a sequence is passed as a parameter to an operation it must *not* contain any functions (observers).

The main reason for placing restrictions 1 and 2 is that we *cannot* specify these kinds of side effects by using either LOBAS or purely algebraic languages. The reason behind restriction 3 is that sequences that mix functions and procedures are not syntactically valid in LOBAS or other algebraic specification languages[26]. Restriction 4 makes it easier to generate test cases using ASTOOT. Note that restriction 4 does not hinder our ability to express test cases involving any parameters to an operation, since when function f has no side effects on its target object (the object to which the message is sent), the target object of a sequence $S.f$ will be observationally equivalent to the target object of S. Techniques for relaxing restrictions Restrictions 2, 3, and 4 are discussed in [6].

3.1 Correctness of an ADT implementation

Consider a class C, intended to implement abstract data type T. Each function in T corresponds to a method of C, and inputting a value of type T to a function corresponds to sending a message to an object of class C. In Eiffel, constructors and transformers are typically coded as procedures; rather than explicitly returning an object of class C, such a procedure "returns" a value by modifying the state of the object to which it has been applied. An observer can be coded as a function which explicitly returns an object of another class. We will refer to the object which a function or procedure message is sent as the *target object* and to the object returned as the *returned object*. For procedures, the target object and the returned object are the same (though typically the *value* of the target object will be changed by the procedure call). Notice that in addition to explicitly returning an object, a function also implicitly "returns" its target object. If the function is side effect free then the value of the target object will be unchanged by the function call.

The syntax of LOBAS, unlike the functional syntax of most algebraic specification languages, allows us to differentiate between the target and returned values. For example, in the sequence `create.add(5).add(3).largest` the final value of the target is a priority queue whose elements are `5` and `3`, and the returned value is `5`.

We will say that objects O_1 and O_2 of class C are *observationally equivalent* if and only if

- C is a built-in class and O_1 and O_2 have identical values, or
- C is a user-defined class and for any sequence S of operations of C ending in a function returning an object of class C', $O_1.S$ is *observationally equivalent* to $O_2.S$ as objects of class C'.

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 and related classes. Two observationally equivalent objects are in the same "abstract state", even though the details of their representations may be different. For example, consider a circular array implementation of a first-in-first-out (FIFO) queue. Two arrays containing the same elements in the same order would be observationally equivalent (as queues), even though the elements could occupy different portions of the underlying arrays.

We now define the notion of correctness that underlies our approach.

> A class C is a *correct* implementation of ADT T, if there is a signature-preserving mapping from operations of T to those of C such that
>
> - for any pair (S_1, S_2) of sequences of operations of T, S_1 is equivalent to S_2 if and only if the corresponding sequences of messages give rise to *observationally equivalent* returned objects.

In other words, there is a one-to-one correspondence between the "abstract states" of T and the "abstract states" of C, which preserves the transitions between abstract states. Note that, based on the definition of *returned object*, our definition of correctness demands that operation sequences consisting entirely of constructors and transformers give rise to observationally equivalent *target* objects and that operation sequences ending in observers return observationally equivalent objects.

Other notions of correctness, some corresponding to other specification methodologies, have also been investigated [3, 11, 18, 8]. Our definition, based on observational equivalence, is similar to that corresponding to trace specifications [3], but is based on the more limited algebraic specification methodology. It is a pragmatic and intuitively appealing one, which lends itself to a convenient testing strategy.

3.2 Test Case Format

This definition of correctness gives rise in a natural way to a framework for testing. If we had an infinite amount of time and a way to check whether two objects were observationally equivalent, we could exhaustively test class C as follows:

- Consider the set $\mathcal{U}$ consisting of all 3-tuples (S_1, S_2, tag), where S_1 and S_2 are sequences of messages, and tag is "equivalent" if S_1 is equivalent to S_2 according to the specification, and is "not-equivalent", otherwise.

- For each element of $\mathcal{U}$, send message sequences S_1 and S_2 to objects O_1 and O_2 of C, respectively, then check whether the returned object of O_1 is observationally equivalent to the returned object of O_2.

- If all the observational equivalence checks agree with the tags, then the implementation is correct; otherwise it is incorrect.

Unfortunately, we have neither an infinite amount of time for testing, nor a fool-proof way of checking observational equivalence. Nonetheless, this scheme suggests an approach to testing. We demand that C and each class that is returned by a function of C include a method called EQN which approximates an observational equivalence checker and we select elements of $\mathcal{U}$ as test cases. In addition to shifting the emphasis of testing from functionality of individual methods to the notion of state, this approach to testing facilitates automation of many aspects of the testing process.

Note that the elements of $\mathcal{U}$ can be viewed as "self-checking" test cases. That is, each test case includes information, in the form of the tag, describing the expected result of execution. Furthermore the format of this expected result (a single boolean) is very concise and is independent of the particular class being tested and of the pair of sequences to be executed. This facilitates automated execution and checking of test cases. Of course, when generating such test cases, it is necessary to consider the specification of the ADT in order to derive the tags. This can either be done semi-automatically by manipulating a formal specification, as described in Section 4 below, or manually by a reasoning about a formal or informal specification.

For example, consider a priority queue of integers, whose functions are described informally as follows:

`create` – creates an empty priority queue,

`add` – adds an integer to the priority queue,

`delete` – removes the largest element of the priority queue,

`largest` – returns the value of largest element of the priority queue, without modifying the contents of the priority queue, and

`empty` – determines whether the priority queue is empty.

By reasoning about this informal specification, a person can generate test cases such as,

1. `(create.add(5).add(3).delete,create.add(3),equivalent)`,
2. `(create.add(5).add(3).delete.largest,create.add(3).largest,equivalent)`,
3. `(create.add(5).add(3).delete,create.add(5),not-equivalent)`, and
4. `(create.add(5).add(3),create.add(3).add(5),equivalent)`.

Test case 1 says that `create`ing an empty priority queue, `add`ing `5` then `3`, then applying `delete` should be the same as creating an empty priority queue and `add`ing `3` to it. Test case 2 says that the objects returned by applying `largest` to those two priority queues should be equivalent. Test case 3 says that if we create an empty priority queue `add 5` and `3`, then `delete`, it should *not* be the same as if we create an empty priority queue and `add 5` to it. Test case 4 says that a priority queue obtained by `add`ing 5 then `add`ing 3 should be observationally equivalent to one obtained by `add`ing 3 then `add`ing 5. Unlike the previous three test cases, this test case captures an aspect of the informal specification that is not expressed in the formal specification, and thus it cannot be derived from the formal specification by using term rewriting[4]. This indicates that, even when a formal specification that partially describes the intended semantics of an ADT is available, manual generation of additional test cases may be useful.

We refer to test cases consisting of a pair of sequences along with a tag as *restricted format* test cases. More general test case formats which are useful for testing classes involving side effects and dynamic binding are introduced in [6].

3.3 The EQN Method

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 sequence to the objects, in practice EQN will approximate a check for observational equivalence.

[4]If an axiom such as `A.add(x).add(y) --> A.add(y).add(x)` were added to the specification, this aspect of the informal specification would be captured. However, the resulting specification would no longer satisfy the finite termination condition.

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, axiom 7 of Figure 1 specifies such an EQN function based on the priority queue specification. Note that this is actually only an approximation of true observational equivalence because it neglects the possible effects of "building up" the priority queues, then removing elements. Thus, it might say that two objects are equivalent when they are not[5]. Also, since EQN calls `largest`, and `delete`, an error in one of these operations may propagate to EQN, causing it to mask out the error. On the other hand, the error propagation can also help in error detection, as demonstrated in Section 5.1, below.

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 a FIFO queue is represented as a linked list, one can traverse the two lists comparing the elements. In general, 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. Furthermore, for some representations of some data structures, writing an implementation-level EQN operation may be extremely difficult and error-prone, even when the other methods are relatively simple.

