A Tool for Writing and Debugging Algebraic Specifications

Johannes Henkel * and Amer Diwan *
{henkel,diwan}@cs.colorado.edu

Abstract

Despite their benefits, programmers rarely use formal specifications, because they are difficult to write and they require an up front investment in time. To address these issues, we present a tool that helps programmers write and debug algebraic specifications. Given an algebraic specification, our tool instantiates a prototype that can be used just like any regular Java class. The tool can also modify an existing application to use the prototype generated by the interpreter instead of a hand-coded implementation. The tool improves the usability of algebraic specifications in the following ways: (i) A programmer can "run" an algebraic specification to study its behavior. The tool reports in which way a specification is incomplete for a client application. (ii) The tool can check whether a specification and a hand-coded implementation behave the same for a particular run of a client application. (iii) A prototype can be used when a hand-coded implementation is not yet available. Two case studies demonstrate how to use the tool.

1. Introduction

Formal specifications have many software engineering benefits. Perhaps the most important advantage of formal specifications is that they can provide an unambiguous and possibly machine checkable documentation of an interface. Clients of the specified interfaces know exactly what the interface provides and can therefore use it correctly. Programmers implementing the interface know exactly how an implementation of the interface should behave and therefore have a gold standard with which to test their implementation. Unfortunately, most programmers do not write formal specifications because they are difficult to write and require significant mathematical maturity on the part of programmers. This paper describes and evaluates a tool for developing algebraic specifications.

There are many kinds of formal specifications, each with their own strengths and weaknesses. For example, axiomatic specifications (e.g., [12]) are well suited for describing how methods manipulate the state of an object. They are thus valuable for programmers who try to understand and extend an existing implementation [9]. On the other hand, for programmers who are interested in using a particular class without worrying about the implementation details, understanding axiomatic specifications can be cumbersome. In contrast, for certain classes, in particular many container classes, algebraic specifications (e.g., [13]) can be short and elegant; they often provide the advantage of capturing the observable behavior without exposing the implementation details of objects. Since container classes are among the most frequently reused classes, and thus particularly benefit from machine checkable documentation, we focus only on algebraic specifications in this paper.

Given an algebraic specification for a class and a client for the class, our tool runs the client using interpretation to simulate the behavior of the specified class. In this way, when a programmer writes an algebraic specification, the system automatically provides an implementation. Our tool interprets algebraic specifications using term rewriting, which is a well studied area [6, 20]. However, to our knowledge our system is the first to seamlessly integrate fully automatic algebraic rewriting techniques with Java classes.

Our system provides three main benefits. First, it gives programmers more for their effort: they not only get the benefit of a formal specification but they also get a prototype of their class, which they can immediately use. This feature may be particularly useful in multi-programmer projects since it allows the developers of some components to test against specifications of other components before those are even implemented. Second, by providing a feature for experimentally validating a specification against an implementation, our tool helps prevent divergence of implementation and its specification as the software system evolves. Third, our system is invaluable for debugging algebraic specifications since it allows programmers to "run" a specification and observe its behavior. When running a specification, there are three possible outcomes: (i) the run produces correct answers, which suggests that the specification may be sound and complete; (ii) the run produces incor-

* This work is supported by NSF grants CCR-0085792, CCR-0133457, and CCR-0086255. Any opinions, fi ndings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

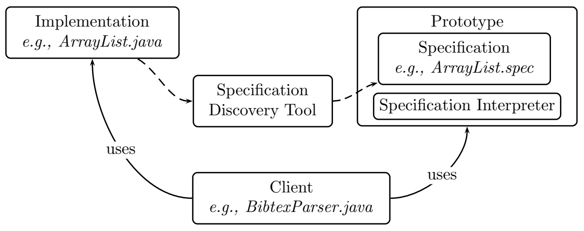


Figure 1. Integrating specification discovery tools and our interpreter

rect answers which indicates a bug in our specification; and (iii) the run fails because the interpreter is unable to produce an answer for a method, which indicates that the specification is incomplete. Notice that debugging specifications is a non-trivial task, especially with realistic classes that require a large number of axioms (e.g., java.util.LinkedList requires more than 100 axioms to completely specify its 39 methods).

Figure 1 shows how our specification interpreter complements our own previous work on algebraic specification discovery [14]. We use our algebraic specification discovery tool to discover a specification (*ArrayList.spec*) from an implementation (*ArrayList.java*). While specification discovery tools are effective in discovering specifications [9, 1, 14], the specifications they produce may be both *unsound* and *incomplete*. This is because the most effective specification discovery tools (to our knowledge) are based on analyzing program runs rather than statically analyzing the code. Our specification interpreter can be used by a programmer to *debug* a discovered specification, which means that a programmer iteratively refines the specification to make it sound and complete within a given context, e.g., within a client that uses the specified class (*BibtexParser.java*). This approach for addressing unsoundness and incompleteness is complementary to Nimmer and Ernst [18], who address unsoundness for a subset of discovered invariants by using the static checker ESC/Java for validation. We demonstrate the applicability of our tools for the scenario shown in Figure 1 in a case study (Section 5.2).

The rest of the paper is organized as follows. Section 2 describes our algebraic specification language. Section 3 describes the design of our algebraic interpreter. Section 4 presents our algebraic rewriting engine. Section 5 reports on our experience with our approach and our tools. In particular, we report on scenarios in which we used our approach to develop algebraic specifications. Section 6 discusses related work and Section 7 concludes.

