

Discovering Algebraic Specifications from Java Classes ^{*}

Johannes Henkel¹ and Amer Diwan¹

University of Colorado at Boulder, Boulder CO 80309, USA
{henkel,diwan}@cs.colorado.edu,
WWW home page: <http://www.cs.colorado.edu/~{henkel,diwan}/>

Abstract. We present and evaluate an automatic tool for extracting algebraic specifications from Java classes. Our tool maps a Java class to an algebraic signature and then uses the signature to generate a large number of terms. The tool evaluates these terms and based on the results of the evaluation, it proposes equations. Finally, the tool generalizes equations to axioms and eliminates many redundant axioms. Since our tool uses dynamic information, it is not guaranteed to be sound or complete. However, we manually inspected the axioms generated in our experiments and found them all to be correct.

1 Introduction

Program specifications are useful for both program development and understanding. Gries [20] describes a programming methodology that develops axiomatic program specifications along with the code itself. At least anecdotal evidence suggests that programs that are developed using Gries' methodology are relatively bug free and easier to maintain. However, developing formal program specifications is difficult and requires significant mathematical maturity on the part of programmers. For this reason, programmers rarely develop full formal specifications with their code.

Recent work proposes tools that automatically discover likely specifications based on program runs [15, 1, 23, 44]. These tools allow programmers to benefit from formal specifications with much less effort. Our research is inspired by these tools. To our knowledge, however, our methodology is the first to tackle high-level functional specifications of significant software components. The main shortcoming of prior tools such as Daikon [15] is that they refer to the inner state of software components when specifying pre- and postconditions.

^{*} This work is supported by NSF grants CCR-0085792 and CCR-0086255, and an IBM Faculty Partnership Award. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Our tool discovers algebraic specifications [22] from Java classes. Algebraic specifications can describe *what* Java classes implement without revealing implementation details. Our approach is as follows. We start by extracting the *signatures* of classes automatically using the Java reflection API. We use the signatures to automatically generate a large number of terms, using heuristics to guide term generation. Each term corresponds to a legal (i.e., does not throw an exception) sequence of method invocations on an instance of the class. We then evaluate the terms and compare their outcomes. These comparisons yield equations between terms. Finally, we generalize these equations to axioms and use term rewriting to eliminate redundant axioms. Note that we can systematically generate more terms if we have low confidence in our axioms.

We evaluate our approach on a variety of classes. Our experiments reveal that our approach is effective in discovering specifications of Java classes even when it does not have access to any implementation details of the classes. Since we employ mostly dynamic techniques, the specifications that we discover are unsound. However, our experiments reveal that the axioms that we discover are actually correct and useful in understanding the behavior of the classes.

The remainder of the paper is organized as follows. Section 2 motivates the need for algebraic specifications. Section 3 describes our approach and an implementation of our approach. Section 4 presents case studies that demonstrate the usefulness and effectiveness of our current prototype. Section 5 discusses extensions to our prototype. Section 6 reviews related work. Finally Section 7 concludes.

2 Motivation

Fig. 1 contains the Java source for an integer stack implementation. Even though the concept of an integer stack is simple, the details of the implementation are tricky. In particular, the implementations of `push` and `pop` may need to resize the internal array which involves copying the contents of one array to another.

There are two levels in understanding this code: *What concept* does it implement and *in which way* does it implement the concept. To understand *what concept* the `IntStack` class implements, one needs to know:

- the `pop` method returns the `int` value that was used as the argument to the most recent call to `push`.
- after applying `pop`, the state of the stack is equivalent to the state before the most recent `push`.
- `pop` will throw an exception if applied to an empty stack.

To understand *in which way* the `IntStack` class implements the concept, one needs to know:

- The inner state of the integer stack is modeled as an array `int [] state` containing the stack elements and an integer `int size` maintaining a pointer to the first free index in the state array.

```

public class IntStack {
  private int [] store;  private int size;
  private static final int INITIAL_CAPACITY=10;
  public IntStack(){
    this.store = new int[INITIAL_CAPACITY];
    this.size=0;
  }
  public void push(int value){
    if(this.size ==this.store.length){
      int [] store = new int[this.store.length*2];
      System.arraycopy(this.store,0,store,0,this.size);
      this.store = store;
    }
    this.store[this.size]=value;
    this.size++;
  }
  public int pop(){
    int result = this.store[this.size-1];
    this.size--;
    if(this.store.length > INITIAL_CAPACITY && this.size*2
       < this.store.length){
      int [] store = new int[this.store.length/2];
      System.arraycopy(this.store,0,store,0,this.size);
      this.store = store;
    }
    return result;
  }
}

```

Fig. 1. An integer stack implementation in Java.

- `push` checks whether `state` is large enough to accommodate one more value and if not, allocates a new `state` array that is twice as big as the old one and copies the old elements over. After potentially scaling the `state` array, `push` adds the new value at the position of `size` in the `state` array and increments `size` by 1.
- ...

By discovering algebraic specifications [21] which do not reveal the implementation, yet characterize the interfaces, we are addressing the first level of understanding: *What concept?* Other tools, e.g. Daikon [15], are more useful for the second level, since their specifications characterize implementation details such as the resizing of the `state` array. Fig. 2 shows an algebraic specification of the Java integer stack discovered by our tool. The first part of the algebraic specification (called the *signature*) gives the type of the operations on the integer stack. The second part of the specification gives the axioms that describe the behavior of the integer stack. The first axiom states that applying the `pop` operation to a stack returns the value that has been pushed previously. The second axiom states that the `pop` operation yields the abstract state in which the stack was before `push` was applied. The third axiom specifies that applying `pop` to the empty integer stack results in an exception. We will further explain these axioms and the syntax that we use in Section 3.1. Note that these axioms refer only to the *observable* behavior of the stack and not to the internal implementation details.

The algebraic specifications discovered by our tool have many uses:

```

TYPE IntStack
FUNCTIONS
  IntStack :  $\rightarrow$  IntStack  $\times$  void
  push : IntStack  $\times$  int  $\rightarrow$  IntStack  $\times$  void
  pop : IntStack  $\rightarrow$  IntStack  $\times$  int
AXIOMS
 $\forall s : \text{IntStack}, i : \text{int}$ 
  pop(push(s, i).state).retval = i
  pop(push(s, i).state).state = s
  pop(IntStack(state).retval)  $\rightsquigarrow$  ArrayIndexOutOfBoundsException

```

Fig. 2. An algebraic specification for `IntStack` (Fig. 1) as discovered by our tool.

- **Reverse Engineering and Program Understanding.** As the example illustrates, algebraic specifications discovered by our tool can be useful for program understanding.
- **Program Documentation.** A programmer can use our tool to document code that he/she has written. In this case, the programmer may want to start with specifications generated by our tool and edit them if necessary.
- **Prototyping.** Let’s suppose the developer wants to implement an efficient integer stack. In this case, the developer can start with a slow and simple (and more likely to be correct) integer stack, use our tools to generate the specifications, and use the specifications to guide and test the efficient final implementation (e.g. [14, 12]).
- **Automatic Test Generation.** The specifications discovered by our tool can be used to automatically generate test cases (e.g. [14, 12]). For example, using the specification in Fig. 2, one can generate the following test case

$$\text{pop}(\text{pop}(\text{push}(\text{push}(\text{IntStack}().\text{state}, 7).\text{state}, 9).\text{state}).\text{retval}) = 7$$

Since this test case has been derived from the first two axioms in Fig. 2, if this test fails, one will know that at least one of these axioms does not hold. By generating more test cases for these particular axioms, the tool will be able to characterize the circumstances under which a particular axiom fails to hold and therefore determine the difference between specification and implementation.