It is also sometimes possible to use a very coarse approximation of observational equivalence as the EQN function. For example, we might consider two FIFO queues to be equivalent if they have the same number of elements, or if they have the same front element. This version of EQN may consider two inequivalent objects to be equivalent. Naturally, using a coarser approximation of observational equivalence will lead to less accuracy in the test results.

Bernot et al. [4] discuss a closely related problem, and suggest that an "oracle hypothesis" be explicitly stated. In the context of our approach to testing, such a hypothesis would describe the conditions under which the implementation of EQN is equivalent an actual check for observational equivalence.

4 The Tools

ASTOOT is a set of tools based on the approach described in Section 3. The current prototype, which handles test cases in the restricted format, has three components: the *driver generator*, the *compiler*, and the *simplifier*. The driver generator takes as input the interface specifications of the class under test (CUT) and of some related classes and outputs a test driver. This test driver, when executed, reads test cases, checks their syntax, executes them, and checks the results. The compiler and simplifier together form an interactive tool for semiautomatically generating test cases from an algebraic specification. Note that when no algebraic specification is available, the drivers produced by the driver generator can be

[5] For example consider an implementation which completely empties the priority queue whenever the total number of `add`s performed reaches a particular number $N > 2$. The recursive EQN would consider `O1.create.add(1).add(2).delete` equivalent to `O2.create.add(1)`, but in fact, performing an additional $N-2$ `add`s followed by $N-2$ `delete`s on each object would leave `O1` empty and leave `O2` non-empty.

used to execute test cases which have been derived by a person reasoning about an informal specification. The structure of ASTOOT is illustrated in Figure 2 and a screen dump of an ASTOOT session is shown Figure 4.

4.1 The Driver Generator

Our approach to testing leads to relatively simple test drivers, which operate by reading in test cases of the form (S_1, S_2, Tag), one at a time, checking that the sequences are syntactically valid, sending sequences S_1 and S_2 to objects O_1 and O_2 of CUT, comparing the returned objects of S_1 and S_2 with EQN, and checking whether the value returned by EQN agrees with Tag. On the other hand, drivers are complicated enough that writing them manually is a tedious and error-prone task. In particular, checking the syntactic validity of the operation sequences involves complicated parsing and type checking. For example, our driver for the priority queue class has over 400 lines of code (not counting inherited classes), most of which deals with checking the syntax of the operation sequences. Luckily, drivers for testing different classes are very similar to one another in structure. This has allowed us to write a tool, the *driver generator*, which automatically generates test drivers. The driver-generator can be viewed as a special-purpose parser generator, which, based on the syntax described in the class interfaces, generates test drivers that parse test cases, as well as executing and checking them.

The driver generator, DG, operates in three phases. The first phase is to collect information about interfaces of the CUT, its ancestors, and all the classes which are parameter types or return types of CUT's operations. DG first checks whether each of these classes has an exported EQN operation.[6] (If, like Eiffel, the implementation language has the facility of selective export then we can let EQN be exported only to the test driver, so the integrity of the implementation can be preserved.) In the second phase, DG builds a test driver, which is a class in the implementation language. The current version of the driver generator is targeted to Eiffel 2.1, but the underlying ideas can be applied to other OO languages. In the third phase DG compiles and executes the test driver with test cases supplied by the user.

4.2 Test Generation Tools

ASTOOT's test generation component has two parts, the compiler and the simplifier, both of which are based on an internal representation called an *ADT tree*. The compiler reads in a specification written in LOBAS and does some syntactic and semantic checking on the specification[7], then translates each axiom into a pair of ADT trees.

An ADT tree is a tree in which nodes represent operations along with their arguments. Each path from the root to a leaf of an ADT tree represents a possible state of the ADT. The branching of ADT tree arises from axioms having *IF_THEN_ELSE* expressions on the right-

[6] For ASTOOT to access functions that are hidden in the implementation, the CUT should export these functions to the test driver generated by ASTOOT. In Eiffel this can be achieved by "selective export" to the test driver; in C++ this can be achieved by making the test driver a friend class of the CUT.

[7] Because the simplifier and the driver generator operate under the assumption that `create` is the instantiation operation, the compiler makes sure there is a constructor named `create` in the specification. Also, the simplifier will insist that the first operation of a sequence is the `create` operation.