2. Our Algebraic Specification Language

Invoking a Java method has seven possible consequences: The method may (i) return a value, (ii) throw an exception, (iii) modify the receiver ("this"), (iv) terminate the program, (v) modify objects reachable from arguments, (vi) modify other objects, reachable from instance variables or static variables, or (vii) modify resources external to the program. (i)-(v) are easy to express in algebraic specifications, while (vi) and (vii) are awkward at best. Since (i)-(iii) are the most common, and from a software engineering viewpoint are the most desirable, our language supports only these. (iv) is trivial. Elsewhere we describe how to extend our language to support (v) [14]. We do not yet know of a good way to address (vi) or (vii).

Algebraic specifications have two parts: an *algebraic signature* and a set of *axioms* [17]. The algebraic signature itself has two parts: *sorts* and *function types*. Intuitively, sorts give the types of interest to the algebra. Function types are the operations from which terms of the algebra are constructed. Equational axioms equate terms in the algebra. Specifications written in our specification language mirror this structure by having the three parts (sorts, function types, and axioms) as follows.

First, the specification file enumerates the sorts. For example, in Java terms, the sorts are the classes that are used by the axioms. For each class, the user may optionally specify a concrete existing implementation for that type. This feature is useful if a programmer wants to experimentally check if a specification matches a real implementation. Primitive types in Java are implicitly sorts of the algebra. For example, the sorts for our linked list specification are as follows:

```
class LinkedList is java.util.LinkedList
class NoSuchElementException is
                              java.util.NoSuchElementException
class Object is java.lang.Object
```

In other words, there are three sorts (*LinkedList*, *NoSuchElementException*, and *Object*). This specification fragment also references real existing implementations (e.g., *java.util.LinkedList*) of the sorts which can be checked against the specification.

Second, the specification file enumerates the function types. For example, one of the function types for our linked list specification is as follows:

```
method add is
        <java.util.LinkedList: boolean add(java.lang.Object)>
```

This function type says that the *add* function is defined on a *LinkedList* and takes an *Object* as its argument. Its return type is boolean (to indicate whether or not the *add* was successful). Note that rather than inventing new syntax, we have tried to use Java syntax as much as possible. We borrow the

Soot syntax for fully qualified names of Java classes and methods [22].

Third, the specification file gives the equational axioms. For example, consider the following two axioms from our linked list specification. The first argument to each operation is the receiver object.

```
forall l:LinkedList forall o:Object                    (Axiom 1)
    removeLast(add(l,o).state).retval == o
forall l:LinkedList forall o:Object                    (Axiom 2)
    removeLast(add(l,o).state).state == l
```

Note the *.retval* and *.state* qualifications. These correspond to the return value of an operation and the (possibly modified) state of *this* after an operation, respectively. Both axioms are universally quantified over all linked lists and all objects. Axiom 1 states that invoking *removeLast* after an *add* returns the value that was last added. Axiom 2 states that if after adding an element to a linked list, *l*, one invokes a *removeLast*, the *this* is modified to be *l* (i.e., what it was before the *add*).

Axioms may be conditional. For example, consider:

```
axiom forall l:LinkedList forall x:Object forall i:int     (Axiom 3)
    if i>=0 then get(addFirst(l,x).state,intAdd(i,1).retval).retval
                   == get(l,i).retval
```

This axiom defines the semantics of the *get* operation in terms of *addFirst*. *get* returns the ith element in the linked list. The basic idea is to traverse down the list while decrementing i as long as $i \geq 0$. *intAdd* performs integer addition. Our system is preinitialized with axioms pertaining to *intAdd*.

3. Approach

Our approach provides a seamless integration between an interpreter for algebraic specifications and Java applications. From the perspective of the developer, there is no difference (except for performance) between calling a Java method and interpreting an algebraic specification that describes the method. This property of our approach means that application developers get a prototype implementation of their classes for free when they develop algebraic specifications for their classes. Once developers get some experience with the prototype, they can replace it with a hand-coded (and probably faster) implementation. Our approach also helps in the testing and development of the hand-coded implementation by providing an option for continuously validating the hand-coded implementation against the algebraic specification for the implementation. Thus, our approach can immediately detect when an implementation deviates from its formal specification.

Figure 2 illustrates the architecture of our system. From the *user supplied* parts we see that the user provides two kinds of input to our system: *Specification Components*, which are the algebraic specification parts of the input, and *Java Applications*, which are the Java parts of the input. The *Algebraic Specifications* are specifications in the language described in Section 2. The *Simulation Set* is the set of classes that are to be simulated by our specification interpreter. For example, if a programmer wants to use a specification for a *LinkedList*, the simulation set would contain only *LinkedList*, and the algebraic specification would specify the behavior of *LinkedList*.

In addition to the algebraic specifications and the simulation set, users of our system also provide a client that uses the classes in the simulation set (*simulation client*). Optionally, users may also provide *simulation subjects* which are real existing implementations of the classes to be simulated. These classes are actually specified as part of the specification (Section 2). If a user provides these classes, our interpreter continuously checks the result of the interpretation against these classes (i.e., it runs them in parallel). Thus, this optional component provides a mechanism for dynamically validating a real implementation against an algebraic specification. Furthermore, when algebraic interpretation fails due to an incomplete specification, the interpreter can issue warnings and continue to execute by using results from the simulation subjects.

