Developing and Debugging Algebraic Specifications for Java Classes

University of Colorado Technical Report CU-CS-984-04*

Johannes Henkel and Christoph Reichenbach and Amer Diwan
Department of Computer Science, University of Colorado at Boulder

December 12, 2004

Abstract

Modern programs make extensive use of reusable software libraries. For example, a study of a number of large Java applications shows that between 17% and 30% of the classes in those applications use container classes defined in the `java.util` package. Given this extensive code reuse in Java programs, it is important for the interfaces of reusable classes to be well documented. An interface is well documented if it satisfies the following requirements: (1) the documentation completely describes how to use the interface; (2) the documentation is clear; (3) the documentation is unambiguous; and (4) any deviation between the documentation and the code is machine detectable. Unfortunately, documentation in natural language, which is the norm, does not satisfy the above requirements. Formal specifications can satisfy them but they are difficult to develop, requiring significant effort on the part of programmers.

To address the practical difficulties with formal specifications, we describe and evaluate a tool to help programmers write and debug algebraic specifications. Given an algebraic specification of a class, our interpreter generates a prototype which can be used within an application just like any regular Java class. When running an application that uses the prototype, the interpreter prints error messages that tell the developer in which way the specification is incomplete or inconsistent with a hand-coded implementation of the class. We use case studies to demonstrate the usefulness of our system.

1 Introduction

Modern software is made up of many components. For example, Table 1 shows the number of classes (# classes) and the number of modules (# jar files) for a

*Author's address: Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0430, USA. This work is supported by NSF grants CCR-0085792, CCR-0133457, and CCR-0086255. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Table 1: Modularity in large Java projects.

	# jar files	# classes
Jedit 4.1	3	644
Xalan J 2.5.2	9	2,395
Jakarta Velocity 1.1b1	21	2,610
Apache Tomcat 5.0.19	51	3,959
Eclipse 3.0-M7	147	21,774
JBoss 4.0.0DR2	214	32,820

representative selection of Java-based open source projects. To enable programmers to modify these systems and to fully understand the impact of their modifications, it is necessary for the interfaces[1] to be well documented. By "well documented" we mean that each interface's documentation fully describes how to use the interface, and the documentation is unambiguous, clear, and correct (i.e., correctly describes the implementations of the interface).

These criteria for "well documented" are hard to meet with informal documentation. Informal documentation does not, in particular, allow us to automatically check that the documentation is clear, unambiguous, and in sync with the implementation. In practice, it takes significant and persistent effort on the part of programmers to write and maintain good documentation, especially for evolving systems.

Formal specifications, on the other hand, are unambiguous and clear. For some kinds of formal specifications it may also be possible to check that an implementations adheres to a specification. Despite the advantages of formal specifications, programmers rarely use them for the following reasons:

1. Programmers are reluctant to learn new languages, especially if there is no immediate gratification for doing so. With current technology, investing into formal specifications pays off late in the development cycle.

2. It is difficult to write a correct formal specification.

3. Maintaining a mapping between the formal specification and an evolving implementation is difficult, especially if concepts expressed in the specification language do not directly correspond to concepts expressed in the implementation language (a typical example for this are side effects).

We describe and evaluate a novel approach that is a step toward addressing the above difficulties with formal specifications. For our implementation and analyses, we chose Java as underlying language, due to its popularity and the presence of an extensive standard library, but also because of its support for reflection and user-defined class loaders (Section 4). Since container classes Java programs heavily

[1] By "interfaces", we mean the abstract notion of "interfaces" which includes but is not limited to interface types in Java.

Table 2: Container class usage in large Java projects.

	# classes	# classes using containers from java.util
Jedit 4.1	644	123 (19.1%)
Xalan J 2.5.2	2,395	398 (16.6%)
Jakarta Velocity 1.1b1	2,610	780 (29.9%)
Apache Tomcat 5.0.19	3,959	1,084 (27.4%)
Eclipse 3.0-M7	21,774	4,757 (21.8%)
JBoss 4.0.0DR2	32,820	7,888 (24.0%)

reuse container classes (Table 2), we focus on documenting these. More specifically, we use a particular style of formal specifications, algebraic specifications, which are especially well suited for describing container classes [GH78].

Our approach has two components. First, our *specification language* is a notation for writing algebraic specifications that is tailored to modeling Java classes. Programmers can document their classes and methods using this language. Compared to previous work, our language simplifies mappings between Java classes and methods and algebraic sorts and operations.

Second, the *specification interpreter* takes an algebraic specification for a class and a client for the same class, and executes this client using interpretation to simulate the behavior of the specified class. Thus, given an algebraic specification, the system automatically provides an implementation for the specified class, providing instant gratification for programmers. The interpreter can also help in debugging specifications and testing specifications and class implementations against each other. Our tool interprets algebraic specifications using term rewriting, which is a well studied area [DP01, TeR03]. However, to our knowledge our system is the first to seamlessly integrate fully automatic algebraic rewriting techniques with Java classes.

We describe four scenarios in which our tool can help developers in producing documentation in a formal specification language. We demonstrate that our approach works by applying it to a number of case studies. We also report on the run time performance of the interpreter.

This paper improves and extends upon the presentation, ideas, and experimental results presented in our previous work [HD04b].

Section 2 describes four usage scenarios for our system. Section 3 describes our specification language. Section 4 describes our Java-embedded algebraic specification interpreter. Section 5 provides a performance evaluation and case studies. Section 6 discusses related work and Section 7 concludes.

2 Usage Scenarios

We now describe four ways of using our system.

