A Technique for Mutation of Java Objects

James M. Bieman Sudipto Ghosh Roger T. Alexander
Computer Science Department
Colorado State University
Fort Collins CO 80523, USA
{bieman, ghosh, rta}@cs.colostate.edu

Abstract

Mutation analysis inserts faults into a program to create test sets that distinguish the mutant from the original program. Inserted faults must represent plausible errors. Standard transformations can mutate scalar values such as integers, floats, and character data. Mutating objects is an open problem, because object semantics are defined by the programmer and can vary widely. We develop mutation operators and support tools that can mutate Java library items that are heavily used in commercial software. Our mutation engine can support reusable libraries of mutation components to inject faults into objects that instantiate items from these common Java libraries. Our technique should be effective for evaluating real-world software testing suites.

1. Introduction

Mutation analysis aids in the assessment of the adequacy of tests [9]. It involves the modification of programs to see if existing tests can distinguish the original program from the modified program. Traditionally, syntactic modifications have been used. A set of *mutation operators* generate the syntactic modifications, which are determined by the language of the program being tested, and the mutation system used for testing. Mutation operators are created with one of two goals: to induce simple syntax changes based on errors that programmers typically make (such as using the wrong variable name), or to force common testing goals (such as executing each branch).

One way to conduct program modification is to change data values while a program is running. We are developing a mechanism to conduct mutation analysis of objects rather than just scalar data values. We inject faults into the objects at runtime. Others have used runtime fault injection, for example, Voas uses fault injection to measure testability [12].

Injected faults must represent plausible errors. Unlike hardware, where a combination of simple faults can be used to model any real fault, software fault models are hard to develop. Mutation of scalar values is straightforward, because their semantics are well understood. For example, we can add one to an integer value. Mutating objects that are instances of user defined types is more difficult. There is no obvious way to modify such objects in a manner consistent with realistic faults, without writing custom mutation methods for each object class. Our approach is to inject faults into objects that instantiate items from common Java libraries.

We address the following issues in the paper:

1. *Derivation of mutation operators for Java*: Prior work defines operators to mutate statements, operators, constants and variables for FORTRAN [11], C [1, 2, 3], Ada [10] and Java [6]. We describe mutation faults that can be injected into Java objects.
2. *Generation of mutants by applying the operators to programs*: Prior work focuses on compile-time or static mutation [9]. We show how to generate Java program mutants during program execution.

2. Mutation Operators for Java

Mutation operators defined for procedural languages are also applicable to Java. Java includes additional features related to the object-oriented paradigm. Offutt et al. defined mutation operators for Ada [10]; these operators address some object-oriented features; they do not address inheritance and are limited to properties within a class.

Kim et al [6] propose mutation operators for Java based on deviations of Java language constructs. The mutation operators do not take object semantics into account. Several operators create mutants that do not compile.

Objects have state and methods implement transitions from one state to another. Doong and Frankl's [4] ASTOOT uses algebraic specifications to generate method test sequences. Kirani and Tsai [7] describe a method sequence testing method: testers select sequences of methods in varying orders and length. Kung et al. [8] describe testing based on state transition diagrams. Mutation operators that are applied to program code are not sufficient to ensure that objects will go through different states.

Interface mutation identifies errors that programmers may make in defining, implementing and using interfaces. Integration mutation operators test the connections between two modules by mutating module interfaces [3].

Ghosh and Mathur [5] use interface mutation on distributed object systems that use CORBA, DCOM and Java-RMI. Interface mutation operators change parameter values in method calls defined in an interface.

The above mutation operators can be easily applied when the parameters are scalars. A simple object mutation is to make an object reference null. A more plausible approach is to modify the state of an object. State mutation operators cannot be applied statically to a program, because the state of the object depends on program execution. Interface mutation operators will not reveal state errors caused by specific method sequences. These errors depend on the order of operation and not the values of the arguments.

3. Mutation Faults for Java Objects

Instead of defining mutation operators for each class in the Java API, we define operators that apply to a whole group of classes that implement a certain interface. We examine mutation operators that apply to the following interfaces:

- Container types defined in the interfaces, *Collection* and *List* in the package *java.util*.
- Iterators defined in the interface *Iterator* in the package *java.util*.
- InputStream defined in the abstract class *InputStream* in the package *java.io*.

We also define default mutation operators.

```
1.     class C {
2.         ...
3.         public void m (Foo f) {
4.             ...
5.             ...
6.         }
7.         ...
8.     }
```

Figure 1. Code of Class C

Injecting The Faults. Figure 1 shows a class *C* containing a method *m()* with parameter *f* which references an object of type *Foo*. Code inside *m* (not shown) uses the parameter *f* for computation. We can mutate the object bound to *f* before it is used by inserting, just after line 3, the following statement:

```
f = (Foo) ObjectMutationEngine.mutate(f);
```

We can also mutate an object returned by a method, as shown in Figure 2, placing the *mutate* call just prior to the *return* statement. The *ObjectMutationEngine* implements the *mutate* methods. Inserting calls to the *ObjectMutationEngine* is relatively easy using a code instrumenter that builds a parse tree and inserts calls to *mutate* into certain nodes in the tree. The *ObjectMutationEngine* is described in Section 4.

```
class C {
    ...
    public Bar m (Foo f) {
        ...
        Bar b;
        ...
        return b;
    }
    ...
 }
```

Figure 2. Mutation of Return Statement

3.1. Default mutation operators

Mutating an instance of an arbitrary user-defined type modifies the fields of the object. The Java reflection API enables us to identify the types of objects. If the field is a scalar, we apply traditional scalar mutation operators:

1. Increment the value by 1
2. Decrement the value by 1
3. Set the value to a constant

Our new mutation operators apply to fields that are objects. If the semantics of the objects are known, it is easier to select the operators. Default operators apply when the semantics are not known. A default operator might make an object reference *null*. Such a mutation will probably raise a *NullPointerException*, limiting its utility. Moreover this is an operation that is applied to the reference, not the object. This mutation treats a variable that refers to an object as having an underlying type of “reference to object” and takes advantage of knowledge of the semantics of variables of this sort. It can only mutate an object reference by assigning it a value, testing for equality between two such variables, and dereferencing its value.