We use a custom Java class loader to load the simulation client. The class loader uses the bytecode engineering library [3] to redirect references to classes belonging to the simulation set to *simulation stubs*. In other words, once we load the simulation client, it references simulation stubs instead of classes that are in the simulation set. The simulation stubs contain methods with the same signatures as the classes they simulate; however, their bodies delegate all calls to the interpreter. We generate simulation stubs on the fly. For example, consider the following code fragment:

```
LinkedList l1 = new LinkedList();
LinkedList l2 = new LinkedList();
Integer five = new Integer(5);
l2.add(five); l1.addAll(l2);
```

Since *LinkedList* is a member of the simulation set, the class loader replaces all references to LinkedList with references to the simulation stub *SIMSTUB_LinkedList* by manipulating the constant pool of the Java bytecode for the class. We generate the simulation stub, *SIMSTUB_LinkedList*, automatically when we encounter the first reference to *LinkedList*.

```
SIMSTUB_LinkedList l1 = new SIMSTUB_LinkedList();
SIMSTUB_LinkedList l2 = new SIMSTUB_LinkedList();
Integer five = new Integer(5);
l2.add(five); l1.addAll(l2);
```

Following is an example of the *add* method in the simulation stub for *LinkedList*. This stub wraps all arguments into an object array and passes a serialized signature, the arguments, and the receiver object to the interpreter. Finally, it

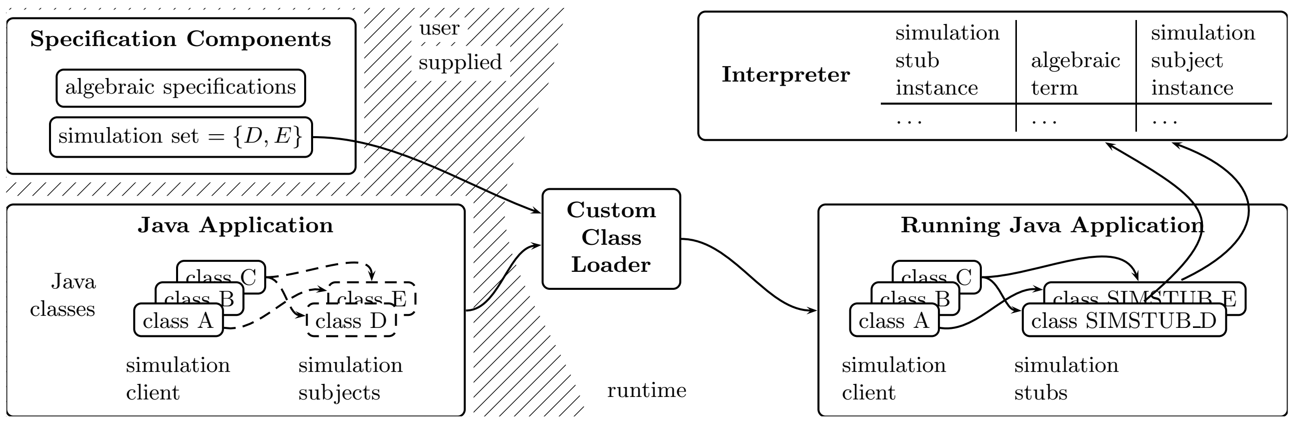


Figure 2. Architecture of our system

unboxes the result of the interpretation into a *boolean*.

public boolean add(Object o){
    return UnboxUtil.unboxBoolean(
        Interpreter.interpret("<LinkedList: boolean add(Object)>",
            new Object[]{o},this); }

For each simulation stub instance (e.g., an object of type *SIMSTUB_LinkedList*), the interpreter maintains both an algebraic term modeling the state of the object and optionally, a simulation subject instance (e.g., an object of type *LinkedList*) (see the Interpreter box in Figure 2). When the simulation client invokes a method on a simulation stub instance, the interpreter extends (and possibly rewrites) the algebraic term associated with that instance. If the programmer has provided simulation subjects, the interpreter also invokes the corresponding method on the simulation subject instance. After executing the code above, the simulation stub instance referred to by *l1* maps to the following algebraic term:

addAll(NewLinkedList().state, ***(Term 1)***
    add(NewLinkedList().state,Integer@3982).state).state

The subterm *Integer@3982* denotes the *Integer* object containing the integer value 5. By applying term rewriting (discussed in detail in Section 4), the interpreter (i) reduces the size of the terms that model the state of an object (ii) computes the return value of the simulated Java methods. As an example for (i), the interpreter uses the axiom

forall o:Object add(NewLinkedList().state,o).state ***(Axiom 4)***
    == addFirst(LinkedList().state,o).state

to transform the algebraic term Term 1 into

addAll(NewLinkedList().state,addFirst(***(Term 2)***
    NewLinkedList().state,Integer@3982).state).state

Next, the axiom

forall l1:LinkedList forall l2:LinkedList ***(Axiom 5)***
    addAll(l1,addFirst(l2,o).state).state
    ==addAll(add(l1,o).state,l2).state

transforms Term 2 into

addAll(add(NewLinkedList().state, ***(Term 3)***
    Integer@3982).state, NewLinkedList().state).state

Next, the axiom

forall l:LinkedList addAll(l,newLinkedList().state).state==l

transforms Term 3 into

add(NewlinkedList().state,Integer@3982).state ***(Term 4)***

As an example for (ii), the interpreter rewrites the term

addAll(NewLinkedList().state, ***(Term 5)***
    add(NewLinkedList().state,Integer@3982).retval

using the axiom

forall l1:LinkedList forall l2:LinkedList forall o:Object
    addAll(l1,add(l2,o).state).retval==true ***(Axiom 6)***

into *true*. Since *true* is a constant, the interpreter can return the constant back to the interpretation stub and the simulation was successful.