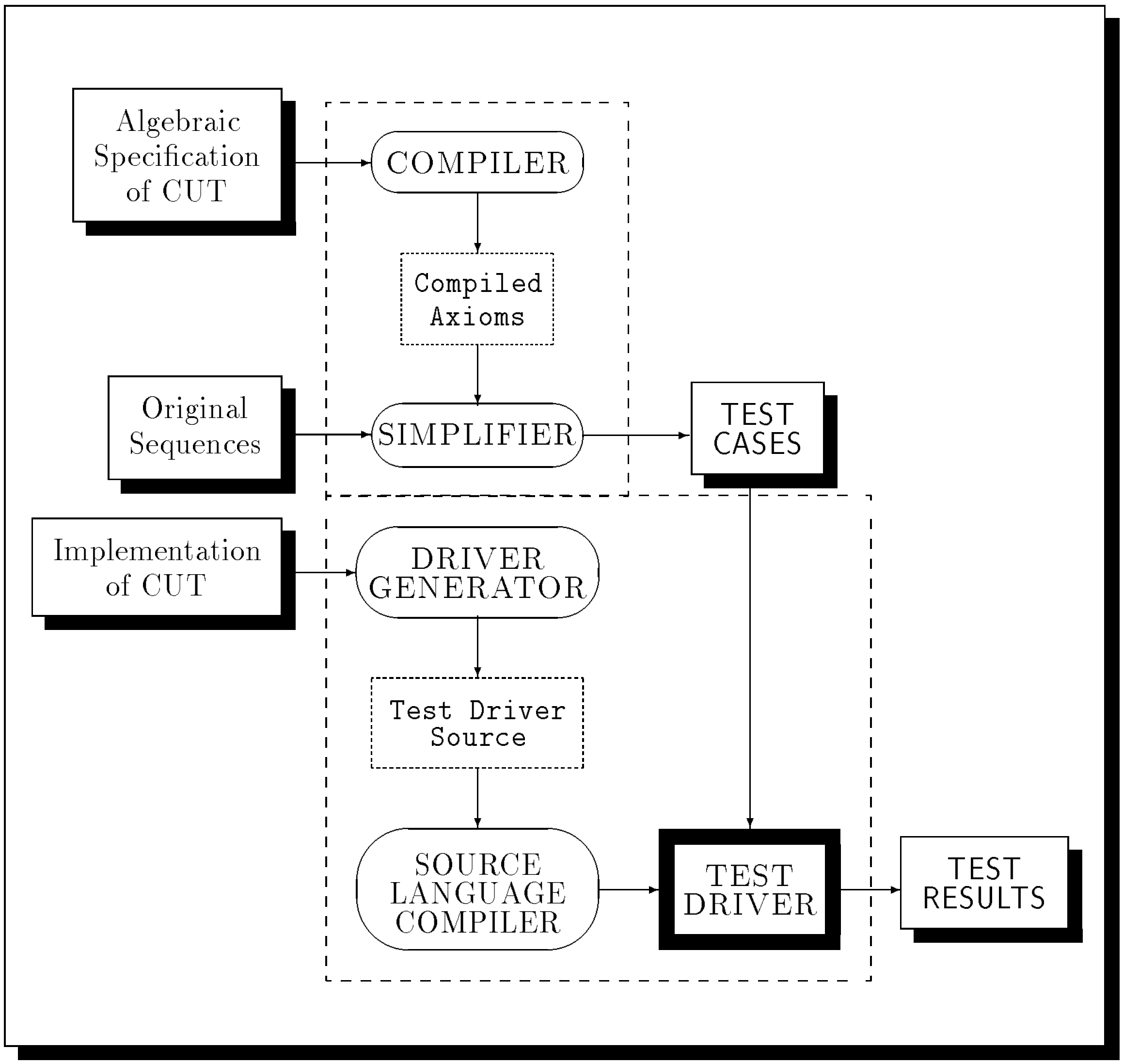

Figure 2: Components of ASTOOT

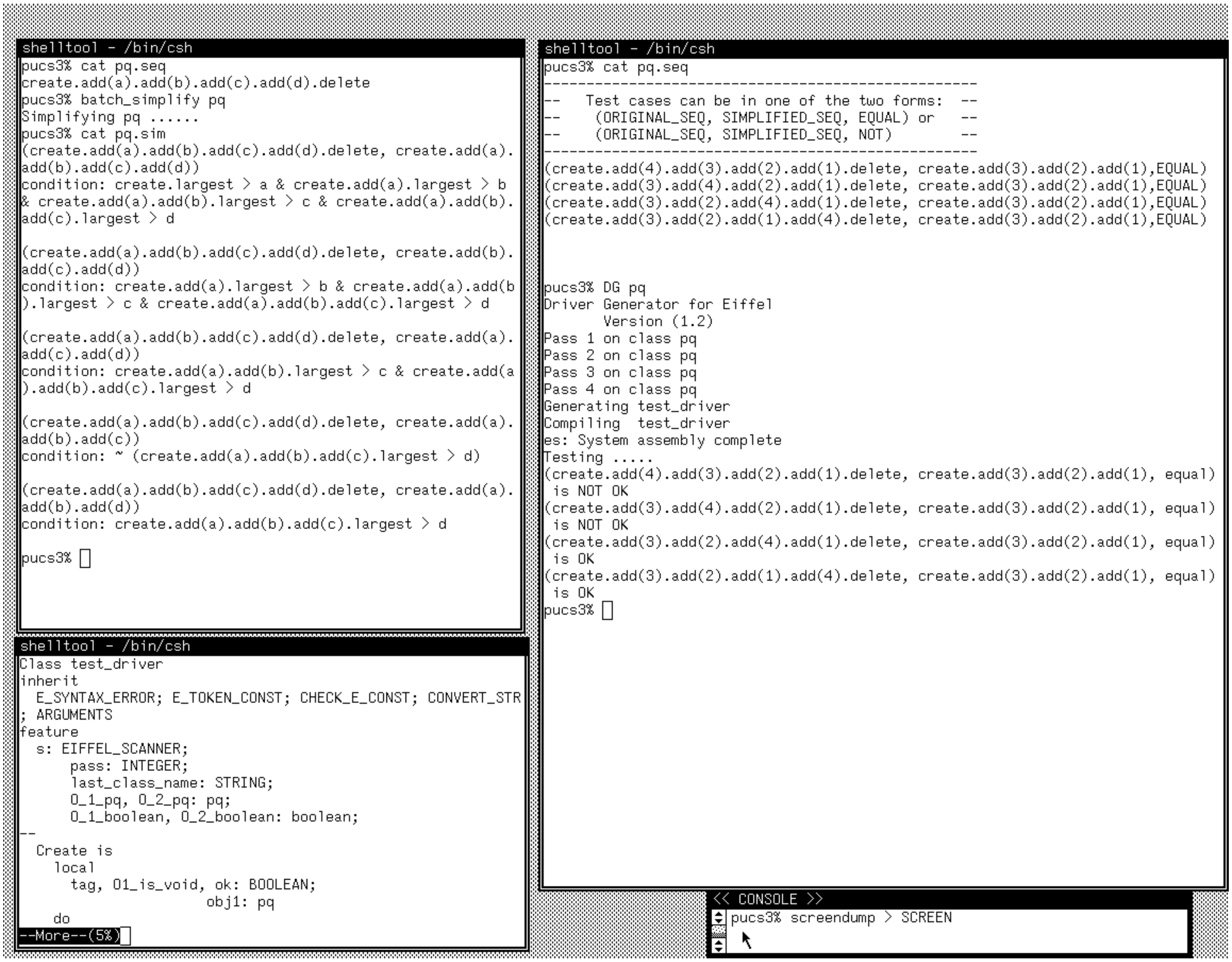

Figure 3: Screen Dump of an ASTOOT session. The upper left window shows the execution of the test generator in batch mode on a priority queue specification. The file pq.seq contains an initial sequence, supplied by the user. The test generator generates five test cases based on this initial sequence and writes them, along with the corresponding constraints on the free variables, to the file pq.sim. The constraint on each test case is obtained by conjoining the condition for that test case with the negations of the conditions on previous test cases. The upper right window shows the four test cases the user has developed by instantiating the free variables with values that satisfy the constraints. (The first of the generated test cases has an unsatisfiable constraint, so it is eliminated by the user). The driver generator is then invoked on an incorrect implementation of the priority queue (described in Section 5.1 below). It invokes Eiffel to compile the class under test, generates a test driver for the class, compiles it, then executes the given test cases. The first two test cases detect a bug, while the second two do not. The lower left window shows a small portion of the test driver which was automatically generated by the driver generator.

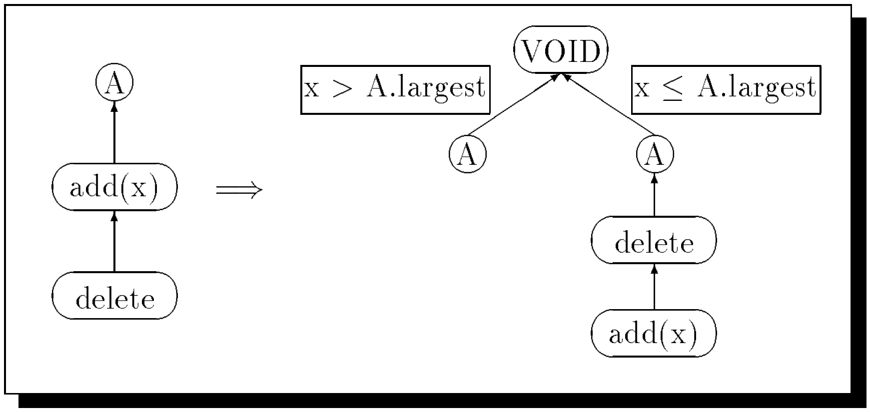

Figure 4: Axiom 6 of Priority Queue in ADT tree form

hand side. Each edge of the ADT tree has a boolean expression, called the *edge condition*, attached to it. The *path condition* of a path from the root to a leaf is the conjunction of all the edge conditions on that path; it indicates the conditions under which the operation sequence on that path is equivalent to the original sequence. The path conditions in a given tree are mutually exclusive. Figure 4 illustrates the ADT tree pair of Axiom 6 in Figure 1. For clarity, the edge conditions are shown in rectangles in the figure; in the implementation, parameters of operations and the operands in the boolean expressions are, themselves, represented by ADT trees.

The simplifier inputs an operation sequence, supplied by the user, translates it into an ADT tree, and applies the transformations to obtain equivalent operation sequences. The process of simplification is as follows:

1. Search through the axioms to find an axiom with a left-hand side that matches some partial path of the ADT tree (ignoring the edge conditions);
2. If an axiom is found, bind all the variables in the axiom to the proper arguments in the partial path of the ADT tree, and simplify the arguments; then replace the partial branch with the right-hand side of the axiom;
3. Repeat steps 1 and 2 until there is no matching axiom.

In the worst case, the ADT tree arising from a sequence of ℓ operations may have m^ℓ paths, where m is the maximum number of branches in any axiom. To deal with this complexity, the current prototype can operate either in batch mode, which builds the entire equivalent ADT tree or in interactive mode, which allows the user to selectively guide the construction of a particular path through the tree.