- **Debugging.** If our tool generates axioms that do not make sense, it may indicate a bug in the implementation. For example, the following axiom most likely indicates a bug in the implementation since it says that applying `pop` four times to any stack causes an exception:

$$\forall s : \text{Stack} (\text{pop}(\text{pop}(\text{pop}(\text{pop}(s).\text{state}).\text{state}).\text{state}).\text{state}) \rightsquigarrow \text{Exception})$$

3 Approach

Fig. 3 gives an overview of our approach. We start by generating terms in an algebra derived from a Java class (Sections 3.1 and 3.2). Terms represent sequences

of invocations of methods on an instance of a class. We use these terms to generate equations. For example, we may find that two terms evaluate to the same value. We use *observational equivalence* [14] to determine if values produced by two terms are equal (Section 3.3): We consider them the same if they behave the same with respect to their public methods. Thus, observational equivalence abstracts away from low-level implementation details and only considers the interface. We discover equations (Section 3.4) and then generalize the equations by replacing subterms with universally-quantified typed variables (Section 3.5). This generalization results in axioms. However, these axioms may or may not be true. To determine (with some probability) that the axioms are true, we check them by generating test cases. Finally, we rewrite our axioms to eliminate many redundant axioms (Section 3.6).

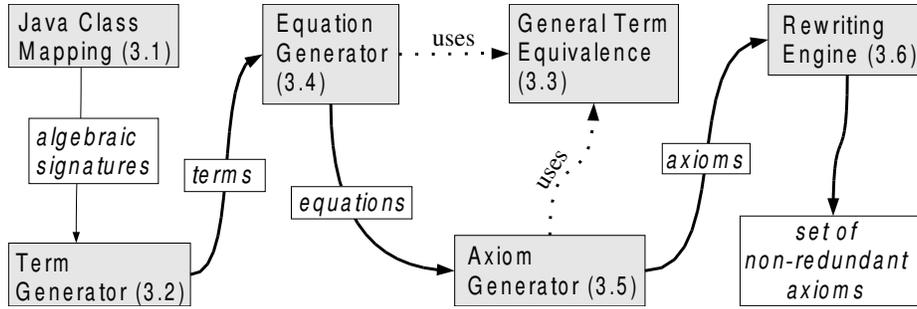


Fig. 3. Architectural overview of our tool for discovering algebraic specifications

3.1 Mapping Java Classes to Algebras

While algebraic specifications have been used to describe object-oriented systems, previous work assumed that a method invocation either has no side effects or that it does not return a value [29]. Our mapping from Java classes to algebras can represent methods with both side effects and return values.

Consider the Java *IntStack* class in Fig. 1. Note that the *pop* method not only returns a value (i.e., it is an *observer*) but also modifies the internal state of the receiver of the *pop* message. To represent such side effects and return value at the same time, each operation in the algebra returns two entities: a reference to the receiver and the normal return value.

More precisely, given a Java method named *m* defined in class *cls* with *n* arguments *arg*₁, *arg*₂, ..., *arg*_{*n*} of types *type*(*arg*₁), *type*(*arg*₂), ..., *type*(*arg*_{*n*}) and return type *returntype*, we construct the signature of an algebraic operation *m* within an algebra *cls* as follows:

$$m : cls \times type(arg_1) \times \dots \times type(arg_n) \rightarrow cls \times returntype \quad (1)$$

The receiver argument (typed *cls*) to the left of the arrow characterizes its original state. The receiver type to the right of the arrow represents the possibly modified state of the receiver as a result of evaluating the operation. For an operation invocation, we use *.state* qualification to refer to the receiver return

value and *.retval* qualification to refer to the normal return value. As an example, consider the following Java fragment.

```
IntStack s = new IntStack(); s.push(4); int result=s.pop();
```

$pop(push(IntStack().state, 4).state).retval$ denotes the integer value stored in `result`, while $pop(push(IntStack().state, 4).state).state$ denotes the state of `s` at the end of the computation. Similarly, the first axiom in Fig. 2 refers to the return value of a *pop*, while the second axiom refers to the state of the stack after a *pop*.

Since constructors do not take an implicit first argument, we leave out the object state from the algebraic signature. Also, for constructors, *returnntype* is always *void*. The signature for constructor `c` of class `cls` looks as follows:

$$c : void \times type(arg_1) \times \dots \times type(arg_n) \rightarrow cls \times void \quad (2)$$

The algebra mapping described here (and implemented in our tool) does not have a means of capturing side effects to arguments other than the receiver of methods. We could generalize our mapping given in Equation 1 to take side effects to arguments into consideration by returning not just the possibly modified receiver but also all the (potentially modified) arguments:

$$\begin{aligned} m : cls \times type(arg_1) \times \dots \times type(arg_n) \\ \rightarrow cls \times returnntype \times type(arg_1) \times \dots \times type(arg_n) \end{aligned} \quad (3)$$

Our implementation cannot handle side effects on static variables and some interactions between objects, i.e. where an object o_1 modifies the state of an owned object o_2 , but the state of o_2 cannot be observed through o_1 . We handle terms that throw exceptions, but we do not capture the (possibly) modified state of an object if an exception has been thrown. The \rightsquigarrow operator distinguishes exception throwing terms from regular terms (e.g., the third axiom in Fig. 2).

In contrast to previous algebraic tools, where users of the tools needed to define elaborate mappings from Java/C++ classes to algebras [29], our implementation for extracting algebraic signatures from Java classes is fully automated and retrieves all the information that is necessary from the Java reflection API.

3.2 Generating Ground Terms

This section considers how to generate terms in an algebra obtained from a Java class. The output of the term generator is an endless stream of terms. The quality of the generated terms is important since the terms are our test cases and thus a critical ingredient for successful dynamic invariant detection [15]. The term generator may either systematically generate all terms up to a given size [10] or randomly generate terms based on some distribution function (e.g., having a bias towards shorter terms).

Our terms satisfy two properties. First, terms do not throw exceptions when evaluated. Second, all terms return the state of an object, but never a return value. For example, we will generate $pop(push(IntStack().state, 5).state).state$ but not $pop(push(IntStack().state, 5).state).retval$ because it computes a return

value, not a state. During equation generation we also generate terms that return values and throw exceptions (Section 3.4).

Our prototype implementation represents terms as trees with nodes representing operation application, constructor application, array construction, constants, and free variables. The children of each term are its arguments. Each node is represented by a Java object containing a reference to the appropriate Java reflection API object that is needed for evaluation. For example, an operation-application node may contain a pointer to the `java.lang.reflect.Method` instance representing the `pop` method of class `IntStack`.