Sometimes the algebraic specification may be incomplete, which means that we cannot compute a return value for a particular method application. For example, if Axiom 6 is missing, the interpreter will not be able to produce the return value (*true*) for the term given above. At this point the interpreter reports an irreducible term to the user. If the user has supplied simulation subjects, the interpreter can use the result produced by the simulation subject instance and continue with the interpretation.

4. Algebraic Term Rewriting

Section 3 illustrated how we use rewriting to interpret algebraic specifications. We now discuss rewriting in greater detail, focusing on the challenges that we encountered.

Any given specification language presents a particular tradeoff between analyzability and expressiveness. Languages that are easy to analyze are usually not as expressive or convenient for the programmer or the specifier, yet expressive languages can quickly become too costly to analyze. Our specification language (Section 2) is very expressive, which means that it presents a number of challenges for our interpreter. We start by giving a high-level overview of our use of rewriting and then discuss the challenges that we encountered.

4.1. Overview of Rewriting

Recall that Java clients of our interpretation invoke operations on simulation stub instances. These simulation stub instances take the place of regular objects (e.g., instances of a *LinkedList*) in a traditional Java program. As the client invokes more methods on simulation stub instances, the terms modeling the state of the objects increase in their size. The rewriting engine is responsible for reducing these terms. Rewriting interprets the axioms in the algebraic specification as rewriting rules that transform one term into another. Each axiom in the user-provided specification gives rise to up to two rewriting rules. For example,

```
forall o: Object addFirst(NewLinkedList().state, o).state
    ==add(NewLinkedList().state, o).state
```

gives rise to two potential rewriting rules, namely

```
forall o: Object addFirst(NewLinkedList().state, o).state
    → add(NewLinkedList().state, o).state                          and
forall o: Object add(NewLinkedList().state, o).state
    → addFirst(NewLinkedList().state, o).state
```

However, the axiom

```
forall l:LinkedList forall o: Object add(l,o).retval==true
```

gives rise to only

```
forall l:LinkedList forall o: Object add(l,o).retval → true
```

since we would not have a binding for *l* and *o* if we had a rewriting rule from *true* to *add(l,o).retval.*

Given a term that needs to be reduced, our interpreter works by applying a sequence of rewriting rules. If the reason for reducing a term is to produce an answer to return to the client, our interpreter applies rewriting rules until it ends up with a constant (e.g., a number of a reference to an object). If the reason for reducing a term is to reduce its size, the interpreter can stop whenever it feels that the term is small enough.

Especially in the first case (i.e., reducing a term to produce a value for the client), our interpreter may fail in two ways. First, the interpreter may be unable to find a sequence of rewritings that produce a constant. This case exposes potential incompleteness in the user-provided axioms. Second, the interpreter may be able to reduce the term to an incorrect constant (e.g., it finds *5* instead of *9*). This case exposes an error in one or more of the axioms. In both of the above cases, our system produces a detailed message describing what failed. As we show in Section 5, these diagnostics are invaluable for producing a correct specification or debugging an existing specification.

4.2. Strategies for Algebraic Term Rewriting

To manage the vast search space for term rewritings, we use two strategies.

Our primary strategy is a greedy one that uses only rewriting steps that reduce the size of the term. It does not use backtracking. If the term to be reduced is a *.retval* term, and this strategy is unable to reduce it to a constant, it resorts to the secondary strategy. We do not use the secondary strategy for *.state* terms because reducing *.state* terms is a performance optimization and not strictly necessary. Thus, we use the secondary strategy only when we absolutely need it.

Our secondary strategy tries all rewriting steps that do not grow the term. If any of these rewriting steps lead to a term that can be reduced in size via a rewriting step, we revert back to the primary strategy. Note that this strategy uses backtracking and is thus much more expensive than the primary strategy.

Even our secondary strategy may be unable to reduce a term if, for example, it is necessary to increase the size of the term before it can be ultimately reduce. Our current implementation does not check the set of rewriting rules for *confluence* [6] or for consistency, which means: (i) it may allow a term to be reduced to two distinct constants; and (ii) it may not find the desirable rewriting sequence, even though it only consists of steps that make the term smaller.

The set of strategies that we have chosen affects the capabilities and the efficiency of our system. While we believe that the strategies we have added to our interpreter make sense in practice, there is still a lot of room for experimentation.

4.3. Conditional Axioms

Conditional axioms lead to additional complexity in the algebraic specification interpreter. Consider Axiom 3, which we explained at the end of Section 2:

```
forall l:LinkedList forall x:Object forall i:int
    if i>=0 then get(addFirst(l,x).state,intAdd(i,1).retval).retval
        == get(l,i).retval
```

For this kind of algebraic axiom (or the corresponding rewriting rule from left to right) we simply make sure that the constraints between *if* and *then* are fulfilled whenever we unify the left side of the axiom with a term. Sets of axioms allowing this kind of constraints, i.e. a set of simple relations between variables and constants, are called a *semi-equational system* in the literature [20].