Figure 1 describes the *extreme specifying scenario*, which is inspired by extreme programming. In this scenario, the developer evolves specification and

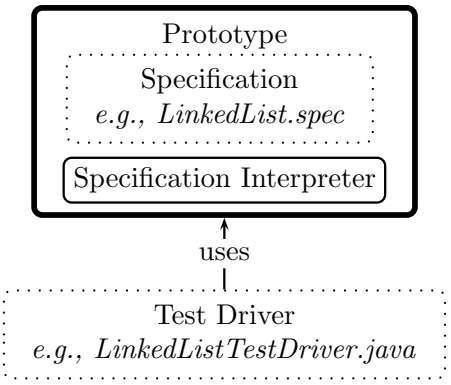

Figure 1: Extreme specifying.

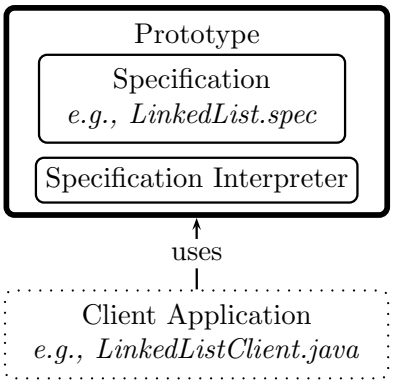

Figure 2: Rapid prototyping.

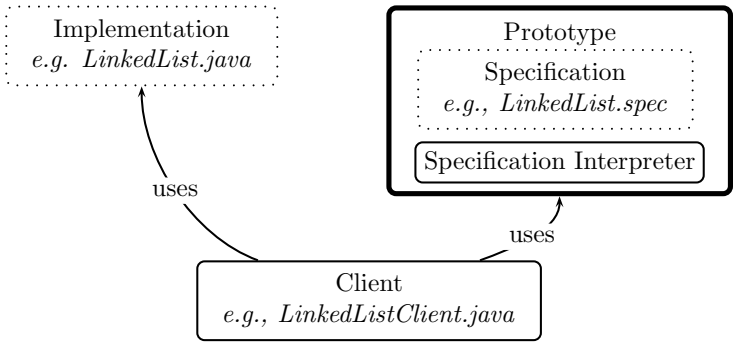

Figure 3: Validating an Implementation.

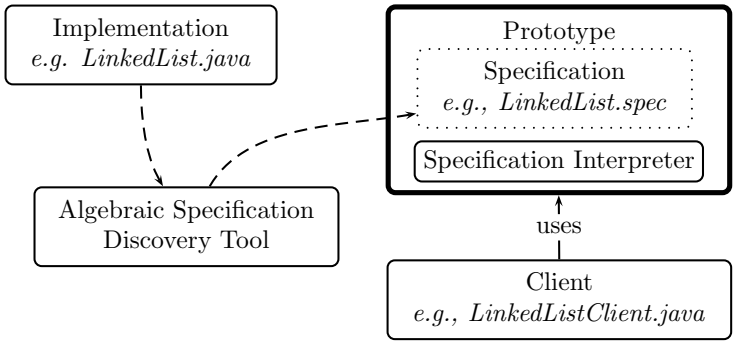

Figure 4: Discovering and Debugging Specifications.

test drivers hand in hand. To construct a complete specification and a complete unit test, the developer repeats these steps: (i) Extend the test driver (`LinkedListTestDriver.java`, in the example); (ii) run the test driver; (iii) if the interpreter fails during the run, use its output to identify and fix inaccuracies and omissions in the specification (e.g., `LinkedList.spec`).

In the extreme specifying scenario, our approach can make developing specifications easier since our system provides support both for incrementally developing specifications and for testing incomplete specifications.

Figure 2 describes the *rapid prototyping scenario*, which looks similar to Figure 1. For this scenario, however, we assume that the specification (`LinkedList.spec`) is complete. A client (`LinkedListClient.java`) can then use the specification just like any regular Java class, except that the performance will usually be inferior to a Java implementation. This means that client code can be tested earlier in the development process.

In the rapid prototyping scenario, our approach can give programmers instant gratification when they write specifications: Once written, a specification also provides a prototype that can be used to aid in the development of other modules.

Figure 3 describes the *validation scenario*, in which the specification interpreter validates an implementation within a particular context. Without changing the source code of the client (`LinkedListClient.java`), the interpreter can execute an algebraic specification (`LinkedList.spec`) and an existing implementation for a class (`LinkedList.java`) at the same time. The interpreter can then report any discrepancy in the behavior between implementation and specification.

In the validation scenario, one can use our approach to keep a specification and an implementation in sync.

Figure 4 describes how we combine our interpreter with our work on discovering specifications to *discover and debug algebraic specifications* for existing implementations. Our dynamic specification discovery tool [HD03] generates a potentially unsound and incomplete specification by observing the behavior of objects. The specification can then be debugged with the algebraic interpreter.

In this scenario, our approach helps in developing specifications for a class that already exists: The discovery tool finds an incomplete specification (which is a good starting point) and the interpreter helps the programmer to refine the discovered specification.

3 An Algebraic Specification Language

We designed our language with the following goals:

- The specification language should be as close as possible to the the Java programming language, without giving up the advantages of algebraic specifications. More specifically, (i) it should be straightforward (automatic, where possible) to map Java signatures and sorts to algebraic signatures and types

and (ii) the axioms should use a Java-like syntax whenever possible. Previous languages and tools require user defined mappings between specifications and implementations (e.g., [HS96]).

- Mappings from Java classes to algebras should represent methods with both side effects and return values. This extends upon previous work, which generally assumes that a method invocation either has no side effects or yields no return value.

3.1 Scope of the 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")[2],

(iv) modify objects passed as arguments[2],

(v) modify objects pointed to by static variables[2],

(vi) modify resources external to the program, or

(vii) terminate the program.

Our language models (i)-(iv). Since it is an algebraic language, it models the *observable state* of objects, which abstracts away from the implementation of objects. Our language does not model shared state ((v) and (vi)), since this is too cumbersome to express with algebraic specifications. Expressing (vii) is trivial. To our knowledge, no solution exists for modeling (v) and (vi) algebraically without significantly increasing the complexity of the language or the specifications.