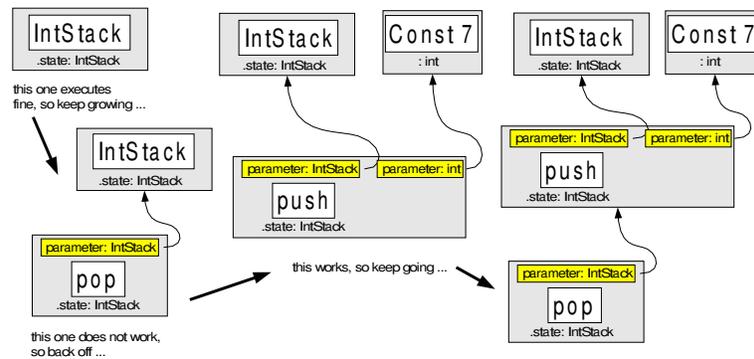


Fig. 4. Growing terms incrementally

Growing Terms We grow terms incrementally, beginning with a constructor. Fig. 4 shows how we generate a term for the `IntStack` algebra. We begin by selecting the constructor `IntStack`. We execute the term using Java reflection. Since it executes without throwing any exception, we continue to extend it by selecting an operation that can be applied to the existing term. When we grow the term with a `pop` and evaluate the result, it throws an exception. We backtrack and now choose `push`. `push` requires an argument, so we generate a term that returns a value of that particular type, in this case the integer constant 7. This time, the term executes without throwing an exception. Thus we can keep growing the term. We then extend the term with `pop`, which also executes without throwing an exception.

Since the goal of the term generator is to explore the state space of instances of a class, operations that do not modify the state of the receiver need not be part of any terms. We can determine that an operation does not modify the state of the receiver by either using a compile-time mod-ref analysis or a run-time heuristic. The run-time heuristic works as follows. Whenever we grow a term, we keep the new term only if the last operation of the term modifies the internal state of the receiver. We use a hash sum of the serialized inner state to determine if an operation changes the inner state. Although this process does not

remove all the operation applications that do not modify the observable state of the object,¹ it significantly improves the quality of terms.

Generating Arguments for Terms We can use terms themselves as object arguments for any operations in a term. However, in the interest of efficiency and quality of ultimate results, we identify two special cases that we handle differently: arguments of a primitive type (such as integer) and arguments that are instances of immutable classes (i.e., classes, such as Object, that do not have any modifying operations).

For arguments of a primitive type, we maintain a precomputed set of constant nodes that have values of the primitive type (e.g., values between 0 and 10). Whenever we need an argument of a primitive type, we select one value from the set. If our set is too small, we will not explore enough of the state space and may end up missing some axioms. If the set is too large, the tool will become too inefficient for practical use or, if a random selection of test cases is used, our terms will populate the state space too sparsely. Our approach for primitive types works well for types such as integers and booleans.²

Our approach for generating arguments of immutable classes is similar to that for primitive types: we precompute a set of values of that type. An *immutable class* is a class that has no methods for modifying the state of an instance after creation (e.g., Object and String classes in Java). For example, instead of using $push(ObjStack().state, Object().state).state$ we prefer to generate $push(ObjStack().state, obj@4).state$ where $obj@4$ is an object in the precomputed set. While $Object().state$ in the first term will generate a new instance of Object each time it is evaluated, the second term contains the particular instance $obj@4$ of Object as a constant. As we elaborate in Section 3.4, our technique of using constants for immutable classes allows us to exploit reference equality for instances of immutable classes which avoids the high cost incurred by the traditional EQN method [14]. However, if we use reference equality, we need to make sure that we still have some chance of computing the same instance when comparing two terms. For example,

$$\begin{array}{l} pop(push(ObjStack().state, obj@9).state).retval \quad \text{and} \\ pop(push(push(ObjStack().state, obj@4), obj@9).state).retval \end{array}$$

would both compute the same instance, namely $obj@9$. Thus, analogously to primitive types, it is important to pick a set that is neither too big nor too small.

For non-immutable classes, we can not safely replace subterms with a particular instance as a precomputed constant, since then the potentially modified state of the instance would be used in the next evaluation of the term. For example, if we execute the term $push(IntStack@4, 9).state$ multiple times, we will compute a different inner state each time.

¹ Just the serialized inner state being different does not guarantee that the objects before and after the applications are different in any externally observable way.

² We have no experience with floating point values.

Algorithm 1 describes how we determine if a class is immutable. Intuitively, we generate a number of instances of the class, apply all operations to each instance, and look at the hash code of the serialized instance before and after operator application.³ If the hashes are the same for a sufficiently large number of instances, then we can be confident that the class is immutable. An alternative to this approach is to use a compile-time mod-ref analysis. If the mod sets of all methods of a class (except constructors) are empty then the class is immutable.

Algorithm 1 Class immutability

 CLASS-IMMUTABLE(C) **begin**
Require: C is a class

repeat
 $i \leftarrow$ instance of C
for all operations op in class C **do**
 $hashBefore \leftarrow$ SERIALIZE-AND-HASH(i)

 $hashAfter \leftarrow$ SERIALIZE-AND-HASH($op(i)$)

if $hashAfter \neq hashBefore$ **then**
return false

end if
end for
until confident of outcome

return true

end

3.3 Term Equivalence

To compare whether or not two ground terms are the same, we use a variation of Doong and Frankl's EQN method [14]. EQN tries to determine whether two terms evaluate to observationally equivalent values. Doong and Frankl define that two values are *observationally equivalent* if they behave the same when we apply arbitrary operations to both of them. Two values can be observationally equivalent even though they have different internal representations. For example, consider the `IntStack` implementation in Java from Fig. 1, where `pop` is implemented as

```
public int pop(){
    return this.store[--this.size];
}
```

In this case, applying 11 `push` operations and 2 `pop` operations results in a different inner state compared to 9 `push` operations,⁴ even though both values are observationally equivalent if the first 9 pushed values were the same.

The EQN method for observational equivalence is effective but inefficient since it needs to apply many operations to both terms in order to gain confidence

³ If the operator requires arguments, one may need to generate many test cases for these arguments as well to become confident.

⁴ Recall that the `IntStack` uses an array representation with an initial size of 10 elements.

that they are indeed the same. Thus, we identify two special cases that we can handle quickly: primitive types and immutable classes. Algorithm 2 describes how we dispatch between the three possibilities. In Algorithm 2, the parameters

Algorithm 2 General term equivalence dispatch

```

GTE-DISPATCH( $t_1, t_2$ ) begin
Require:  $t_1, t_2$  are terms of the same type  $T$ .
  if  $T$  is a primitive type then
    return PRIMITIVE-EQUALS( $t_1, t_2$ )
  else
    if REFSEQ-APPLIES( $t_1, t_2$ ) then
      return EVAL( $t_1$ )=refEVAL( $t_2$ )
    else
      return EQN( $t_1, t_2$ )
    end if
  end if
end

```

for GTE-DISPATCH are terms, not the values computed by the terms. This is necessary since each of PRIMITIVE-EQUALS, REFSEQ-APPLIES and EQN needs more than a single sample of what the terms evaluate to. EVAL(t) denotes the result of the evaluation of t and =_{ref} checks reference equality.