Our system also allows the more complex *join systems* [20]. A join system allows conditional axioms with arbitrary terms in the condition. For such axioms we need to use the rewriting system to also determine the value of the condition (*true* or *false*). While this all seems straightforward, it can lead to infinite recursion. Furthermore, we find that the debugging trace for a join system can become hard to digest since deeply nested sequences of constraints, checks, and rewriting attempts are common. We feel that join systems, despite their increased complexity over semi-equational systems, are worth it: they often allow more elegant expression of behavior than semi-equational systems. For example, the following axiom uses the *contains* operation in a constraint to say that, if the hash set *h* already contains *o*, the size of *h* will not change if we add *o* again. This same axiom is much harder to write in a semi-equational system.

```
forall h:HashSet forall o:Object
    if contains(h,o).retval==true then
    size(add(h,o).state).retval == size(h).retval
```

To see how this axiom can be used as a rewriting rule, consider rewriting the term

```
size(add(add(NewHashSet().state,Object@1234
        ).state,Object@1234).state).retval
```

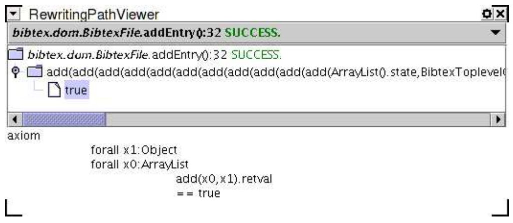
First, we note that without considering the condition in the axiom, the left side of the axiom unifies with the term with the unification mapping $m = \{ h \mapsto$ *add(NewHashSet().state,Object@1234).state*, $o \mapsto$ *Object@1234*$\}$. However, before we can apply the rewriting, we need to determine if the condition is *true*. We apply *m* to the condition to get: *contains(add(NewHashSet().state,Object@1234).state,Object@1234).retval ==true*. Using the axioms for the *contains* operation (omitted for brevity), the algebraic interpreter will reduce this relation by rewriting it to *true==true*. Thus, the check succeeds and the original rewriting rule can now be applied, yielding *size(add(NewHashSet().state,Object@1234).state).retval*.

4.4. References to External Methods

Sometimes the specification of one class may need to reference methods from a class outside the simulation set. For example, when writing the specification for a hash set's *add* method, we would like to write:

```
forall h:HashSet forall o1:Object forall o2:Object
    if equals(o1, o2).retval==true then
        contains(add(h,o1).state,o2).retval==contains(h,o2).retval
```

However, this axiom uses the *equals* method of *o1* which is not part of the specification of a hash set. Similar problems arise when writing specifications for an iterator. There are two ways of addressing this problem: (i) Include the specification of *equals* in the specification for hash set; (ii) Extend the specification language to allow calls to Java methods, such as *equals*. The first approach, while seemingly more elegant than the second approach, has one disadvantage: it forces us to specify the behavior of *equals* for all possible



```
bibtex.dom.BibtexFile.addEntry():32 SUCCESS.
bibtex.dom.BibtexFile.addEntry():32 SUCCESS.
  add(add(add(add(add(add(add(add(add(add(add(ArrayList().state,BibtexToplevel(
    true
axiom
        forall x1:Object
        forall x0:ArrayList
                add(x0,x1).retval
                == true
```

Figure 3. User interface for rewriting engine

objects that could be added to a HashSet. Generic containers in the Java language will make this approach more viable, but even with generics, dynamic class loading can load new subclasses for which the behavior of *equals* is different than any given specification. Our current prototype supports both the first and the second solution: One can declare that an operation as *external* which means that whenever the interpreter encounters a term in which all parameters are constants, the Java implementation for the method is evaluated. For example, suppose that *equals* has been declared an *external* method. When the interpreter encounters *equals(Object@1423,Object@1111).retval* it will execute the appropriate *equals* implementation before resuming algebraic interpretation. This mechanism is also useful for extending the interpreter with arithmetic and helper functions.

4.5. Debugging Support

When the specification is incomplete, the interpreter prints the irreducible term, which provides a starting point for manually completing the specification. In some cases, it is useful to also examine the trace provided by the rewriting engine. This trace records all rewriting operations that take place. By searching through this trace, the user can find out whether or not a particular rewriting rule has been applied and which intermediate terms have been generated in the interpretation process. We also use this trace to debug our interpreter: The rewriting engine prints a counter value into the trace for each rewriting step. When we find suspicious activity in the rewriting trace, we used a conditional breakpoint in a Java debugger to jump to the execution of the rewriting step in question.

As an alternative for examining the rewriting trace, we developed a user interface for the rewriting engine as shown in Fig. 3. Using the drop down menu at the top of the window, the user selects which rewriting computation to view. Below, a tree view shows how each term is a reduction of its parent by using one rewriting step. When a user selects a term in the tree view, the viewer displays the axiom that generated the selected term from its parent in the text area at the bottom of the window.