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* [28]. 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 results, no matter what choice is made as to which axiom to rewrite or which axiom to apply first.

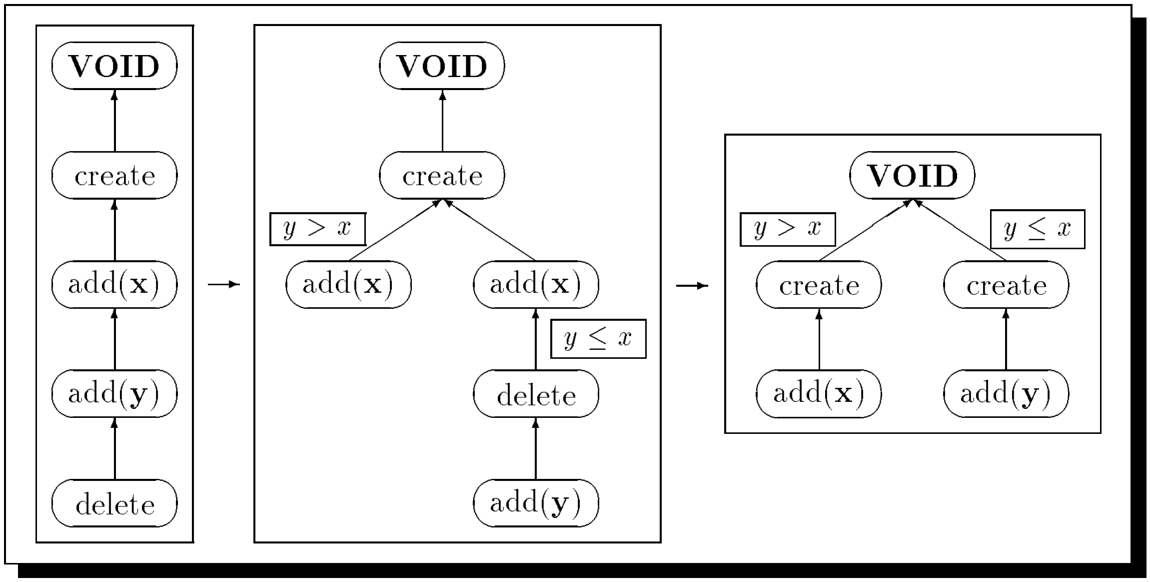

Figure 5: Simplification of the Sequence create.add(x).add(y).delete

An example, involving batch-mode simplification of the sequence `create.add(x).add(y).delete` for the priority queue is shown in Figure 5. The simplifier will generate test cases of the form:

> `(create.add(x).add(y).delete, create.add(x), equivalent)`
> with the path condition "y > x", and
>
> `(create.add(x).add(y).delete, create.add(y), equivalent)`
> with the path condition "y $\leq$ x".

Note that the simplifier also suggests test cases with `not-equivalent` tags. For instance, we can exchange the path conditions and the test cases from above to get the following test cases:

> `(create.add(x).add(y).delete, create.add(x), not-equivalent)`
> with the constraint "y $\leq$ x", and
>
> `(create.add(x).add(y).delete, create.add(y), not-equivalent)`
> with the constraint "y > x".

For an ADT tree with n paths, the simplifier will generate n test cases that have `equivalent` tags. In principle, the simplifier could also generate $n(n-1)$ test cases that have `not-equivalent` tags, where n is $O(m^\ell)$, ℓ is the length of the original sequence, and m is the maximum number of branches in any axiom. Because there are too many such cases in an ADT tree, the current version of the simplifier leaves selection of such test cases to the user.

Note that the test cases generated by the simplifier contain symbolic values. To make them acceptable to the test driver, the user has to resolve the path conditions (constraints) and instantiate the symbolic values with the corresponding actual values. In principle, this could sometimes be done automatically by a constraint solving system. In the current prototype, constraint solving is left to the user.

Two important questions remain: how should one select original sequences to input to the simplifier, and how should one select paths through the resulting ADT trees, in order to increase the likelihood of exposing errors?

5 Case Studies

To gain insight into what kind of original sequences the person using the test generation tools should select and what kind of paths through the ADT tree should be generated in interactive mode, we performed two case studies, involving generating many tests for a buggy priority queue implementation and for a buggy sorted-list implementation. We choose the priority queue ADT because we knew it to be sufficiently complicated to exhibit many interesting phenomena. We purposely introduced the bug, but believe that it is one which could easily occur in practice. The sorted-list was based on a 2-3 tree, implemented for a graduate algorithms class. The bug was a slight variation on one which had actually occurred during program development.

We wished to gain insight into the following questions:

(ℓ): How does the length of the original sequence affect the likelihood that a test case will detect an error?

(p): How does the selection of parameters for operations in the original sequence affect the likelihood that a test case will detect an error?

(r): How does the ratio of `add`'s to `delete`'s in the original sequence affect the likelihood that a test case will detect an error?

We addressed these questions by randomly generating and executing several thousand test cases with various original sequence lengths, various ranges in which parameters could lie, and various frequencies of occurrence of different operations. For each original sequence we generated the corresponding simplified sequence, then executed the test case *(original sequence, simplified sequence, equivalent)*. Note that it would have been extremely difficult to execute and check so many test cases, had it not been for ASTOOT's "self-checking" test case concept.

5.1 Testing A Buggy Implementation of Priority Queue

In this case study, the CUT was a priority queue, implemented using a *heap* with a bug in the `delete` operation[8]. Specifically, the *Downheap* (or *sift*) operation performed by `delete` has an *off-by-one* error which causes it to sometimes fail to swap with the bottom row. The erroneous `delete` code is shown in the appendix.

[8] Recall that a heap is a complete binary tree in which each node is greater than or equal to its children; in the heap implementation of a priority queue, the `delete` operation is performed by removing the root, replacing it by the right-most leaf, then "sifting" that element down to its proper position.

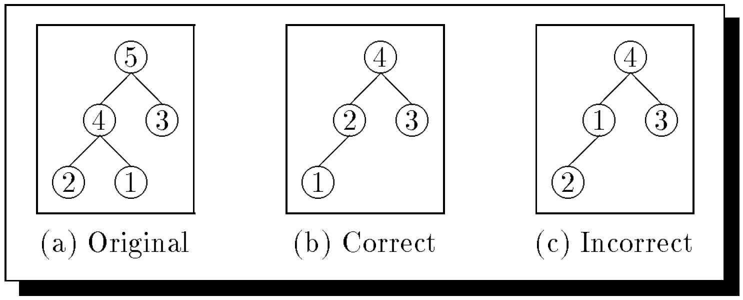

Figure 6: Illustration of Buggy Priority Queue

In Figure 6, (a) is the heap resulting from sequence `create.add(5).add(4).add(3).add(2).add(1)`, (b) is the heap resulting from applying a *correct* `delete` to (a). (c) is the resulting heap when the *incorrect* `delete` is applied to (a); note that `1` has failed to swap with `2` in the bottom row.

As discussed in Section 3.3, since EQN calls `delete`, the bug in `delete` is *propagated* to EQN. Even though the original sequence in test case `(create.add(5).add(4).add(3).add(2).add(1).delete,create.add(4).add(3).add(2).add(1),equivalent)` produces an incorrect heap (Figure 6(c)), EQN reports that the original sequence and the simplified sequence are equivalent due to the bug in `delete`. Thus, in this case, the error is *masked* by the propagation of the bug from `delete` to EQN.