GTE-DISPATCH is conservative in that it is accurate whenever it exposes non-equivalence. It may be inaccurate when it finds equivalence.

We now describe the algorithms for the three cases in more detail.

Algorithm 3 Equivalence for primitive types

```

PRIMITIVE-EQUALS( $t_1, t_2$ ) begin
Require:  $t_1, t_2$  are terms computing values of the same primitive type.
   $result_{1a} \leftarrow$  EVAL( $t_1$ ),  $result_{1b} \leftarrow$  EVAL( $t_1$ )
   $result_{2a} \leftarrow$  EVAL( $t_2$ ),  $result_{2b} \leftarrow$  EVAL( $t_2$ )
   $consistent_1 \leftarrow result_{1a} =_{val} result_{1b}$ 
   $consistent_2 \leftarrow result_{2a} =_{val} result_{2b}$ 
  if not  $consistent_1$  and not  $consistent_2$  then
    return true
  end if
  if not  $consistent_1$  or not  $consistent_2$  then
    return false
  end if
  return  $result_{1a} =_{val} result_{2a}$ 
end

```

Equivalence for Primitive Types Consider the question whether $pop(push(IntStack().state, 4)).retval$ and 4 are equivalent terms. We can confirm equivalence by checking that both terms evaluate to 4.

Now consider the term $hashCode(IntStack().state).retval$. Since the `hashCode` function will compute a different value for each `IntStack` instance, the term will evaluate to a different value each time. To identify such cases,

Algorithm 3 evaluates each term twice. If both terms evaluate to the same value both times then they are equal. If only one of the terms evaluates to the same value both times, then the two terms cannot be equal. Finally, if both terms evaluate to different values each time they are evaluated, then we assume that they are equal.

Algorithm 4 Checking for reference equality

 REFEQ-APPLIES(t_1, t_2) **begin**
Require: t_1, t_2 are terms of the same reference type.

return EVAL(t_1) =_{ref} EVAL(t_1)
 and EVAL(t_2) =_{ref} EVAL(t_2)

end

Comparing the References Computed by Terms Since our algorithm currently handles only side effects to instance variables of receivers and we use instances of immutable classes from a fixed set (Section 3.2), there may be many situations where we can use reference equality rather than resorting to the more expensive observational equivalence algorithm (Section 3.3). Thus, we use a heuristic similar to that for primitive types (Algorithm 4): we evaluate each term twice and see if each term evaluates to the same value in both evaluations. If they do, then we can simply compare the references that they return. If they do not, then we use the observational equivalence procedure. For example, for the terms $pop(push(ObjStack().state, obj@123).retval)$ and $obj@123$, REFEQ-APPLIES returns true, while it returns false for

$$pop(push(ObjStack().state, obj@123).state)$$

 and $ObjStack().state$.

Observational Equivalence Algorithm 5 shows pseudocode for our version of EQN.⁵ EQN approximates observational equivalence of two terms of class C as follows. First, it checks if both terms evaluate to objects with identical inner state by using SERIALIZE-AND-HASH which serializes an object and computes a hash value of that serialization. If the objects have the same inner state, we can safely assume that they are observationally equivalent.

If SERIALIZE-AND-HASH fails to prove equivalence we test for observational equivalence. We start by generating term stubs that take an argument of type C and apply the stubs to t_1 and t_2 . We then pick an observer⁶ and apply it to both terms. We compare the outputs of the observers using GTE-DISPATCH (Algorithm 2). If GTE-DISPATCH returns false, then we know that t_1 and t_2 are not observationally equivalent, otherwise we become more confident in their equivalence.

⁵ This algorithm could run into an infinite recursion. This can be cured by adding a recursion limit which we left out for clarity.

⁶ An observer is an operation op such that the type of $op(\dots).retval$ is not void.

Algorithm 5 Observational equivalence

EQN(t_1, t_2)

Require: t_1, t_2 are terms, evaluating to instances of some class C

```

if SERIALIZE-AND-HASH(EVAL( $t_1$ ))=vatSERIALIZE-AND-HASH(EVAL( $t_2$ ))
then
  return true
end if
repeat
  Generate a term  $stb$ , with an argument of type  $C$ 
  for all observers  $ob$  applicable to evaluation results of  $stb$  do
     $stubapp_1 \leftarrow stb(t_1), stubapp_2 \leftarrow stb(t_2)$ 
     $obsapp_1 \leftarrow ob(stubapp_1, \dots).retval, obsapp_2 \leftarrow ob(stubapp_2, \dots).retval$ 
    if not GTE-DISPATCH( $obsapp_1, obsapp_2$ ) then
      return false
    end if
  end for
until confident of outcome
return true

```

For example, consider applying this procedure to terms from the *IntStack* algebra. An example of a stub is $\lambda x.push(push(x, 2).state, 3).state$ and an example of an observer is *pop*. If t_1 is $push(IntStack(), 3).state$, the application of the stub and the observer yields

$$pop(push(push(push(IntStack().state, 3).state, 2).state, 3).state).retval$$

3.4 Finding Equations

The form of the equations determines the form of the algebraic specifications that our system will discover. Our current implementation handles only equalities. This is a limitation of our current implementation and not of our approach.

We can easily add new types of equations and can enable or disable equation types. So far we have added three kinds of equations to our implementation:

- **State Equations: Equality of Distinct Terms.** For example,

$$pop(push(IntStack().state, 4).state).state = IntStack().state$$

is a state equation. These equations are useful in characterizing how operations affect the observable state of an object. We generate these equations whenever we find that two distinct terms are equivalent. An approach that is able to find some but not all state equations is as follows. We take all terms produced by the term generator, evaluate them, and hash the evaluation results in a table. Whenever we detect a conflict in the hash table we have a possible equation. Note that since hashing does not use full observational equivalence, this method will only find some of the state equations. We call the optimized equation generator *state/hash*, while the more general generator (using observational equivalence) is called *state/eqn*.

- **Observer Equations: Equality of a Term to a Constant.** These equations take the following form (*obs* is an observer and *c* is a constant of primitive type):

$$obs(term_1, arg_2, \dots, arg_n).retval = c$$

Observer equations characterize the interactions between operations that modify and operations that observe. For example,

$$pop(push(IntStack().state, 4).state).retval = 4$$

is an observer equation.

To generate observer equations we start with a term and apply an operation that returns a constant of a primitive type. We form an equation by equating the term to the constant.

- **Difference Equations: Constant Difference Between Terms.** These equations take the following form (*obs* is an observer, *op* is an operation computing a value of a primitive type, and *diff* is a constant of a primitive type):

$$op(obs(term_1.state).retval, diff).retval = obs(term_2).retval$$

For example, given the operation *size* with the signature

$$size : IntStack \rightarrow IntStack \times int$$

which returns the size of the integer stack, we could formulate the following equation.

$$\begin{aligned} &IntAdd(size(IntStack().state), 1).retval \\ &= size(push(IntStack().state, 3).state).retval \end{aligned}$$

In this example, *op* is *IntAdd* (i.e., integer addition) and *diff* is 1.