5. Experience

We describe two scenarios in which we applied our algebraic specification interpreter. Section 5.1 gives an example of developing a specification from scratch. Section 5.2 shows how we used axioms generated by our specification discovery tool [14] and then debugged the specification using a client application. Section 5.3 provides evidence that the prototype generated by our tool from the specification has acceptable performance to be usable for many applications.

5.1. Extreme Specifying: A Case Study

Programmers can use our system to incrementally develop a specification (and thus a prototype) based on the needs of the code that they are developing. For example, when developing a Java class (“client”) the programmer may not know all the requirements on classes that it uses (“helpers classes”). Thus it would be premature to develop the full specification of a helper class before writing the client. On the other hand, the programmer cannot develop and test the client before writing a prototype of the helper. Our tool helps in this dilemma by allowing a programmer to develop a specification and prototype of the helper class as needed by the client. This section presents an example where a programmer develops a specification and prototype of a hash set hand-in-hand with the client of the hash set.

The programmer starts by writing the client:

```
1 class Client {
2   public static void main(String args[]){
3     Integer one = new Integer(1);
4     HashSet s = new HashSet();
5     s.add(one);
6     System.out.println(”test 1 = ”+s.contains(one));}}
```

At this point, the programmer sees that the client needs a hash set, which must support the methods *add* and *contains*. Thus, the programmer creates the following incomplete specification:

```
specification HashSetSpecification
class HashSet
method NewHashSet is <void <init>()>
method add is <boolean add(java.lang.Object)>
method contains is <boolean contains(java.lang.Object)>
define HashSet
```

Note that the specification also includes a *NewHashSet* operation for creating a new hash set. Also note that the programmer starts with an empty set of axioms (i.e., there is nothing under the *define HashSet*) directive. In other words, the interpreter can build up the terms but has no rewriting rules to reduce them. When the programmer gives the “Client” class and the specification to the interpreter, the interpreter responds with:

```
Client.java, line 5: Algebraic Interpreter failed to compute a value.
term = add(NewHashSet().state,Integer@1776).retval
Client.java, line 6: Algebraic Interpreter failed to compute a value.
term = contains(add(NewHashSet().state,Integer@1776
                    ).state,Integer@1776).retval
```

The first error message says that the interpreter could not determine the return value of the invocation *s.add*. The second error message complains about not being able to produce a return value for *s.contains*. To eliminate these error messages and to compute the expected result, the programmer adds the following axioms:

```
forall o:Object add(NewHashSet().state,o).retval          (Axiom 7)
        ==true
forall o:Object forall h:HashSet                          (Axiom 8)
        contains(add(h,o).state,o).retval==true
```

The first axiom says that adding any object to a new hash set returns *true*. Note that this is inadequate in general since it does not say anything about adding to a non-empty HashSet. The second axiom says that immediately after adding an object to the HashSet, invoking *contains(add(h,o).state,o).retval* returns *true*. This axiom too is limited since *contains* returns true only if the element being checked was the last one added. With these two axioms, the client runs successfully.

The programmer now continues implementing the client and adds *System.out.println(”test 0 = ”+s.contains(one));* immediately before Line 5. Since this statement invokes a *contains* on an empty hash set, the programmer also remembers to add this axiom:

```
forall o:Object                                           (Axiom 9)
        contains(NewHashSet().state,o).retval==false
```

On running the modified client and specification set, our system gives the following error message:

```
test 0 = true
Client.java, line 6: Algebraic Interpreter failed to compute a value.
term = add(contains(NewHashSet().state,
                    Integer@7905).state,Integer@7905).retval
```

The problem is that the programmer forgot to specify how *contains* affects the state of the object. This mistake is easy for programmers to overlook since they are primarily thinking in terms of what *contains* does and not what it does not do. The debugging output of our tool, which prints all rewriting attempts and intermediate terms (too verbose to include in this paper), can also come in handy at this point to find what is missing from the axioms. Since *contains* does not modify the state of the set, all we need to add is the following axiom:

```
forall h:HashSet forall o:Object contains(h,o).state==h
```

After this new axiom, the client executes successfully. Needless to say, the specification of a hash set is still far from complete. As the programmer adds more behavior to the client class, our interpreter exposes more of the limitations of the specification. Ultimately, this iterative process may lead to a complete specification of the hash set. It is worth noting here that the quality of the test client is key to debugging the algebraic specification. Thus, once a programmer has finished developing the client (and thus the specification), it is worthwhile to generate more clients for the hash set with the intention of "testing" the specification of the hash set.

5.2. Debugging a Discovered Specification

In this case study, we used the specification discovery tool [14] to generate a specification for the *java.util.ArrayList* class contained in Sun's Java Development Kit. We then used the algebraic interpreter to debug the discovered specification. Our client application is a BibTeX parser.[1] We chose this client application because it is not dependent on libraries other than the Java standard libraries, it uses collection classes, and we were familiar with the code.

Similar to what we describe in Section 5.1, debugging the discovered specification is an iterative process consisting of three steps: (i) using the specification interpreter to run the client application, (ii) understanding the debugging output, (iii) adding new algebraic axioms to the specification or modifying the existing axioms.