3.2 Definition of the Algebraic Specification Language

Algebraic specifications have two parts: an *algebraic signature* (e.g., lines 2-6 in Figure 6) and a set of *axioms* [Mit96] (e.g., lines 8-11 in Figure 6). The algebraic signature itself has two parts: *sorts* (e.g., lines 2-3 in Figure 6) and *operations* and their signatures (e.g., lines 4-6 in Figure 6). Intuitively, sorts give the types of interest to the algebra. Operations are used to construct the terms of the algebra. The axioms equate terms in the algebra (e.g., lines 8-11 in Figure 6). We now describe the parts of specifications written in our language in more detail.

3.2.1 Specification Name

The developer starts a specification with the `specification` keyword, followed by a name for the specification. This name is for documentation purposes only. For example, the specification shown in Figure 6 has the name "ObjectStackSpecification" (line 1, Figure 6).

[2]This includes objects reachable via indirection.

```
1  package edu.colorado.cs.simpleadts;
2
3  public class ObjectStack {
4
5      private Object [] store;
6      private int size;
7      private static final int INITIAL_CAPACITY=10;
8
9      public ObjectStack(){
10        this.store = new Object[INITIAL_CAPACITY];
11        this.size=0;
12     }
13
14     public void push(Object element){
15        if(this.size == this.store.length){
16           Object [] store = new Object[this.store.length*2];
17           System.arraycopy(this.store,0,store,0,this.size);
18           this.store = store;
19        }
20        this.store[this.size++]=element;
21     }
22
23     public Object pop(){
24        Object result = this.store[this.size];
25        this.store[this.size]=null;
26        this.size--;
27        if(this.store.length > INITIAL_CAPACITY
28                && this.size*2.7 < this.store.length){
29           Object [] store = new Object[this.store.length/2];
30           System.arraycopy(this.store,0,store,0,this.size);
31           this.store = store;
32        }
33        return result;
34     }
35
36 }
```

Figure 5: An object stack class implemented in Java.

```
1  specification ObjectStackSpecification                    ] spec. name
2  class ObjectStack is edu.colorado.cs.ObjectStack          ]
3  class Object is java.lang.Object                          ] sorts
4  method NewObjectStack is <void <init>()>                  ]
5  method push is <void push(java.lang.Object)>              ]
6  method pop is <javal.lang.Object pop()>                   ] operations
7  define ObjectStack                                        ] simulation set
8  forall s:ObjectStack forall o:Object       (Axiom 1)      ]
9        pop(push(s, o).state).retval == o                   ]
10 forall s:ObjectStack forall o:Object       (Axiom 2)      ]
11       pop(push(s, o).state).state == s                    ] algebraic axioms
```


Figure 6: Example Specification for an `ObjectStack` class (see Fig. 5).

3.2.2 Sorts

Sorts are the algebraic equivalents to Java types. A 1-1 mapping exists between sorts and the Java types of the same name: For each type in Java, there is exactly one sort and vice versa. Thus, sorts belong to packages, just like Java types do. The first major section of the specification file defines shortcuts for them. For example, lines 2-3 in Figure 6 define shortcuts for the `Object` and `ObjectStack` sorts; these shortcuts are used in the remainder of the specification.

3.2.3 Operations

This part of the specification file enumerates the operations and their signatures (e.g., lines 4-6 in Fig. 6). Line 4 in Fig. 6 declares the `NewObjectStack` operation, which corresponds to the constructor for the `ObjectStack` class (Fig. 5, lines 9-12)[3]. Line 5 in Fig. 6 declares the `push` operation, which corresponds to the Java method `push` in Fig. 5, lines 14-21. We borrow the Soot syntax for fully qualified names of Java methods [VRGH+00].

Even though we use Java-like syntax to define the operation signatures, the actual resulting signatures differ from the corresponding Java method signatures: since algebraic specifications do not support implicit parameters or side-effects, we need to include the otherwise implicit "`this`" argument and all potential side-effects in the return value. In our language, all operations return a tuple consisting of all directly accessible values that could potentially be modified or generated by a Java method: receiver, return value, and all arguments. Our language does not model state accessible through other means (such as global variables). We also assume that programmers do not use `public` fields and fields with `package` visibility and instead use `public` getters and setters.

More precisely, given a Java method named `m` defined in a class represented by sort *cls* with n arguments $arg_1, \ldots, arg_n$ of sorts $sort(arg_1), \ldots, sort(arg_n)$, with the return type represented by sort *ret*, we construct the signature of an algebraic operation m within an algebra *cls* as follows:

$$\begin{aligned} m : cls \times sort(arg_1) \times ... \times sort(arg_n) \\ \rightarrow cls \times ret \times sort(arg_1) \times ... \times sort(arg_n) \end{aligned} \tag{1}$$

The receiver argument (of sort *cls*) to the left of the arrow characterizes the original state passed into m. The receiver type to the right of the arrow represents the possibly modified state of the receiver as a result of evaluating the operation. The right hand side of the arrow also includes the return value (of sort *ret*, which can be the sort corresponding to `void`), and the potentially modified arguments (of sorts $sort(arg_i)$). For simplicity, this includes arguments passed by value (Java's primitive types), even though it is guaranteed that those remain unchanged. In case the operation corresponds to a Java constructor, the receiver argument to the

[3]In this section, *constructor* always refers to the constructor of a Java class. This is different from the notion of an algebraic constructor, which is the only kind of operation allowed to occur in a normal form.

left of the arrow is `void`. Note that even though potentially modified arguments can be modeled in our language our interpreter does not currently support them.