To generate such axioms, we generate two terms, apply an observer to both terms, and take their difference. In practice, we found that difference equations with a small value for *diff* are the most interesting ones. Therefore, we only generate such an equation if *diff* is lower than a fixed threshold. This technique filters out most spurious difference equations.

3.5 Generating Axioms

Our axioms are 3-tuples (t_1, t_2, V) , where t_1 and t_2 are terms and V is a set of universally-quantified, typed variables that appear as free variables in t_1 and t_2 . An equation is simply an axiom with $V = \{\}$.

The generation of axioms is an abstraction process that introduces free variables into equations. For example, given the equation

$$\begin{aligned} &IntAdd(size(IntStack().state).retval, 1).retval \\ &= size(push(IntStack().state, 3).state).retval \end{aligned} \quad (4)$$

our axiom generator can abstract $IntStack().state$ into the quantified variable s of type $IntStack$ and 3 into i of type int to discover the axiom

$$\begin{aligned} &\forall s : IntStack \forall i : int \\ &IntAdd(size(s).retval, 1).retval = size(push(s, i).state).retval \end{aligned} \quad (5)$$

Algorithm 6 Axiom generation

```

generateAxiom(Algebra) begin
  (term1, term2) ← generate an equation in Algebra
  Subterms ← the set of unique subterms occurring in term1 and term2
  V ← a set of typed, universally-quantified variables x1, ..., xn
    with one xi for each subtermi ∈ Subterms
  for xi ∈ V do
    Replace each occurrence of subtermi with the variable xi in term1 and term2
    Generate a large set of test cases testset where each test case is a set of
      generated terms {test1, ..., testn}, such that testj can replace xj ∈ V
    for testcase ∈ testset do
      Set all xj ∈ V to the corresponding testj ∈ testcase
      if EQN-DISPATCH(term1, term2) = false then
        Undo the replacement of subtermi in term1 and term2
        Stop executing test cases
      end if
    end for
  end for
  Eliminate all xi from V which occur neither in term1 nor in term2
  return the axiom (term1, term2, V)
end

```

Algorithm 6 describes the axiom generator, with minor optimizations left out for clarity. To generate an axiom for a particular algebra, we first use any of the equation generators as described in Section 3.4 to come up with an equation. We then compute the set of all subterms of *term*₁ and *term*₂. For example, given the terms *push(IntStack().state, 4).state* and *IntStack().state*, the set of subterms would be {*push(IntStack().state, 4).state*, *IntStack().state*, 4}. We then initialize *V* as the set of universally-quantified variables, so that for each subterm there is exactly one corresponding universally-quantified variable in *V*. The loop then checks for each subterm, whether we can abstract all occurrences to a free variable. First, we replace all occurrences of the subterm with a free variable. Then, we generate test cases, where each test case replaces all the free variables in the terms with generated terms. We compare whether *term*₁ and *term*₂ are equivalent under all test cases. If not, we undo the replacement of the particular subterm. At the end, we eliminate all free variables that do not occur in the terms and return the axiom.

3.6 Axiom Redundancy Elimination by Axiom Rewriting

The axiom generator (Section 3.5) generates many redundant axioms. For example, for the *IntStack* algebra, our generator may generate both

$$\forall x_6 : \text{IntStack}, \forall i_6, j_6 : \text{int} \quad \text{pop}(\text{pop}(\text{push}(\text{push}(x_6, i_6).\text{state}, j_6).\text{state}).\text{state}).\text{state}) = x_6 \quad (6)$$

and

$$\forall x_7 : \text{IntStack} \forall i_7 : \text{int} . \text{pop}(\text{push}(x_7, i_7).\text{state}).\text{state} = x_7 \quad (7)$$

We eliminate redundant axioms using term rewriting [35]. We use axioms that satisfy these two requirements as *rewriting rules*: (i) the left and right-hand sides must be of different length; and (ii) the free variables occurring in the shorter side must be a subset of the free variables occurring in the longer side. When using a rewrite rule on an axiom, we try to unify the longer side of the rewrite rule with terms in the axiom. If there is a match, we replace the term with the shorter side of the rewrite rule.

Whenever we are about to add a new axiom we note if any of the existing rewrite rules can simplify the axiom. If the simplified axiom is a rewrite rule we try to rewrite all existing axioms with this rule. If the rewriting makes an axiom redundant or trivial we throw it away. If a rewritten axiom yields a rewrite rule then we use that rule to simplify all existing axioms. This process terminates since each rewriting application reduces the length of the terms of the axioms that it rewrites, which means that in general, the addition of an axiom can only lead to a finite number of rewriting and elimination steps.

We now sketch how to rewrite the example Axioms (6) and (7) as shown above. Suppose that Axiom (6) already exists and we are about to add Axiom (7). First, we try to rewrite Axiom (7) using Axiom (6) as a rewriting rule. Unfortunately since the left (longer) term of Axiom (7) does not unify with any subterm in Axiom (6) rewriting fails. We find that Axiom (7) does not already exist and it is not a trivial axiom so we add it to the set of known axioms. Since Axiom (7) is a rewriting rule, we try to rewrite all existing axioms, namely Axiom (6). We find that the left side of Axiom (7) unifies with the following subterm of Axiom (6)

$$\text{pop}(\text{push}(\text{push}(x_6, i_6).\text{state}, j_6).\text{state}).\text{state})$$

with the unifier

$$\{x_7 \rightarrow \text{push}(x_6, i_6).\text{state}, i_7 \rightarrow j_6\} .$$

Therefore, we instantiate the right side of Axiom (7) with the unifier we found and obtain

$$\text{push}(x_6, i_6).\text{state}$$

which we use as a replacement for the subterm that we found in Axiom (6). Therefore, Axiom (6) rewrites to

$$\forall x_6 : \text{IntStack}, \forall i_6 : \text{int} (\text{pop}(\text{push}(x_6, i_6).\text{state}).\text{state} = x_6) \quad (8)$$

which is equivalent to Axiom (7). Since the rewritten Axiom (6) is identical to Axiom (7), we eliminate Axiom (6). In summary, we end up with Axiom (7) as the only axiom in the final set of axioms.

4 Evaluation of our Prototype

We conducted our evaluations on a Pentium 4 1.4 GHz workstation with 512 MB of RDRAM running SuSE Linux 8.1 and Sun JDK 1.4.1. We configured our system as follows: As a default, we used a term size of 5 for the equation

generators⁷ and a test case size of four.⁸ For `HashMap`, we chose a term size of 4 for all equation generators except for the observer equation generator, where a size of 6 was beneficial. We configured the system for `HashSet` in the same way as `HashMap`, except that we found that a test case size of 3 was sufficient. We also configured three distinct instances of `Object` and similar small pools for primitive types.

Table 1 describes the our benchmark programs. Column “# op” gives the number of operations in the class and “# observ” gives the number of operations that are observers (i.e., operations with a non-void return type).