On the other hand, consider the test case `(create.add(4).add(3).add(2).add(1).delete,create.add(3).add(2).add(1),equivalent)`, The original sequence produces a heap with 3 in the root, 1 in the root's left child, and 2 in the root's right child. The simplified sequence produces a heap with 3 in the root, 2 in the root's left child, and 1 in the root's right child. These two heaps are both correct and should be observationally equivalent. However in checking executing EQN to check observational equivalence, we call the erroneous `delete` routine. After the first call to `delete` the original heap has 2 in the root and 1 in the left child, while the (incorrect) "heap" resulting from the simplified sequence has 1 in the root and 2 in the left child. After one more call to `largest` which compares the roots, EQN reports that these two sequences are *not* equivalent, so the bug is detected. Thus, in this case, propagation of the error to EQN helps in error detection.

In order to carry out these case studies, we needed to generate tens of thousands of test cases. In principle, we could have done this using the ASTOOT test case generator by *randomly* generating original sequences with symbolic values as parameters, and sending each original sequence to the simplifier to generate test cases. This would give $O(2^{\ell})$ test cases for each original sequence with ℓ operations. Each test case would have symbolic values constrained by the path condition of the corresponding path. To be realistic, we would have to *randomly* choose some test cases and would have to *randomly* instantiate the symbolic values of each test case with actual values that satisfy the constraint of that test case either manually or with the aid of a constraint solver.

Note that the number of test cases needed for these experiments is several orders of magnitude larger than the number of test cases one would typically use in practice to test an implementation of this size. In order to generate this huge number of test cases efficiently and to do a broad range of testing with the three variables, ℓ, p, and r of a test set R, we used a C program to randomly generate test cases with actual values of those three variables, rather than using the ASTOOT test generator. This C program consists of three modules. The first module generates original sequences one at a time according to the three parameter of R, which are

ℓ — the number of operations (excluding `create`) in an original sequence,

p — the parameter to `add` is an integer in the range $[1 \ldots p]$, and

r — the ratio of `add`'s to `delete`'s appearing in the original sequences.

Operations of an original sequence are read in by the second module one at a time and applied to a priority queue that is implemented by a list. The third module inspects the contents of the list, generates a simplified sequence, and outputs an appropriate test case. Note that the simplified sequences are the same as if they were generated the test case generator of ASTOOT and instantiated with real values that satisfy the constraints.

For each test set we generated 1000 test cases. The average number of `add`'s in simplified sequences is approximately

$$\begin{cases} \frac{\ell(r-1)}{r+1} & \text{if } r \geq 1 \\ 0 & 0 \leq r < 1 \end{cases}$$

Results of Priority Queue Case Study

The percentages of test cases that expose the bug in each test set are shown in Figure 7. Inspection of these graphs shows:

(ℓ): For large values of p, the parameter range, long original sequences are better than short ones. However, if the parameter range is too small, longer original sequences may do worse than shorter ones. In fact, the results of test sets $R_{(100,10,3)}$, $R_{(100,10,6)}$, and $R_{(100,10,9)}$ are the worst in $r = 3$, $r = 6$, and $r = 9$ respectively, despite the fact that they have long original sequences.

(p): As the parameter range p increases, test cases tend to get better. However, in each case there appears to be a threshold above which the error detection probability levels off.

(r): Likelihood of exposing an error depends somewhat on r.

In this buggy implementation, failure only occurred when it was necessary to swap with the right-most element in the bottom row of the heap. Apparently, the long sequences were potentially more likely to cause the object to enter such a state, either during application of the original or simplified sequences, or through propogation of the error to the EQN operation. However, simply using a long sequence, without regard to the parameters chosen could lead to objects that never got into these "interesting" states. If the range of parameter values is too small, there will be many duplicates in the heap, so when an item is deleted, it is less likely that the sifted item will be *strictly* smaller than all of the elements it is compared to; thus it is less likely that it was supposed to swap with the bottom row.

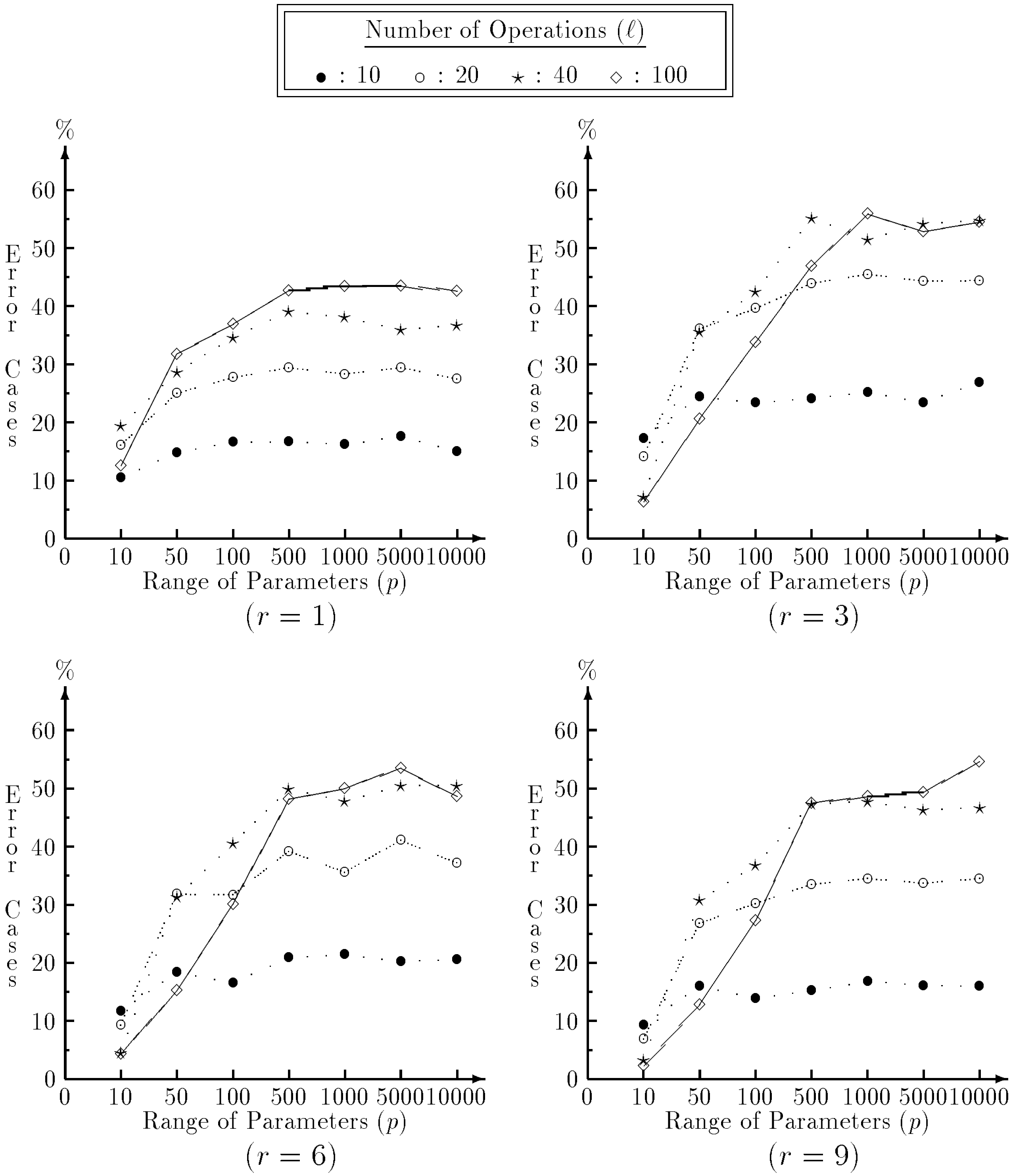

Figure 7: Results of Testing Priority Queue Using Randomly Generated Test Suites

```
– – Sorted list without duplicated elements
class SORTED_LIST export
    create, add, delete, nb_elements, find, eqn
constructor
    create;
       – – create an empty list
    add(x: INTEGER)
       – – If x is not in the list
       – – then add x to the list in the proper order
transformer
    delete(i: INTEGER)
       – – If 1 ≤ i ≤ nb_elements then
       – – delete the i-th element
observer
    nb_elements: INTEGER;
       – – Number of elements in the list
    find(i: INTEGER): INTEGER;
       – – Return value of the i-th element
       – – precondition: 1 ≤ i ≤ nb_elements
    eqn (other: SORTED_LIST): BOOLEAN
       – – Is other equivalent to the list?
end
```

Figure 8: Specification of Sorted List

5.2 Testing A Buggy Implementation of Sorted List

The second case study used an abstract data type, *sorted list of integer*, with six operations, `create`, `add`, `delete`, `find`, `nb_elements`, and `eqn`. The interfaces, preconditions, and informal specification of the sorted-list are shown in Figure 8. The EQN operation compared the lengths of the lists, then compared them element-by-element. Note that we did not use any formal specification for this sorted list. The test cases were generated by using a C program similar to the one in the case study of priority queue.