The algebraic signature for `push` (line 5, Fig. 6) is:

$$\begin{aligned} push : ObjectStack \times Object \rightarrow \\ ObjectStack \times void \times Object \end{aligned} \tag{2}$$

The `push` operation takes an `ObjectStack` (the receiver in Java) and an `Object` (the parameter in Java). It computes a new `ObjectStack`, has no return value, and computes a potentially modified `Object` instance for the argument. Similarly, the algebraic signature for `pop` (line 6, Fig. 6) is:

$$\begin{aligned} pop : ObjectStack \rightarrow \\ ObjectStack \times Object \end{aligned} \tag{3}$$

Notice how the algebraic signature can be derived automatically from the Java signature.

Hidden Operations The language supports hidden operations, which are algebraic operations that do not correspond to Java operations. We sometimes need such operations to specify the semantics of other operations [TWW82] (see Section 5.3.1 for examples). The `hidden` annotation on an operation indicates that the operation is hidden, for example:

```
hidden method size is
                <edu.colorado.cs.simpleadts.ObjectStack: int size()>
```

Having defined the hidden operation `size` this way, we could now use it within the algebraic specification, even though there is no corresponding Java method in Fig. 5.

External Operations *External operations* are ones for which we do not have or cannot express an algebraic specification but for which we would like to provide a nontrivial semantics; Section 4.5.4 gives an example. The *external* annotation on a operation signature indicates that the operation is external.

3.2.4 Simulation Set

The *simulation set* is a set of classes whose behavior the algebraic interpreter (Section 4) simulates by interpreting their specification (Section 4.1 has the details).

3.2.5 Algebraic Axioms

The algebraic axioms (which are equational axioms) give the specification needed to define the behavior of the operations belonging to classes in the simulation set. For example, consider lines 8-11 of Fig. 6. The `.retval` and `.state` qualifications select elements from the result tuple. `.state` retrieves the first element, which is the possibly modified receiver object (`this`). `.retval` retrieves the second element,

which is the method's return value. `.arg1`, `.arg2`, etc. retrieve the potentially modified arguments. Both axioms are universally quantified over all object stacks and all objects. Axiom 1 (Figure 6) states that invoking `pop` after a `push` returns the object that was last pushed. Axiom 2 states that invoking `pop` on an `ObjectStack` right after invoking `push` reverts the stack back into its prior, pre-push state.

Our language also supports conditional axioms, such as:

```
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.

We also support the more complex *join systems* [TeR03]. A join system allows conditional axioms with arbitrary terms in their condition.

4 An Algebraic Specification Interpreter

Given an algebraic specification for a class and a client for the same class, the client can be executed while the algebraic specification interpreter is performing interpretation to simulate the behavior of the specified class. In this way, the system automatically provides an implementation for existing specifications. Our tool interprets algebraic specifications using term rewriting, which is a well studied area [DP01, TeR03]. However, to our knowledge our system is the first to seamlessly integrate fully automatic algebraic rewriting techniques with Java classes.

The interpreter provides three main benefits. First, it gives programmers more for their effort: They not only get the benefit of a formal specification as documentation, but also a prototype of their class, ready for immediate use. We expect this feature to be particularly useful in multi-programmer projects since it allows developers of some components to test against specifications of other components before those are even implemented. Second, by providing a feature for experimentally validating specifications against implementations, our tool helps prevent divergence of implementation and its specification as the software system evolves. Third, our system is invaluable for debugging algebraic specifications (either hand crafted or one discovered by a discovery tool [HD03]) since it allows a specification to be "run" and its behavior to be observed. 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 in general; (ii) the run produces incorrect answers which indicates a bug in the specification; or (iii) the run fails because the interpreter is unable to produce an answer for a method, which indicates that the specification may be incomplete.

We now describe implementation and algorithmic details of the interpreter.

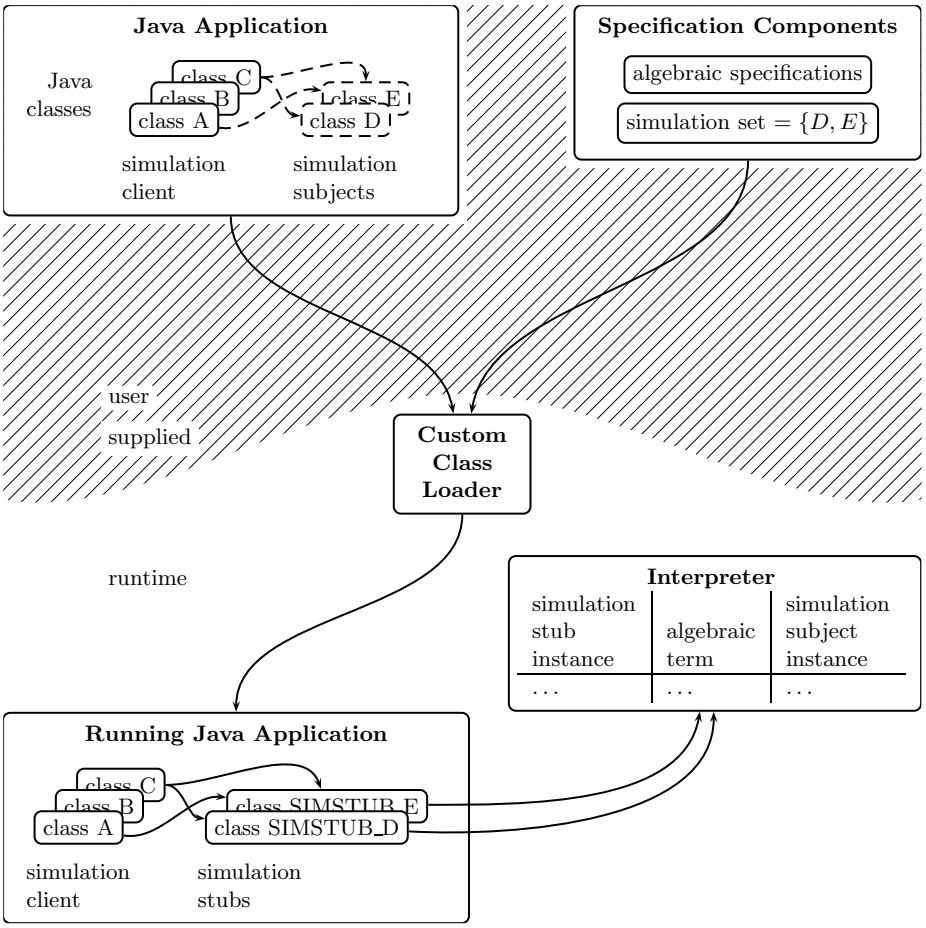

Figure 7: Architecture of our system

4.1 Inputs to the Interpreter

Figure 7 illustrates the architecture of our system. From the *user supplied* parts we see that the user provides two kinds of inputs to our system: *Specification Components*, which are the algebraic specification parts of the input, and *Java Application*, which is the Java part of the input. The *Algebraic Specifications* are specifications in the language described in Section 3.

The algebraic specifications also give the *simulation set* (Section 3.2.4). This simulation set indicates the classes designated for interpretation (e.g., line 7 in Figure 6). `define` clauses define the scope of the specification, i.e. the classes that are being characterized by the specification. Note that this is usually a subset of the sorts (e.g., Figure 6 mentions `Object` and `ObjectStack`, but defines only `ObjectStack`).

The *simulation client* is a Java application that uses the classes in the simulation set. Optionally, users may also provide *simulation subjects* which are real existing implementations of the classes to be simulated. If a user provides these classes, our interpreter continuously checks the result generated by invocations on instances of these classes against interpretation results (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 executing by using results from the simulation subjects.

4.2 Integration with Java

We use a custom Java class loader to load the simulation client. This class loader uses the bytecode engineering library [Apa03] to redirect references to classes belonging to the simulation set to corresponding *simulation stubs*. These simulation stubs contain methods with the same signatures as the classes they simulate, though 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.

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

4.3 Generating Simulation Stubs

We generate the simulation stub, `SIMSTUB_LinkedList`, automatically when we encounter the first reference to `LinkedList`. The 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 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)); }
```