Table 1. Java classes used in our evaluation

Java Class	Description	Source	# op	# observ.
<code>IntegerStack</code>	minimal integer stack	Henkel	11	5
<code>IntegerStack2</code>	efficient integer stack (Fig. 1)	Henkel	11	5
<code>ObjectStack</code>	same as above but for objects	Henkel	11	5
<code>FullObjectStack</code>	another Object stack	Henkel	15	8
<code>IntegerQueue</code>	a FIFO queue for integers	Henkel	15	8
<code>ObjectMapping</code>	a mapping between objects	Henkel	15	9
<code>ObjectQueue</code>	a FIFO queue for objects	Henkel	15	8
<code>LinkedList</code>	linked list	Sun JDK 1.4.1	39	30
<code>HashSet</code>	hash set	Sun JDK 1.4.1	23	17
<code>HashMap</code>	hash map	Sun JDK 1.4.1	22	15

Section 4.1 gives performance characteristics for our system. Section 4.2 presents data that suggests that our tool is successful in exercising most of the class under consideration and is thus likely to be mostly sound. Finally Section 4.3 discusses some axioms that our tool discovers.

4.1 Performance of the Tool

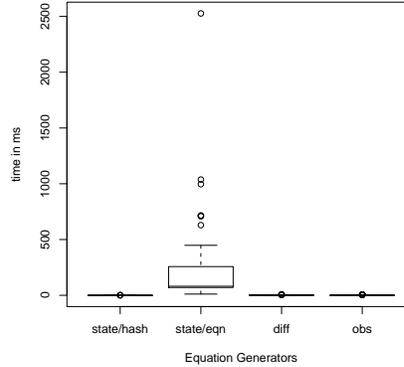
Table 2. Timings for our benchmark programs

benchmark	# before rewriting	# final axioms	time
<code>IntegerStack</code>	116	18	8.48 sec
<code>IntegerStack2</code>	142	18	6.69 sec
<code>ObjectStack</code>	116	9	8.22 sec
<code>FullObjectStack</code>	145	27	12.97 sec
<code>IntegerQueue</code>	404	28	1.59 min
<code>ObjectMapping</code>	194	24	8.97 sec
<code>ObjectQueue</code>	404	24	1.59 min
<code>LinkedList</code>	1045	116	2.37 min
<code>HashSet</code>	6830	74	1.75 h
<code>HashMap</code>	8989	71	15.28 h

⁷ Term size is defined as the number of nodes in the tree representing the term. For example, the size of `push(IntStack().state, 4).state` is 3.

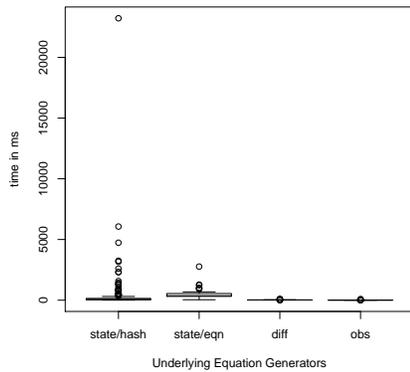
⁸ This means that test cases can have up to $5+4 = 9$ nodes in their term representation.

algebra	time in sec (# generated terms)			
	size 10		size 11	
IntegerStack	0.95	(166)	1.96	(418)
IntegerStack2	0.73	(251)	1.44	(566)
ObjectStack	0.84	(166)	1.47	(418)
FullObjectStack	1.34	(166)	2.21	(418)
IntegerQueue	2.54	(358)	4.49	(722)
ObjectMapping	0.72	(82)	0.94	(82)
ObjectQueue	2.15	(358)	3.89	(722)
LinkedList	27.84	(960)	64.96	(1684)
HashMap	133.57	(4492)	294.4	(6922)
HashSet	129.99	(1094)	273.97	(1399)

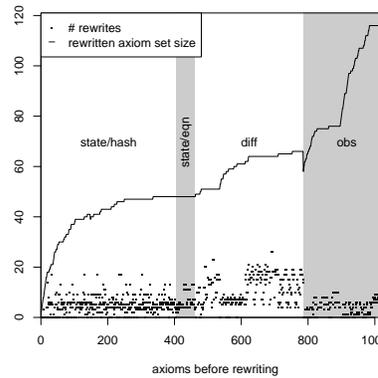


a. Term generation efficiency

b. Efficiency of equation generation (LinkedList)



c. Efficiency of axiom generation (LinkedList)



d. Learning curve (LinkedList)

Fig. 5. Efficiency and effectiveness of our tool

Table 2 gives the overall performance of our system. For each benchmark, we display the number of axioms before rewriting, after rewriting and the time it took to generate the axioms. The table shows that our redundancy reduction by rewriting is very effective. It also shows that our system is fast for all of the classes except for `HashSet` and `HashMap`. `HashMap` is more demanding since many of its operations take two arguments, which enlarges the state space. Running our system with smaller test term sizes for these classes takes less time, but introduces erroneous axioms which can 'infect' many other correct axioms during the rewriting.

We see from Table 2 that `LinkedList` has the most axioms (116). While 116 axioms is quite a lot, it is worth noting that the Java standard implementation of linked list has a large number of operations (39) and thus, 116 axioms is not excessive.

Fig. 5 explores the performance of our system in detail. Fig. 5a gives the time to generate all terms of sizes 10 (first column) and 11 (second column) respectively. We see that the number of terms does increase significantly with term length, even though we prune away many useless terms. We also see that classes with a large number of terms (e.g., `HashSet`) are the ones that take the most time in our system.

Fig. 5b and c are box plots that give the distribution of the time to generate the different kinds of equations and axioms for `LinkedList`. As an example, consider the state/eqn plot in Fig. 5b. The box denotes the interquartile range, which contains the 50% of the values that fall within the second and third quartile. The line inside the box is the median, which overlaps with the bottom of the box due to a strong bias towards low values. Values above the whisker are outliers. From Fig. 5b and c we see that the state axioms and equations are the most expensive to generate with state/eqn being the slowest. For the most part, observer and difference equations and axioms are fast to generate and all equations and axioms of those type take approximately the same time. These results suggest that it may be worthwhile to try other orderings or kinds of equations in order to speed up the equation and axiom generation.

Fig. 5d gives insight into the behavior of our tool when discovering axioms for `LinkedList`. The x axis denotes time in terms of axioms generated by the axiom generator. The “learning curve” is the number of axioms that have been discovered and rewritten. Note that this curve does not ascend monotonically, as the discovery of one axiom can lead to the elimination of numerous other axioms. The dots denote the number of rewriting events per discovered axiom. We run our system as follows: first, we discover state axioms using the hash table optimization (state/hash), then we generate the general state axioms (state/eqn), then we generate the difference axioms and finally the observer axioms. The shaded areas in Fig. 5d denote these different zones.

4.2 Coverage Measurements

Since our system is based on a dynamic approach it is neither sound nor complete. One way in which our system can fail to be sound is if the generated terms do not adequately exercise the full range of behavior of the class under consideration. In this section we use basic block coverage to determine the effectiveness of our term generator in exploring the class code.