The sorted list was implemented using a *2-3 tree* (a special case of *B-tree*). The implementation has approximately 1000 lines of Eiffel 2.1 code, and the buggy version was produced by deleting one particular line from the correct version of implementation. The absence of this statement affects the state of the 2-3 tree only when the following situation occurs:

1. A node ((θ) in Figure 9(a)) has three children, such that the first child (α) has three children, and both the second child (β) and the third child (γ) have two children.

2. one of of γ's children is then deleted.

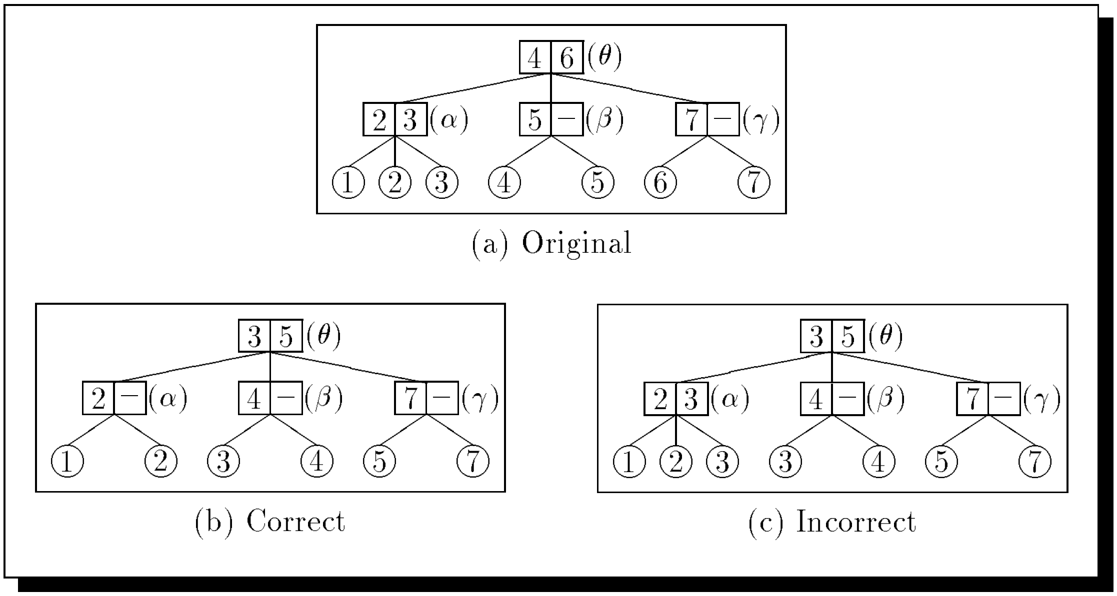

Figure 9: Illustration of the Buggy 2-3 Tree

For example, after deleting `6` of γ from the 2-3 tree in Figure 9(a), the correct procedure is:

1. copy `5` from β to γ,
2. delete `5` from β,
3. copy `3` from α to β, and
4. delete `3` from α.

The line that is missing from the buggy version does step (4) in the above procedure. As illustrated in Figure 9, deleting `6` from (a) will get 2-3 tree (c).

As in the priority queue case study, test sets were randomly generated with various original sequence lengths, various parameter ranges, and various ratios of `add`'s to `delete`'s. The original sequences contained `create`, `add` and `delete` operations and the simplified sequences contained only `create` and `add` operations.

Results of Sorted List Case Study

The results, shown in Figure 10 exhibit similar phenomena to those discussed for the priority queue example. Since the number of elements in a sorted-list is at most equal to the range of parameter values, only long sequences of operations with a large range of parameter values will produce 2-3 trees with a large number of leaves. The error in this program is such that failure only occurs when a deletion is performed on a 2-3 tree in a particular kind of state. Apparently, 2-3 trees with a large number of leaves are more likely to enter such a state.