4.4 Modeling Object State with Algebraic Terms

For each simulation stub instance (e.g., an object of type `SIMSTUB_LinkedList`), the interpreter maintains 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 7). 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 accordingly. If simulation subjects are provided, the interpreter also invokes the corresponding method on the simulation subject instance. After executing the code in Section 4.3 , `l1` refers 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 4.5), the interpreter (i) reduces the size of the terms that model the state of an object and (ii) computes the return value of the simulated Java methods. To illustrate this, assume that we have a given set of axioms to interpret with. As an example for (i), assume that the interpreter uses an 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.

4.5 Algebraic Term Rewriting

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 3) is very expressive, which means that it presents a number of challenges to our interpreter. We start by giving a high-level overview of our use of rewriting and then discuss individual challenges we encountered.

4.5.1 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.

4.5.2 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 continues by applying the secondary strategy.

The secondary strategy tries to apply 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 new rewriting step, we revert back to the primary strategy. Note that this strategy uses backtracking and thus is much more expensive than the primary one.

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.

Even our secondary strategy may be unable to reduce a term if it is necessary to increase the size of the term before it can ultimately be reduced. Also, note that our current implementation does not check the set of rewriting rules for *confluence* [DP01] or for consistency. In other words, depending on the internal ordering of rewriting rules, (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 chose affects the capabilities and the efficiency of our system. While we believe that these strategies make sense in practice, there is still considerable room for experimentation.

4.5.3 Conditional Axioms

Conditional axioms lead to additional complexity in the algebraic specification interpreter. For example, consider again the following axiom:

```
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
```

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 constraints of this kind, i.e. sets of simple relations between variables and constants, are called a *semi-equational systems* in the literature [TeR03].

Since our language and system support *join systems* (Section 3.2.5) we may need to use the rewriting system to also determine whether the condition is satisfied. 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 the potential complications: they often allow for more elegant expression of behavior than semi-equational systems do. For example, the following axiom uses the `contains` operation in a constraint to say that if a hash set `h` already contains an object `o`, the size of `h` will not change if `o` is added 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

$$\mathrm{m} = \left\{ \begin{array}{lcl} \mathrm{h} & \mapsto & \text{add(NewHashSet().state, Object@1234).state} \\ \mathrm{o} & \mapsto & \text{Object@1234} \end{array} \right\}$$

However, before we can apply the rewriting, we need to determine if the condition is `true`. We apply `m` to the condition, yielding

```
contains(add(NewHashSet().state, Object@1234).state,
        Object@1234).retval == true
```

Using axioms for the `contains` operation (omitted for brevity), the algebraic interpreter will reduce this relation by rewriting it to the tautology `true == true`. Thus, the check succeeds and the original rewriting rule can now be applied, yielding `size(add(NewHashSet().state,Object@1234).state).retval`.

4.5.4 References to External Methods

Sometimes the specification of one class may need to reference methods from classes 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) Including the specification of `equals` in the specification for hash set, or (ii) extending 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 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 is `external`, which means that whenever the interpreter encounters a term in which all parameters are constants, the Java implementation for this 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 operations.

4.6 Incomplete or Buggy Specifications

If a specification is incomplete, the interpreter may encounter a method application for which it cannot produce a return value. For example, consider the partial `LinkedList` specification used in Section 4.4: If Axiom 6 is missing, the interpreter will not be able to produce the return value (`true`) for the term given above. If a specification is buggy, the interpreter may produce an incorrect answer for a method invocation (which it can detect right away if the user has supplied simulation subjects). In both of these cases, the interpreter reports the problematic term to the user. As we show in Section 5, these diagnostics are invaluable for producing a correct specification or debugging an existing specification. If the user has provided simulation subjects, the interpreter can use the results produced by the simulation subject to continue execution.