Overall, we find that our terms yield a high coverage: 100% for `ObjectStack`, `FullObjectStack`, `IntegerQueue`, `IntegerStack2`, `ObjectMapping`, and `ObjectQueue`; 62.5% for `IntegerStack`; 72.8% for `LinkedList`; 84.6% for `HashSet`; and 73.3% for `HashMap`. When our coverage is less than 100% it is often because our terms do not adequately explore the argument space for operations (e.g., for `LinkedList` we did not generate a term that called `removeAll` with a collection that contains a subset of the receiver’s elements). Other reasons include: dead code (e.g., some code in `LinkedList.subList`); corner cases (e.g., corner cases for `initialCapacity` in `HashTable`).

We are currently using these results and observations to improve our tool.

4.3 Manual Inspection of Axioms

Table 3. Representative selection of example axioms

1	$intPlus(0, 1).retval = 1$
2	$\forall x_0 : HashMap (size(x_0).state = x_0)$
3	$\forall x_0 : HashMap (putAll(x_0, x_0).state = x_0)$
4	$\forall x_0 : HashMap, \forall x_1, \forall x_2 : Object (get(put(x_0, x_1, x_2).state, x_1).retval = x_2)$
5	$\forall x_0 : Object (get(put(HashMap().state, Object@0, x_0).state, Object@1).retval = null)$
6	$\forall x_0 : LinkedList, \forall x_1 : Integer (addAll(x_0, x_1, LinkedList().state).state = x_0)$

To make sure that the axioms discovered by our tool were correct, we manually verified the generated axioms for all classes and found no mistakes in the axioms. The axioms though many for some classes (e.g., *LinkedList*), were relatively easy to read (at most an hour of our time per class).

Table 3 gives a few sample axioms from our tool. Axiom 1 in Table 3 is a trivial one and arises because our tool does not have background axioms for integer arithmetic.

Axiom 2 says that invoking *size* on a *HashMap* does not modify its internal state. Our system generates such axioms for each pure observer (i.e., an operation that has a non-void return value and does not change the state of the object), for example, it finds 16 such axioms for *LinkedList*.

Axiom 3 says that if we add all mappings of a *HashMap* into an equivalent *HashMap*, the receiver's state does not change. Axiom 4 gives a partial characterization of the *get* and *put* operations.

Axiom 5 is one of the axioms that could be more abstract if our tool would support conditional axioms: Instead of using the constants *Object@0* and *Object@1*, the axiom could then be rewritten into

$$\forall x_0, x_1, x_2 : Object \\ x_1 \neq x_2 \Rightarrow get(put(HashMap().state, x_1, x_0).state, x_2).retval = null$$

Axiom 6 points out an incompletely specified behavior in the documentation for *LinkedList*. *addAll* adds all elements in the third argument to x_1 at position x_0 . Axiom 6 says that no matter what position we specify, adding an empty list to x_1 does not modify x_1 . In other words, *addAll* does not verify that x_0 is within the bounds of the list.

Finally, it is worth noting that while we can manually verify that the axioms generated by our tool are correct, it is much harder to verify that our tool generates all the axioms needed to fully specify the interface of a class.

5 Limitations and Future Work

The limitations of our implementation fall into three categories: unsoundness, incompleteness, and inefficiency. Our future work will strive for improvements in each of these areas. Note that given a time limit, better efficiency will allow

us to execute more tests and discover more axioms and thus improve soundness and completeness.

- **Limited Side Effects.** We currently allow only side effects to the receiver of a method. This means that our specifications may be unsound since non-equivalent states are considered equivalent, and they may be incomplete because some side-effects are not described. Section 3.1 sketches how to deal with side effects to non-receiver arguments. We are also planning to model some interactions between objects that we are currently unable to capture.
- **Arguments to Methods are Naively Generated.** Currently, our methodology for choosing arguments for methods is insensitive to the body of a method. We plan to use domain analysis [31] to select arguments more carefully in order to achieve better coverage faster. Also, we plan to extend domain analysis beyond simple types. This will improve the efficiency of our tool.
- **No Support for Conditional Axioms.** Support for conditional axioms would enhance completeness. We can support conditional axioms by changing the abstraction mechanism in the axiom generator (Section 3.5). More specifically, if a particular test case does not satisfy an axiom, we can add it to an exception set and then derive a constraint for the conditional axiom from the exception set. capture the same information.
- **Measuring Unsoundness and Incompleteness.** We plan to use the axioms to emulate the data types that they describe. This can be achieved by re-generating implementations from our discovered specifications (e.g., [33, 38]). Using the generated implementation in parallel with the original implementation for realistic benchmark programs will allow us to study the soundness and completeness of our tool in practice.

6 Related Work

We now describe related work in algebraic specifications, dynamic invariant detection, automatic programming, static analysis, and testing.

6.1 Algebraic Specifications

We drew many ideas and inspirations from previous work in algebraic specifications for abstract data types [22]. Horebeek and Lewi [28] give a good introduction to algebraic specifications. Sannella *et al.* give an overview of and motivation for the theory behind algebraic specifications [41]. A book by Astesiano *et al.* contains reports of recent developments in the algebraic specification community [5].

Antoy describes how to systematically design algebraic specifications [3]. In particular, he describes techniques that can help to identify whether a specification is complete. His observations could be used in our setting; however, they are limited to a particular class of algebras.

Prior work demonstrates that algebraic specifications are useful for a variety of tasks. Rugaber *et al.* study the adequacy of algebraic specifications for a reengineering task [38]. They specified an existing system using algebraic specifications and were able to regenerate the system from the specifications using a code generator. Janicki *et al.* find that for defining abstract data types, algebraic specifications are preferable over the trace assertion method [6, 30]

6.2 Dynamic Invariant Detection

Recently, there has been much work on dynamic invariant detection [15, 44, 1, 23]. Dynamic invariant detection systems discover specifications by learning general properties of a program's execution from a set of program runs.

Daikon [15] discovers Hoare-style axiomatic specifications [27, 20]. Daikon is useful for understanding *how* something is implemented, but also exposes the full complexity of a given implementation. Daikon has been improved in many ways [16, 17, 13] and has been used for various applications including program evolution [15], refactoring [32], test suite quality evaluation [24], bug detection [23], and as a generator of specifications that are then checked statically [36].

Whaley *et al.* [44] describe how to discover specifications that are finite state machines describing in which order method calls can be made to a given object. Similarly, Ammons *et al.* extract nondeterministic finite state automata (NFAs) that model temporal and data dependencies in APIs from C code [1]. These specifications are not nearly as expressive as algebraic specifications, since they cannot capture what values are returned by the methods.

Our preliminary studies show that the current implementation of our tool does not scale as well as some of the systems mentioned above. However, we are unaware of any dynamic tool that discovers high-level specifications of the interfaces of classes. Also, unlike prior work, our system interleaves automatic test generation and specification discovery. All previous systems require a test suite.