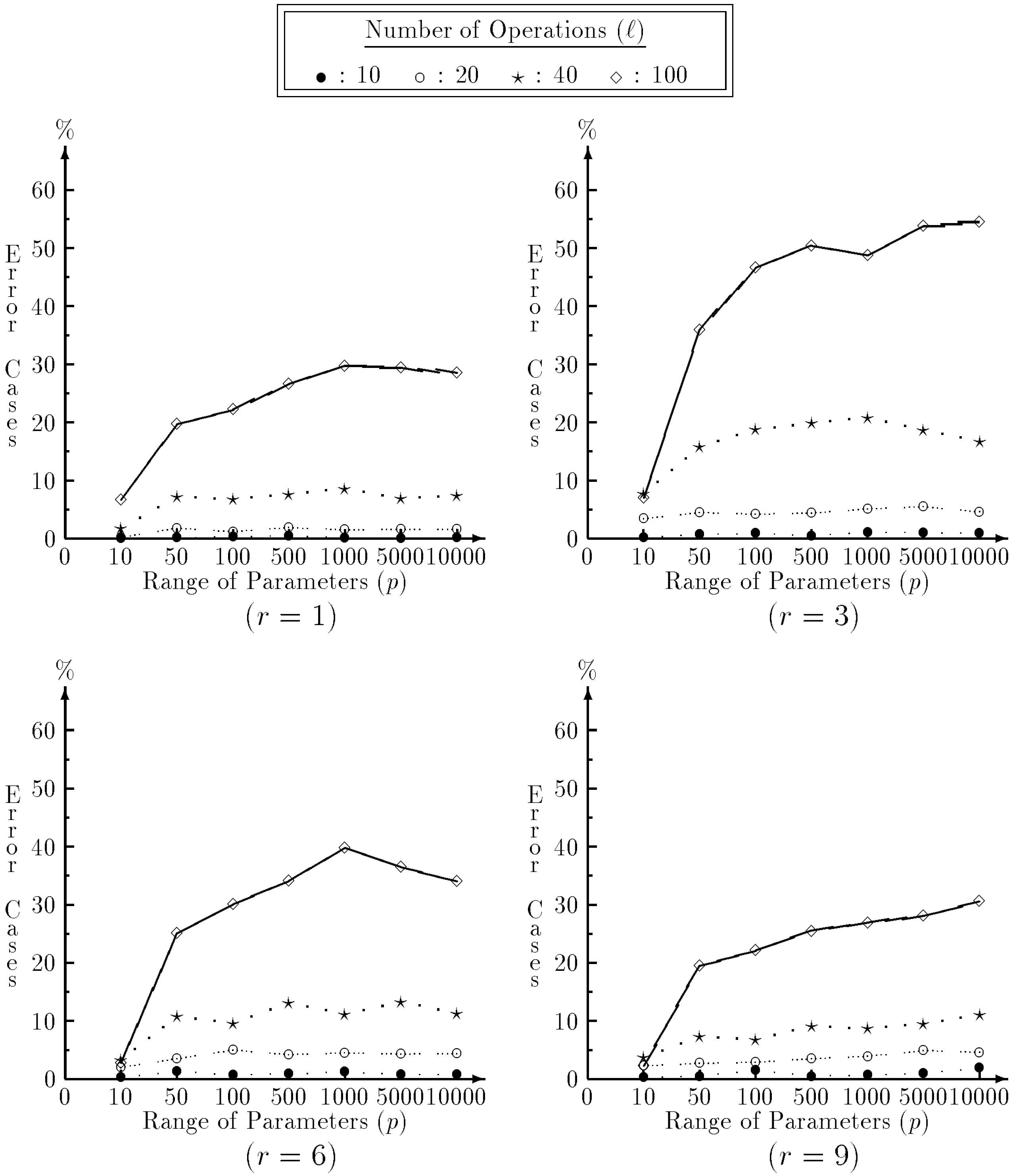

Figure 10: Results of Testing 2-3 Tree Using Randomly Generated Test Suites

5.3 Discussion of Case Studies

These case studies were intended to provide insight into the effects of such factors as the length of the original sequence, the relative frequencies of different operations in the original sequence, and the range of parameters to operations. In both of the case studies, the results showed that long original sequences do better than short ones, provided that the range of parameters is large enough to take advantage of the length. In addition, different ratios of `add`'s to `delete`'s in the original sequence gave different results.

We certainly do not want to over-generalize from these two small examples. However it seems safe to say that the *potential* that the relative values of the parameters would be important was apparent from the specification. In both of these cases, the specification involved *comparison* of items, using the *less than* operator. It is thus very reasonable to expect that different orderings of the parameters added would lead to different states, some of which might be more likely than others to expose the error. On the other hand, had we been testing a stack or queue ADT, we would not expect that the particular parameters would matter at all, and had we been testing a set ADT, we would expect the number of duplications to be important, but would not necessarily expect the relative order of parameters to be important (unless of course the set was implemented using an ADT based on comparison, such as a 2-3 tree.)

Another phenomenon we noticed was that different ratios of `adds` to `deletes` led to different probabilities of error detection. When the ratio is one, it is unlikely that the objects will grow very large in the course of testing. In our examples, small objects were apparently not usually complicated enough to excite the failure.

We offer the following tentative guidelines as to how to generate test cases:

- Use (at least some) long original sequences, with a variety of relative frequencies of different constructors and transformers.
- If the specification has conditional axioms (with comparison operators) choose a variety of test cases for each original sequence, with various parameters chosen over a large range. Equivalently, choose a variety of different paths through the ADT tree arising from each original sequence.

While these guidelines might seem obvious, previous research has suggested limiting the complexity of sequences[9] [5, 10] and ignoring the semantics of the specification [23, 22].

6 Related Work

We now compare our approach to related work on testing data abstractions. Previous systems generally fall into one of two categories – test execution tools and test generation tools. In contrast, our approach gives rise to both test generation and test execution tools.

[9] Gaudel's group suggests using relatively simple sequences but including a "regularity hypothesis" asserting that if the simple sequences (such as those with length less than some n) give correct outputs, so will more complex sequences. Our results can be interpreted as saying that such regularity hypotheses do not hold for the ADT's examined for small n. Similar observations lead Gaudel et al. to introduce additional "uniformity hypotheses".

6.1 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) [9]. 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, recall that Axiom 6 of the priority queue specification shown in Figure 1 says,

$$\begin{aligned} &\text{A.add(x).delete} \;--> \textbf{if } \text{x} > \text{A.largest } \textbf{then } \text{A} \\ &\qquad\qquad\qquad\qquad\quad \textbf{else } \text{A.delete.add(x)} \end{aligned}$$

Executing the DAISTS test case `(A = create.add(1).add(2), x = 3)` on this axiom is equivalent to our test case `(create.add(1).add(2).add(3).delete,create.add(1).add(2), 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,`not-equivalent`). This has significant ramifications – even exhaustive testing with DAISTS may fail to detect an error that results in two states being erroneously combined 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* tools, i.e., the drivers produced by the driver generator, can be used when only an informal specification is available, as in our second case study.

Hoffman et al. [21, 19] have developed several test execution tools for abstract data types, based on trace-specifications [3]. Their most recent system, *Protest*, consists of two subsystems:

1. *Protest/1* tests C implementation using test cases containing the expected output,
2. *Protest/2* compares the behavior of a C implementation to that of a user-supplied oracle written in Prolog.

A test case of Protest/1 is a 5-tuple *(trace, expexc, actval, expval, type)* where *trace* is a sequence of operations which puts the ADT into some state, *expexc* is the exception raised by the trace, *actval* is an observer, *expval* is the expected value of applying the observer to that state, and *type* is the data type of *actval* and expval. In Protest/2 the values *expexc* and *expval* are generated by a Prolog oracle written by the user.

Protest, which is a program written in Prolog, executes test cases by calling the operations in the implementation under test through an interface supplied by the user. For each operation, the interface defines a Prolog predicate which calls the corresponding C function in the implementation. The user also needs to write functions that can be called to construct objects of user-defined classes to be passed as parameters to the operations in the implementation. Like ASTOOT and DAISTS, Protest uses functions supplied by the user to check the equivalence between objects of corresponding ADTs. But, unlike ASTOOT and DAISTS, which use the specification under test to generate expected outputs, Protest/2 uses Prolog oracle to produce expected outputs. This oracle is another program that needs to be tested on its own.

Another distinction between this approach and ours is that by using the EQN function to check outputs, in effect, we combine many Protest test cases into a single test case. On the other hand, Protest's handling of exceptions is certainly an important idea, which we would like to try to incorporate into future versions of ASTOOT.

Antoy and Hamlet [2] have proposed a system that compares a class implementation to a more abstract representation which is based on term re-writing and is derived directly from the specification. The user supplies an explicit representation function mapping the concrete representation to the abstract representation. The code is instrumented to check that diagrams corresponding to each method commute, i.e., that applying the representation function then the abstract analog of the method gives rise to an abstract state that is equivalent to the one obtained by applying the method then applying the representation function. Such a system would, in some cases, give more accurate checks for correctness than would our approach of using an *approximation* of observational equivalence (the EQN function) to compare concrete representations. However, it imposes on the programmer the highly non-trivial task of writing a correct representation function.

6.2 Test Case Generation

Two previous approaches to generating test cases from algebraic specifications have been reported. Gaudel's research group [4, 5, 10] has developed a general theory of testing based on *testing contexts*, which are triples consisting of a set of hypotheses about the program, a set of test data, and an oracle. This approach has the nice property that if it can be established that the hypotheses hold and if the test set exposes no errors, then the program is guaranteed to be correct. (But of course, establishing that the hypotheses is a non-trivial task, involving analysis of the program text). Our approach provides test data and oracles; furthermore, the oracles appear in a simple and uniform format. An interesting direction for future research would be extending our approach to include hypotheses, perhaps by deriving conditions under which one sequence pair can be used to represent a class of sequence pairs and conditions under which one instantiation of parameters can be used to represent a class of instantiations.

Gaudel's group has also built a tool for testing data abstractions based on the theory of testing contexts. The tool inputs a specification written in a dialect of Prolog, and, based on some definition of the complexity of sequences, uses a Prolog interpreter to generate sequences of operations of given complexities, sometimes subject to additional constraints. This approach might provide a useful means to generate interesting original sequences for our simplifier.

Jalote et al. [23, 22] 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, corresponding to different paths through the ADT tree, can lead to profoundly different abstract states of the specification. Thus, it is necessary to select many different paths through the ADT tree arising from a given original sequence, or, equivalently, to choose values of parameters that exhibit different relationships to one another. This phenomenon was demonstrated in our case study of priority queue, where failure only occurred when it was necessary to swap with the bottom row of the heap.