Out of the 10 algebraic axioms to execute the BibTeX parser successfully, our discovery tool can produce 3 axioms exactly as needed. As an example, the following two axioms specify how the first element of an *ArrayList* can be obtained by applying the *get* operation for index 0:

forall x0:Object ***(Axiom 10)***
    get(add(newArrayList().state,x0).state,0).retval == x0
forall l:ArrayList forall o1:Object forall o2:Object ***(Axiom 11)***
    get(add(add(l,o1).state,o2).state,0).retval
    ==get(add(l,o1).state,0).retval

We manually added 7 axioms to the specification. Five of those axioms describe the behavior of *Iterator* instances generated by *ArrayList* objects. For example, the following axiom states that an iterator created from an empty list does not have a next element:

hasNext(iterator(ArrayList().state).retval).retval==false

The discovery tool currently cannot find these 5 axioms because the state of the *Iterator* object is modeled as the return value of the operation *iterator()* of another class (*ArrayList*). This scenario is not covered by the currently implemented equation generators. However, the discovery tool provides extension points for adding new equation generators. An appropriate equation generator can be implemented without changing the infrastructure.

1 Available at `www.cs.colorado.edu/~henkel/stuff/javabib/`.

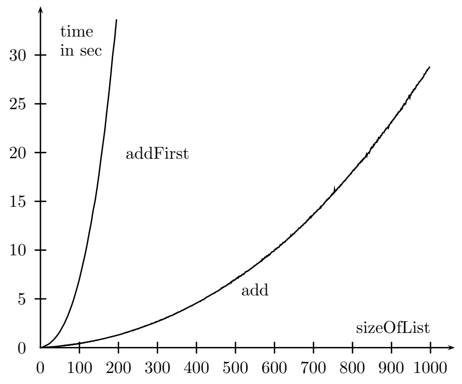


Figure 4. Term Rewriting Benchmark

Adding four of the axioms which describe the behavior of *Iterator* was straight forward. The following axiom was more involved:

forall l:ArrayList
    next(iterator(l).retval).state
    ==iterator(remove(l,0).state).retval

This axiom describes how the next operation applied to an iterator transforms the iterator's state. Unfortunately, if this axiom was used as a left to right rewriting rule, it would increase the size of the term. Thus, our interpreter will not use it (see Section 4.2). However, our interpreter allows hidden operations, which can be used in rewriting rules, but are externally invisible [21]. We introduced a hidden operation *removeFirst*, which eliminates the problem:

forall l:ArrayList ***(Axiom 12)***
    next(iterator(l).retval).state
    ==iterator(removeFirst(l).state).retval

The two remaining axioms we had to add describe the behavior of the hidden operation *removeFirst*. The specification discovery tool can find variations of both axioms which use *remove(_,0)* instead of *removeFirst(_)*. For example, it found

forall x0:Object
    remove(add(ArrayList().state,x0).state,0).state
    == ArrayList().state

ArrayList has a large number of operations, which means that many axioms are needed to fully document it. By using the specification interpreter, we focused on the axioms needed for a particular run of our client application. In other words, understanding the 10 executed axioms of our specification is enough for understanding the behavior of *ArrayList* for the particular run. Thus, the 10 executed axioms can be considered a dynamic slice of the specification.

We describe the full case study elsewhere [15].

5.3. Performance

To evaluate the performance of our rewriting engine, we use the following benchmark, which is parameterized with *sizeOfList*.

```
Object o = new Object();
LinkedList l = new LinkedList();
for (int i = 0; i < sizeOfList; i++) l.add(o);
l.get(sizeOfList−1);
```

This benchmark creates a linked list with *sizeOfList* elements (line 3) and then retrieves the last element (line 4). In Figure 4, we plot the time it takes for the rewriting engine to compute the result value of the *get* method call for line 4 in the benchmark (y-axis) for different values of *sizeOfList* (x-axis). We measure the execution times on a Dell PowerEdge 600SC Pentium 4 2.4 Ghz with 2 GB of RAM running Sun's JDK 1.4.2 on SuSE Linux 8.1.

We present data for two different specifications of the *get* method. The *get* method returns an element at a particular position in the list (counting from the first element). The *add* and *addFirst* add an entry to the end and beginning of the list, respectively. Axiom 13 and Axiom 14 make up the first specification of *get* and Axiom 15 and Axiom 16 make up the second specification of *get*:

```
forall l:LinkedList forall o:Object                          (Axiom 13)
    get(addFirst(l,o).state,0).retval==o
forall l:LinkedList forall o:Object forall i:int             (Axiom 14)
    if i>=0 then get(addFirst(l,o).state,intAdd(i,1).retval).retval
        == get(l,i).retval
forall l:LinkedList forall o:Object forall i:int             (Axiom 15)
    if size(l).retval == i then get(add(l,o).state,i).retval == o
forall l:LinkedList forall o:Object forall i:int             (Axiom 16)
    if size(l).retval > i then
        get(add(l,o).state,i).retval == get(l,i).retval
```

The main difference between the two specifications is that the first one expresses *get* in terms of *addFirst* while the second one expresses *get* in terms of *add*. Given our simulation client, we would expect the second to be a better match because the client also builds up the list in terms of *add*. More specifically, if we use the first specification of *get*, our rewriting engine will first have to rewrite the term that corresponds to the entire linked list in terms of *addFirst* before it can start to reduce it.