6.3 Automatic Programming

Automatic programming systems [7, 9, 8, 2, 25, 43] discover programs from examples or synthesize programs from specifications by deduction. The programs are analogous to our specifications in that our specifications are high-level descriptions of examples. Algorithmic program debugging [42] is similar to automatic programming and uses an inductive inference procedure to test side-effect and loop-free programs based on input output examples and then helps users to interactively correct the bugs in the program. Unlike the above techniques whose goals are to generate programs or find bugs, the goal of our system is to generate formal specifications (which could, of course, be used to generate programs or find bugs).

6.4 Static Analysis

Program analyses generate output that describes the behavior of the program. For example, shape analyses [39] describe the shape of data structures and may be useful for debugging. Type inference systems, such as Lackwit [37] generate types that describe the flow of values in a program. Our system is a dynamic black-box technique that does not need to look at the code to be effective. However, various static techniques can be used to guide our system (for example, we have already experimented with mod-ref analyses).

6.5 Testing

Woodward describes a methodology for *mutation testing* algebraic specifications [45]. Mutation testing introduces one change (“mutations”) to a specification to check the coverage of a test set. Woodward’s system includes a simple test generation method that uses the signatures of specifications.

Algebraic specifications have been used successfully to test implementations of abstract data types [18, 40, 29, 4]. One of the more recent systems, Daistish [29] allows for the algebraic testing of OO programs in the presence of side effects. In Daistish, the user defines a mapping between an algebraic specification and an implementation of the specification. The system then checks whether the axioms hold, given user-defined test vectors. Similarly, the system by Antoy *et al.* [4] requires the users to give explicit mappings between specification and implementation. Our system automatically generates both the mapping and the test suite.

Our work builds upon Doong and Frankl’s definition of observational equivalence and we were inspired by their algorithm for generating test cases from algebraic specifications [14]. Their system semi-automatically checks implementations against generated test cases. Later work improved on Doong and Frankl’s test case generation mechanism [12] by combining white-box and black-box techniques. Our tool can potentially benefit from employing static analysis of the code when generating test cases (white box testing).

In addition to the above, prior work on test-case generation [26, 11, 34] is also relevant to our work, particularly where it deals with term generation. Also, methods for evaluating test suites or test selection [46, 19] are relevant. We do not use these techniques yet but expect that they will be useful in improving the speed of our tool and the quality of the axioms.

Korat is a system for automated testing of Java programs [10]. Korat translates a given method’s pre- and post-conditions into Java predicates, generates an exhaustive set of test cases within a finite domain using the pre-condition predicate and checks the correctness of the method by applying the post-condition predicate. Our approach for generating terms borrows ideas from Korat.

7 Conclusion

We describe a dynamic approach for automatically discovering algebraic specifications from Java classes. These specifications are in the form of axioms in

terms of public methods of classes. Thus, they describe the observable behavior of classes and are not burdened by implementation details. Since our approach is dynamic our system is neither sound nor complete. However, compared to other dynamic systems, our system generates its own test cases (terms) and can thus keep generating more terms until it attains adequate confidence in the discovered axioms. For example, our system may determine that the basic block coverage for the test cases is inadequate and may decide to generate further test cases.

Our experiments with a number of Java classes reveals that our system generates axioms that are both correct and useful for understanding and using these classes. Our experiments also reveal some situations when our approach fails to discover certain axioms. More specifically, we find situations where conditional axioms would be useful.

Finally, our approach is not specific to Java but can be applied to other object-oriented languages that have sufficient reflection capabilities.

Acknowledgements

We thank Michael Burke, Dan Connors, Daniel von Dincklage, Matthias Hauswirth, Martin Hirzel, James Martin, Christoph Reichenbach, William Waite, Alexander Wolf, and the anonymous POPL and ECOOP reviewers for their insightful comments and suggestions on the paper and on earlier versions.

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. D. Angluin. Inference of reversible languages. *Journal of the ACM (JACM)*, 29(3):741–765, 1982.
3. S. Antoy. Systematic design of algebraic specifications. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, Pittsburgh, Pennsylvania, 1989.
4. S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1), Jan. 2000.
5. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification*. Springer, 1999.
6. W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Information Systems Methodology: Proceedings, 2nd Conference of the European Cooperation in Informatics, Venice, October 1978; Lecture Notes in Computer Science*, volume 65. Springer Verlag, 1978.
7. A. W. Biermann. On the inference of turing machines from sample computations. *Artificial Intelligence*, 3:181–198, 1972.
8. A. W. Biermann. The inference of regular Lisp programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8:585–600, Aug. 1978.
9. A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, Sept. 1976.

10. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *International Symposium on Software Testing and Analysis*, Rome, Italy, July 2002.
11. U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, Portland, Oregon, 2000.
12. 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.
13. N. Dodoo, A. Donovan, L. Lin, and M. D. Ernst. Selecting predicates for implications in program analysis. <http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.pdf>, March 2002.
14. R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering*, 3(2), Apr. 1994.
15. 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.
16. M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, June 2000.
17. M. D. Ernst, W. G. Griswold, Y. Kataoka, and D. Notkin. Dynamically discovering program invariants involving collections. TR UW-CSE-99-11-02, University of Washington, 2000. revised version of March 17, 2000.
18. J. Gannon, P. McMullin, and R. Hamlet. Databstraction implementation, specification and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.
19. T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.
20. D. Gries. *The science of programming*. Texts and monographs in computer science. Springer-Verlag, 1981.
21. J. V. Guttag, J. J. Hornig, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer Verlag, 1993. (out of print).
22. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.
23. S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.
24. M. Harder, B. Morse, and M. D. Ernst. Specification coverage as a measure of test suite quality. September 25, 2001.
25. S. Hardy. Synthesis of Lisp functions from examples. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 240–245, 1975.
26. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169–180. ACM Press, 1990.
27. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

28. I. V. Horebeek and J. Lewi. *Algebraic specifications in software engineering: an introduction*. Springer-Verlag, 1989.
29. 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.
30. R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *ACM Transactions on Software Engineering*, 27(7), July 2001.
31. B. Jeng and E. J. Weyuker. A simplified domain-testing strategy. *ACM Transactions on Software Engineering*, 3(3):254–270, July 1994.
32. Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *International Conference on Software Maintenance*, Florence, Italy, 2001.
33. H. Lin. Procedural implementation of algebraic specification. *ACM Transactions on Programming Languages and Systems*, 1993.
34. V. Martena, A. Orso, and M. Pezze. Interclass testing of object oriented software. In *Proc. of the IEEE International Conference on Engineering of Complex Computer Systems*, 2002.
35. J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.
36. J. W. Nimmer and M. D. Ernst. Automatic generation of program specifications. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA), Rome*, July 2002.
37. R. O’Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*, pages 338–348, 1997.
38. S. Rugaber, T. Shikano, and R. E. K. Stirewalt. Adequate reverse engineering. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering*, pages 232–241, 2001.
39. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.
40. 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.
41. D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.
42. E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertation 1982. MIT Press, 1982.
43. P. D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.
44. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the International Symposium of Software Testing and Analysis*, 2002.
45. M. R. Woodward. Errors in algebraic specifications and an experimental mutation testing tool. *IEEE Software Engineering Journal*, 8(4):237–245, July 1993.
46. H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.