7 Conclusion

We have described a new approach to testing classes which places emphasis on the fact that classes are implementations of data abstractions, a set of tools based on this approach, and two case studies. In this approach, each test case consists of a tuple of sequences of messages, along with tags indicating whether these sequences should put objects of the class-under-test into equivalent states and/or return objects which are in equivalent states. A test case in the restricted format consists of a single pair of sequences with a tag indicating whether the two objects resulting from application of these sequences should be observationally equivalent. Tests are executed by sending the sequences to objects of the class-under-test, then invoking a user-supplied equivalence checking mechanism. This approach allows for substantial automation of many aspects of testing, including test case generation, test driver generation, test execution, and test checking.

ASTOOT is a set of tools based on this approach. ASTOOT consists of a tool which automatically generates test drivers from class interface specifications and a tool which semi-automatically generates test cases from an algebraic specification of the class under test. The drivers generated by ASTOOT's driver generator automatically execute and check test cases which have been supplied either by the test generator or by manual generation. Consequently ASTOOT allows for substantial automation of the entire testing process.

We performed two case studies, one using a buggy implementation of a priority queue, and the other using a buggy 2-3 tree implementation of a sorted list. These case studies provided some insight into the effects of such factors as the length of the original sequence, the relative frequencies of different operations in the original sequence, and the range of parameters to operations.

The approach and tools described in this paper assume that the specification and implementation satify several restrictions which limit the kind of side-effects operations may have. Several extensions to the basic model, intended to make this testing scheme more applicable to "real-world" object-oriented programs, rather than just "pure" abstract data type implementations are described elsewhere [6, 7]. These include a *general format* for test cases, which allows testing of classes whose methods have side effects and a *dynamic format* that allows testing of virtual classes and some observations on the impact of inheritance on testing.

Directions for future research include the following:

- Interface the test generator with a constraint-solving system in order to decrease the need for manual intervention in test generation.

- Perform additional case studies, including exploration of more complicated ADT's and implentations with a larger variety of errors. While the particular errors in each of these case studies tended to be exposed when the number of duplicate elements in the sequence of insertions was low, it is easy to envision other errors for which the opposite would be true. Much more experience is needed in order to develop better intuition into what kind of test sequences should be generated for arbitrary classes with unknown errors. Ultimately, such intuition can be incorporated into heuristics to guide the selection of initial sequences and paths through the ADT trees, thus enhancing the test generator.

- Explore whether various strategies involving picking "special values" as parameters (such as inserting elements in ascending or descending order) help or hinder;
- Develop specification languages which are better able to express such aspects of object-oriented programming as side-effects, inheritance, and dynamic binding, then building tools based on them.
- Explore the impact of inheritance on testing.

While we have focused so far on unit-testing, there are also many interesting questions pertaining to how to system-test object-oriented software. We hope to address these questions in the future, and ultimately, to use the results to expand and improve ASTOOT.

Acknowledgments

The authors would like to thank Dan Hoffman and the anonymous referees for several useful suggestions.

References

[1] S. Antoy. Systematic design of algebraic specifications. In *Proceedings of 5th International Workshop on Software Specification and Design*, pages 278–280, May 1989.

[2] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. Technical Report TR 91–1, Rev. 1, Portland State University, Jan. 1992.

[3] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*, pages 111–130. Addison-Wesley Publishing Company, Reading, MA, 1986.

[4] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, Nov. 1991.

[5] N. Choquet. Test data generation using a prolog with constraints. In *Proceedings of Workshop on Software Testing*, pages 132–141. IEEE Computer Society, 1986.

[6] R.-K. Doong. *An Approach to Testing Object-Oriented Programs.* PhD thesis, Polytechnic University, Brooklyn, NY, January 1993. Also appeared as Computer Science Department Technical Report No. PUCS-110-92.

[7] R.-K. Doong and P. G. Frankl. Case studies on testing object-oriented programs. In *Proc. of Symposium on Testing, Analysis, and Verification (TAV4)*, pages 165–177, Oct. 1991.

[8] J. D. Gannon, R. G. Hamlet, and H. D. Mills. Theory of modules. *IEEE Transactions on Software Engineering*, 13(7):820–829, July 1987.

[9] J. D. Gannon, P. R. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, July 1981.

[10] M. Gaudel and B. Marre. Generation of test data from algebraic specifications. In *Proc. Second Workshop on Software Testing, Verification, and Analysis*, pages 138–139, Banff, Canada, July 1988.

[11] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. *Current Trends in Programming Methodology*, 4, 1978.

[12] J. A. Goguen and T. Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Aug. 1988.

[13] A. Goldberg and D. Robson. *Smalltalk–80: The Language and its Implementation.* Addison–Wesley, Reading, MA, 1983.

[14] J. J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[15] J. J. Guttag. Notes on type abstraction (Version 2). *IEEE Transactions on Software Engineering*, 6(1):13–23, Jan. 1980.

[16] J. J. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[17] J. J. Guttag, E. Horowitz, and D. R. Musser. Some extensions to algebraic specifications. In *Proceedings of Language Design for Reliable Software*, pages 63–67, Mar. 1977.

[18] J. J. Guttag, E. Horowitz, and D. R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, Dec. 1978.

[19] D. Hoffman and C. Brealey. Module test case generation. In R. A. Kemmerer, editor, *Proceedings of ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, pages 97–102. ACM Press, Dec. 1989.

[20] D. Hoffman and R. Snodgrass. Trace specifications: Methodology and models. *IEEE Transactions on Software Engineering*, 14(9):1243–1252, Sept. 1988.

[21] D. M. Hoffman and P. Strooper. Automated module testing in Prolog. *IEEE Transactions on Software Engineering*, 17(9):934–943, Sept. 1991.

[22] P. Jalote. Testing the completeness of specifications. *IEEE Transactions on Software Engineering*, 15(5):526–531, May 1989.

[23] P. Jalote and M. G. Caballero. Automated testcase generation for data abstraction. In *Proceedings of COMPSAC 88*, pages 205–210, Chicago, IL, Oct. 1988.

[24] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Elmsford, NY, 1970.

[25] B. H. Liskov and S. N. Zilles. Specification techniques for data abstractions. *IEEE Transactions on Software Engineering*, 1(1):7–19, Mar. 1975.

[26] J. M. Mclean. A formal method for the abstract specification of software. *Journal of ACM*, 31(3):600–627, July 1984.

[27] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, New York, NY, 1988.

[28] D. R. Musser. Abstract data type specification in the AFFIRM system. *IEEE Transactions on Software Engineering*, 6(1):24–32, Jan. 1980.

[29] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1991.

[30] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

Appendix

```
  -- Buggy delete
delete is
  local
    parent_ptr, child_ptr,
    l_child_ptr, r_child_ptr: INTEGER;
    parent, child: INTEGER;
    stop: BOOLEAN
  do
    if length > 0 then
      -- Move the last element to the first
      array.enter(1, array.entry(length));
      length := length - 1;

      from
        parent_ptr := 1;
        l_child_ptr := 2 * parent_ptr;
        r_child_ptr := 2 * parent_ptr + 1
      until
        stop or l_child_ptr >= length
      -----------------------^------------
      --  The correct statement is      --
      --  stop or l_child_ptr > length  --
      ------------------------------------
      loop
        -- find proper child
        if ((l_child_ptr = length) or
            (array.entry(l_child_ptr) >
            (array.entry(r_child_ptr)))) then
          child_ptr := l_child_ptr
        else
          child_ptr := r_child_ptr
        end; -- if

        parent := array.entry(parent_ptr);
        child := array.entry(child_ptr);
        if parent < child then
          -- swap
          array.enter(parent_ptr, child);
          array.enter(child_ptr, parent);
          parent_ptr := child_ptr;
          l_child_ptr := parent_ptr * 2;
          r_child_ptr := l_child_ptr + 1
        else
          stop := true
        end; -- if
      end; -- loop
    end; -- if
  end; -- delete
```