Our results (Figure 4) confirm the intuition above. The horizontal axis of Figure 4 gives the *sizeOfList* parameter and the vertical axis gives the time in seconds to execute line 4 of the benchmark. The *addFirst* and *add* curves give the execution times for the two specifications for different values of *sizeOfList*. We see that the specification that matches the simulation client is faster than the specification that does not match the simulation client. In future work we plan to implement memoization techniques. Thus, subsequent invocations of *get* will be able to reuse much of the work of rewriting the list term to use *add* instead of *addFirst*.

There are two points to take away from this data. First, while the prototype implementation produced by our system is much slower than a hand-coded implementation (e.g., executing the benchmark for *sizeOfList=1000* with the official JDK implementation takes less than 1 millisecond), it may still be fast enough to be used for prototyping. Second, some specifications may execute much faster than other (equivalent) specifications, depending on the match between the specification and the simulation client.

6. Related Work

Previously [14] we described a system that can discover algebraic specifications automatically from Java classes. The output of that system can be used as a starting point for developing a specification of an existing Java class. The current paper and our previous paper share the goal of making formal specification techniques more appealing for practical use. Both techniques use the same specification language and are designed to be used together.

There is a vast body of prior work on term rewriting systems [6, 20]. Prior work has also studied the idea of using term rewriting to simulate a software component. For example, Wang and Parnas proposed the trace rewriting method to simulate software modules [23]. However, they focus on the rewriting technique for their system and unlike us, do not integrate their system into a programming language or provide details of an implementation. Implementations of other rewriting engines and rewriting language have been used to provide prototyping [10, 7, 20], but again, to our knowledge, they do not interact with a client written in a modern programming language. Thus, these systems do not provide the software engineering benefits that our approach offers. Antoy and Hamlet [2] propose self-checking ADTs, which integrate rewriting into C++ and Java classes. Among other details, our system differs by (i) fully automating the integration of Java code and the algebraic interpreter with a custom class loader, and (ii) a more expressive algebraic specification language that has been customized for being embedded into Java (e.g., we allow operations to both modify the state of an object and return a value). Antoy and Hamlet manually implement representation mappings as C++/Java functions to allow intensional comparisons, which might be a useful addition to our current system.

Other previous work uses algebraic specifications as assertions to check whether implementations are consistent with a given specification [11, 16, 8, 5, 4, 19]. Some of these systems require test drivers to be written (e.g. [11]), others generate test cases by themselves from the algebraic specifications [8, 5, 4]. Sankar [19] uses a theorem prover to determine which of the algebraic terms generated by a running program need to be equivalent and then checks whether the implementation implements the equivalences correctly. While some of these systems interact with real implementation languages, our system is different in that it (i) seamlessly integrates with a real implementation language by exploiting reflection and dynamic class loading in Java; and (ii) automatically constructs a prototype from an algebraic specification.

7. Conclusion

We describe the design, implementation, and usage of an interpreter for algebraic specifications that is seamlessly integrated with Java. The goal of the system is to make algebraic specifications more cost effective and easier to write and debug. Our tool creates a prototype implementation of a class from its algebraic specification. A Java program can use this prototype implementation just like any hand-coded implementation of the class.

Our approach helps in writing and debugging algebraic specifications because programmers can now execute their specifications and optionally compare the execution of the specification to a hand-coded implementation. Executing the specifications exposes both errors and missing axioms in the specifications. We illustrate the usefulness of this approach by giving case studies and by presenting performance results for the prototype produced by our tool.

Acknowledgments We thank the members of CU Boulder's programming languages group, the members of CU Boulder's software engineering research laboratory, members of the software technology department at IBM Research, and the anonymous referees for listening to our ideas and giving great feedback.

References

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1), Jan. 2000.

[3] Apache Software Foundation. BCEL—byte code engineering library. http://jakarta.apache.org/bcel/.

[4] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object oriented programs. *ACM Transactions on Software Engineering*, 7(3), July 1998.

[5] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering*, 10(4):56–109, Jan. 2001.

[6] N. Dershowitz and D. A. Plaisted. *Handbook of Automated Reasoning*, volume 1, chapter Rewriting. Elsevier, 2001.

[7] N. Dershowitz and L. Vigneron. Database of rewriting systems. http://www.loria.fr/ vigneron/RewritingHP/systems.html, 2003.

[8] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering*, 3(2), Apr. 1994.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *ACM Transactions on Software Engineering*, 27(2):1–25, Feb. 2001.

[10] K. Futatsugi. CafeObj official homepage. http://www.ldl.jaist.ac.jp/cafeobj/, 2003.

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

[12] D. Gries. *The science of programming*. Texts and monographs in computer science. Springer-Verlag, 1981.

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

[14] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[15] J. Henkel and A. Diwan. Case study: Debugging a discovered specification for java.util.arraylist by using algebraic interpretation. Technical Report CU-CS-970-04, University of Colorado at Boulder, 2004.

[16] M. Hughes and D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *Proceedings of the International Symposium on Software Testing and Verification*, San Diego, California, 1996.

[17] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[18] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 2001.

[19] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, Victoria, British Columbia, Canada, Sept. 1991.

[20] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[21] J. W. Thatcher, E. G. Wagner, and J. B. Wright. Data type specification: Parameterization and the power of specification techniques. *ACM Transactions on Programming Languages and Systems*, 4(4), Oct. 1982.

[22] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[23] Y. Wang and D. L. Parnas. Simulating the behavior of software modules by trace rewriting. *ACM Transactions on Software Engineering*, 20(10), Oct. 1994.