5 Evaluation

We now evaluate the performance and effectiveness of the algebraic specification interpreter. Section 5.1 presents performance data for a micro-benchmark that

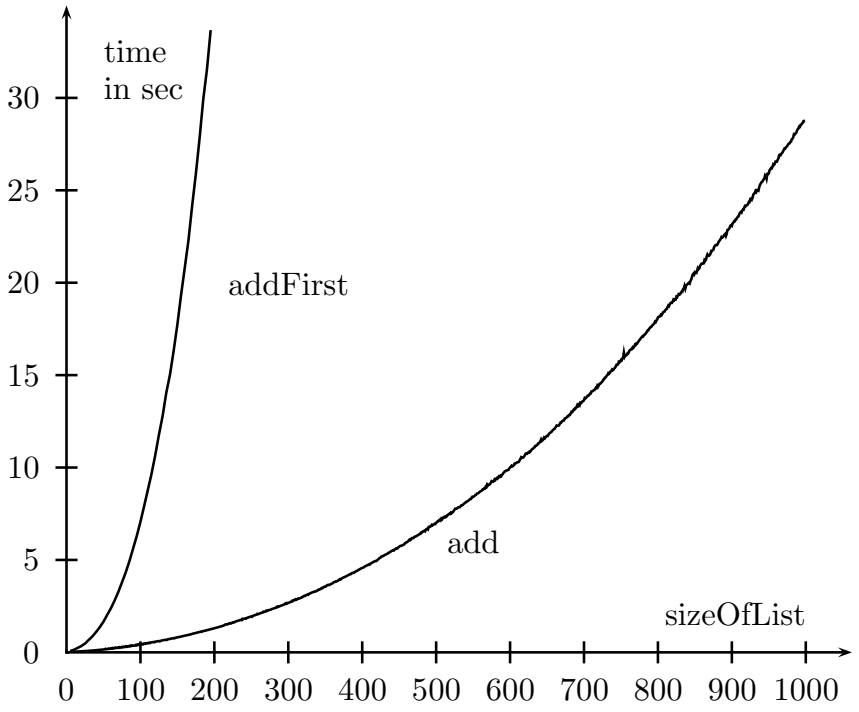

Figure 8: Term Rewriting Benchmark

suggests that our algebraic specification interpreter is usable for prototyping applications; the data also shows how the choice of algebraic axioms for modeling a particular property affects runtime performance. Sections 5.2 and 5.3 give examples for the "extreme specifying" scenario (Figure 1), in which we use our algebraic specification interpreter for developing specifications from scratch. Section 5.4 gives an example for the "discovering and debugging algebraic specifications" scenario (Figure 4).

5.1 Performance Evaluation

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

```
1 Object o = new Object();
2 LinkedList l = new LinkedList();
3 for (int i = 0; i < sizeOfList; i++) l.add(o);
4 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 8, 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 a list (counting from the first element). The `add` method adds an entry to the end of a list, resulting in the following specification of `get`:

```
forall l:LinkedList forall o:Object                                (Axiom 7)
  get(addFirst(l,o).state,0).retval==o

forall l:LinkedList forall o:Object forall i:int                   (Axiom 8)
  if i>=0 then get(addFirst(l,o).state,intAdd(i,1).retval).retval
    == get(l,i).retval
```

The `addFirst` method, on the other hand, adds an entry to the beginning of a list, as expressed in the following alternative specification:

```
forall l:LinkedList forall o:Object forall i:int                   (Axiom 9)
  if size(l).retval == i then get(add(l,o).state,i).retval == o

forall l:LinkedList forall o:Object forall i:int                  (Axiom 10)
  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` (Axioms 7 and 8), our rewriting engine will first have to rewrite the term that corresponds to the entire linked list to be in terms of `add` before it can start to reduce it.

Our results (Figure 8) confirm the intuition above. The horizontal axis of Figure 8 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 one millisecond), it may still be fast enough to be used for prototyping of many applications. Second, some specifications may execute much faster than other (equivalent) specifications, depending on the match

between the specification and the simulation client.

5.2 Extreme Specifying: Hash Set

Programmers can use our system to incrementally develop specifications (and thus prototypes) based on the needs of the code they are developing. For example, when developing a Java class ("client"), the programmer may not know all the requirements on classes it uses ("helper 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 class. Our tool helps in this dilemma by allowing a programmer to develop the specification (and thus a prototype) of the helper class incrementally as needed by the client. This section presents a case study where we developed a partial specification (and, therefore, an automatically derived interpreted prototype) of a hash set hand-in-hand with the client of this hash set. We started 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, we saw that the client needed a hash set, which had to support the methods `add` and `contains`. Thus, we created the following incomplete specification:

```
1 specification HashSetSpecification
2 class HashSet
3 method NewHashSet is <void <init>()>
4 method add is <boolean add(java.lang.Object)>
5 method contains is <boolean contains(java.lang.Object)>
6 define HashSet
```

Note that the specification includes a `NewHashSet` operation for creating a new hash set. Also note that we started with an empty set of axioms (i.e., there was nothing under the `define HashSet` directive). In other words, the interpreter could build up the terms but had no rewriting rules to reduce them. When the "Client" class and the specification were passed to the interpreter, the interpreter responded 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, we added the following axioms:

```
forall o:Object
       add(NewHashSet().state, o).retval == true

forall o:Object forall h:HashSet
       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, however, the client ran successfully.

We now continued implementing the client, inserting the statement `System.out.println("test 0 = "+s.contains(one));` immediately before Line 5. Since this statement invokes a `contains` on an empty hash set, we also added this axiom:

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

On running the modified client and specification set, our system gave 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 was that we forgot to specify how `contains` affects the state of the object. This mistake is easy for programmers to make 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), comes 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 needed to add was the following axiom:

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

After this new axiom, the client executed successfully. Needless to say, the specification of a hash set was still far from complete. When behavior was added to the client class, our interpreter exposed more of the limitations of the specification. Ultimately, such an iterative process may lead to a complete specification of the hash

```
public class PriorityQueue {
       public PriorityQueue(Comparator c){...}
       public Object get(){...}
       public boolean equals(Object other){...}
       public boolean add(Object object){...}
       public boolean addAll(PriorityQueue collection){...}
       public boolean contains(Object object){...}
       public int size(){...}
}
```

Figure 9: A priority queue for Java

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.

5.3 Extreme Specifying: Priority Queue

In this case study, one of the authors, previously untrained on our system, used the specification interpreter to develop the algebraic specifications for a priority queue.

Figure 9 gives the class signature of this queue. In addition to the methods given in Figure 9, the `PriorityQueue` also inherits the `toString()`, `hashCode()`, and `getClass()` methods from `Object`. The constructor of the `PriorityQueue` takes an instance of `java.util.Comparator`, which determines the preorder on the queue's values.

5.3.1 Initial Specification

Unlike the earlier case study of extreme specification (Section 5.2), in this case study we started with a fairly complete manually-constructed specification. Figure 10 gives the sorts and operation types in the `PriorityQueue` algebra. We omitted the methods inherited from `Object` from our specification.

Our initial specification had 36 axioms. Some examples are given below:

```
axiom forall p:PQ
  add(p, null).state == p
axiom forall p:PQ forall o:O
  add(p, o).retval == true

axiom forall c:C
  get(PQ(c).state).state == PQ(c).state
axiom forall c:C
  get(PQ(c).state).retval == null
```

```
specification PriorityQueueSpecification

class C is java.util.Comparator
class PQ is edu.colorado.cs.PriorityQueue
class O is java.lang.Object

method compare is
   <java.util.Comparator: int compare(java.lang.Object, java.lang.Object)>
method PQ is
   <edu.colorado.cs.PriorityQueue: void <init> (java.util.Comparator)>
method add is
   <edu.colorado.cs.PriorityQueue: boolean add(java.lang.Object)>
method get is
   <edu.colorado.cs.PriorityQueue: java.lang.Object get()>
method addAll is
   <edu.colorado.cs.PriorityQueue: boolean addAll(java.util.Collection)>
method contains is
   <edu.colorado.cs.PriorityQueue: boolean contains(java.lang.Object)>
method size is
   <edu.colorado.cs.PriorityQueue: int size()>
method equals is
   <edu.colorado.cs.PriorityQueue: boolean equals(java.lang.Object)>
```

Figure 10: Header part of the Algebraic Specification for `PriorityQueue`

These four axioms specify the base cases for the `add` and `get` methods (implementing *enqueue* and *dequeue*, respectively); they use the short sort name `PQ` defined in the algebraic signature for `PriorityQueue`.

The axioms for inserting and removing data were more complex than the ones for `add` and `get`. We started with the following:

```
axiom forall p:PQ forall o:O                                    (Axiom 11)
  if compare( ? , o, get(p).retval).retval > 0
    then get(add(p, o).state).retval == o


axiom forall p:PQ forall o:O                                    (Axiom 12)
  if 0 >= compare( ? , o, get(p).retval).retval
    then get(add(p, o).state).retval == get(p).retval
```

Without discussing axioms 11 and 12 yet, note the question marks–we need to replace these with an algebraic expression that computes the comparator passed to the constructor of the priority queue. In our algebraic specification so far, there was no way of accessing this comparator. We remedied this situation by adding a new operator,`comp`, that returned the comparator. To hide it from the Java API, we declared it as hidden.

```
axiom forall q:PQ forall o:O
  comp(add(q, o).state).retval == comp(q).retval

axiom forall c:C
  comp(PQ(c).state).retval == c

axiom forall p:PQ forall o:O                                    (Axiom 11')
  if compare(comp(p), o, get(p).retval).retval > 0
    then get(add(p, o).state).retval == o

axiom forall p:PQ forall o:O                                    (Axiom 12')
  if 0 >= compare(comp(p), o, get(p).retval).retval
    then get(add(p, o).state).retval == get(p).retval
```

5.3.2 Debugging the specification

To verify the correctness of our priority queue specification in the absence of a priority queue implementation, we built a test suite for all of the methods we had specified so far, comprising a total of 92 individual tests. We then ran this suite using our interpreter to simulate the specification (which as mentioned above had 36 axioms).

Our interpreter exposed the following kinds of problems in our specification:

- Specification incompleteness

- Specification incorrectness
- Interpretation incompleteness
- Specification verbosity

Specification incompleteness Our interpreter exposed several cases when the specification was incomplete. For example, the interpreter produced the following message:

```
Algebraic Interpreter failed to compute a value. term =
  size(PQ(edu.colorado.cs.PQTestClient$2@3842).state).retval
```

This message indicated that we had not fully specified `size(PQ(x))`. We corrected this error by adding the following axiom:

```
axiom forall c:C
  size(PQ(c).state).retval == 0
```

The interpreter exposed a number of similar bugs. For example, there were several cases where we specified the return value (`.retval`), but failed to specify the effects on the receiver (`.state`). One of the more interesting cases was the following:

```
Algebraic Interpreter failed to compute a value.
term =
   size(get(get(add(PQ(...).state, ...).state).state).state).retval
```

The problem here was that we specified `size` only in terms of `add`, whereas the irreducible term in the error message also makes use of `get`.

Interpretation incompleteness To make `size` to work with terms to which `get` had been applied, we first attempted to introduce a new hidden method `remove` which removed an element added to the queue. The motivation for this was that the semantics for `get` could then be specified in such a way that any occurrences of `get` could be rewritten to terms containing only `add`. Using such a `remove` method, `get` could be specified as follows:

```
axiom forall p:PQ                                          (Axiom 13)
  if get(p).retval != null
    then get(p).state == remove(p, get(p).retval).state
```

The intuition underlying this axiom was that whatever `get` returns should be precisely the element removed from the state as a side effect of the execution of `get`. This axiom, however, failed to work due to the previously mentioned limitations of our term rewriting system (Section 4.5.2): To use the rule above, the rewriting engine would have to grow the term, which neither of our rewriting strategies support.

Thus, we used the following specification instead:

```
axiom forall p:PQ forall o:O                                    (Axiom 13a)
  if get(add(p, o).state).retval == o
    then get(add(p, o).state).state == p

axiom forall p:PQ forall o:O                                    (Axiom 13b)
  if o != get(add(p, o).state).retval
    then get(add(p, o).state).state == add(get(p).state, o).state
```

Specification incorrectness We encountered several situations where our specification was incorrect: For example, we had unintentionally flipped the semantics of the comparator. The interpreter enabled us to find and fix these problems easily.

Specification verbosity During the specification of `equals`, we needed one axiom to specify the conditions under which two priority queues are equal, and another to express the induction step for situations when equality cannot be determined right away (i.e., where the first two elements of a queue are equal). In addition to that, we needed the following three axioms to specify conditions when queues are not equal:

```
axiom forall q:PQ forall c:C forall o:O
   equals(add(q, o).state, PQ(c).state).retval == false

axiom forall q:PQ forall c:C forall o:O
   equals(PQ(c).state, add(q, o).state).retval == false

axiom forall q1:PQ forall q2:PQ
   if get(q1).retval != get(q2).retval
   then equals(q1, q2).retval == false
```

These axioms cover all cases not covered by the equality and induction step axioms, and their only purpose is to express that a failure to show equality implies that `false` should be returned. Thus, we feel this specification is unnecessarily verbose, which makes it hard to understand.

Summary After debugging our specification with the algebraic specification interpreter, we ended up with 30 axioms. We were unable to check all of the axioms for the `equals` and `addAll` methods, because they would have required the rewriting engine to rewrite to larger terms.

We can address the specification verbosity of `equals` by introducing a notion of default logic into our language and system, meaning, in this particular instance, that a failure to show equality would imply inequality. There are tradeoffs involved with this approach that we will explore in future work.

5.4 Discovering and Debugging a Specification: Array List

In this case study, we used our specification discovery tool [HD03] 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.[4] 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 5.2, 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 needed 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 14)
  get(add(ArrayList().state,x0).state,0).retval == x0

forall l:ArrayList forall o1:Object forall o2:Object                (Axiom 15)
  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
```

Adding four of the axioms which describe the behavior of `Iterator` was straightforward. 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 would not use it (see Section 4.5.2). We introduced a hidden operation `removeFirst`, which eliminated the problem:

```
forall l:ArrayList                                                  (Axiom 16)
  next(iterator(l).retval).state
  ==iterator(removeFirst(l).state).retval
```

[4] Available at `www.cs.colorado.edu/~henkel/stuff/javabib/`.

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; the 10 executed axioms can be considered a dynamic slice of the specification.

More details about this case study, including all discovered axioms, are available in a technical report [HD04a].

6 Related Work

Previously [HD03] 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 [DP01, TeR03]. 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 [WP94]. 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 [Fut03, DV03, TeR03], 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 [AH00] 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 [GMH81, HS96, DF94,

CTC01, CTCC98, San91]. Some of these systems require test drivers to be written (e.g. [GMH81]), others generate test cases by themselves from the algebraic specifications [DF94, CTC01, CTCC98]. Sankar [San91] 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 a system for helping developers in documenting their classes using algebraic specifications.

First, we describe a new algebraic specification language which is more closely tied to the implementation language (Java) than previous languages; in particular, our language establishes a 1-1 correspondence between Java signatures and algebraic signatures. This feature should make writing algebraic specifications less daunting for programmers.

Second, we describe and evaluate an algebraic specification interpreter that is seamlessly integrated with Java. The interpreter 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. The algebraic specification interpreter 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.

Third, we describe four scenarios in which programmers can use our tool.

Finally, we evaluate both the performance and usefulness of our interpreter. We evaluate the performance of the interpreter by timing it on example specifications. We evaluate the usefulness of our interpreter by using it for a number of case studies that span two of the four scenarios described above. We demonstrate that our system is indeed effective in helping programmers document their code using algebraic specifications.

Acknowledgements

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 POPL, ECOOP, and ICSE for listening to our ideas and giving great feedback.

References

[AH00] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[Apa03] Apache Software Foundation. BCEL—byte code engineering library, 2003. http://jakarta.apache.org/bcel/.

[CTC01] 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, January 2001.

[CTCC98] 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.

[DF94] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering*, 3(2), April 1994.

[DP01] N. Dershowitz and D. A. Plaisted. *Handbook of Automated Reasoning*, volume 1, chapter Rewriting. Elsevier, 2001.

[DV03] Nachum Dershowitz and Laurent Vigneron. Database of rewriting systems. http://www.loria.fr/ vigneron/RewritingHP/systems.html, 2003.

[Fut03] Kokichi Futatsugi. CafeObj official homepage. http://www.ldl.jaist.ac.jp/cafeobj/, 2003.

[GH78] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GMH81] 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.

[HD03] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[HD04a] Johannes Henkel and Amer 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.

[HD04b] Johannes Henkel and Amer Diwan. A tool for writing and debugging algebraic specifications. In *International Conference on Software Engineering (ICSE)*, 2004.

[HS96] 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.

[Mit96] John C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[San91] S. Sankar. Run-time consistency checking of algebraic specifications. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, Victoria, British Columbia, Canada, September 1991.

[TeR03] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[TWW82] 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), October 1982.

[VRGH+00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[WP94] Y. Wang and D. L. Parnas. Simulating the behavior of software modules by trace rewriting. *ACM Transactions on Software Engineering*, 20(10), October 1994.