Another mutation operator for arbitrary object types is one that recursively applies mutation operators to the nested fields until the scalar fields are reached. The following operators can also be applied:

1. Cloning the object referred to by the variable and assigning the reference to the clone to the variable. This tests the sensitivity of a program to the object's identity rather than its state.
2. Creating a new object whose type is compatible with the declared type T of the object reference — we instantiate a new object whose type is a descendant of T.

3.2. Mutation operators for containers

Operators for the Collection Interface.

1. **Make the Collection empty:**

   In Java every *Collection* needs to implement the method *clear()*. The mutate method for making the *Collection* empty just invokes the *clear()* method on this object.

2. **Remove an element from the Collection:**

   The element to be removed from the *Collection* could be the first, last or some random element. Every *Collection* provides a *remove()* method that takes an object as a parameter. We can index the array of objects returned by the *Collection.toArray()* method to select any object to be removed.

3. **Add an element to the Collection:**

   We can add some arbitrary element to the *Collection* using the *add()* method provided in any *Collection*. This element can be a clone of some existing element, or it can be generated by the element constructor that takes no parameters or the default constructor of the element.

4. **Mutate the elements:**

   For every element inside the *Collection*, the mutate method for the element's type can be invoked.

5. **Reorder m of the n elements in the Collection:** Since there is no notion of order embodied in a *Collection*, and there it no method provided in the API that can manipulate the order, we cannot apply the reorder operator to the *Collection* interface directly. We can still use the method *toArray()* to obtain an array of the objects, reorder the array and create a new *Collection* from the objects in the array.

Operators for the List interface. The mutation operators defined for the *Collection* interface also apply to the *List* interface. The *Collections* class provides a number of static methods, such as *shuffle(List list)* and *shuffle(List list, Random rnd)* that can be used to reorder the elements in the *List*.

Operators for container implementations. The specific semantics of implementations can be used to mutate container objects. For example, a binary tree has a notion of ordering. This notion can be used to implement a **reorder** mutation operator.

3.3. Operator for the Iterator interface

The *Iterator* interface in the Java API generates the next element in the iteration using the method *next()*. A **skip** operator makes the iterator *skip* elements. It is implemented by a *mutate* method that calls the *next()* method one or more times.

The **skip** operator does not affect all instances of the mutated type but just the instance held by some client. Since we are only mutating one iterator and not the original container, any other client using the container or a different iterator over the container will not see a difference.

3.4. Mutation operator for inputstreams

The *InputStream* abstract class in the Java API provides a method called *skip(long n)* which skips over and discards *n* bytes of data from this input stream. We define a mutation operator for inputstreams that results in the skipping of bytes of data. This mutate method will call the *skip()* method with an appropriate length parameter.

4. Architecture of Mutation Engine

We create a special mutator class for each type or family of types to be mutated. Each mutator class implements the Mutator interface, shown in Figure 3, which specifies one operation, *mutate()*. Mutators are named as *<base type>Mutator*. For an object of type *A*, its mutator will be named *AMutator*.

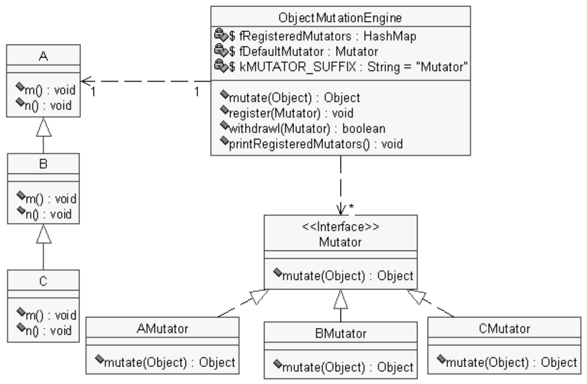

Figure 3. Object Mutation Engine with 3 Mutators.

A preprocessor instruments the code with invocations of *ObjectMutationEngine.mutate()* described in Section 3. The *mutate* method forwards the mutation request to the appropriate Mutator object. *ObjectMutationEngine.mutate()* identifies the actual class of its object parameter and any interfaces that it implements using the Java reflection API; then it looks for a match among a collection of registered Mutator objects. A Mutator object, *theMutator*, matches an object to mutate, *mutationCandidate*, if

1. The base type of *theMutator* is that of the actual type of the *mutationCandidate* (the declared type in the original program of the variable that refers to *mutationCandidate* is not known by *theMutator*),
2. The base type of *theMutator* is that of an interface implemented by the *mutationCandidate*, or
3. The base type of *theMutator* is a parent of the type of the *mutationCandidate*.

Figure 3 shows the *ObjectMutationEngine* and three classes *A*, *B*, and *C* with the associated mutators *AMutator*, *BMutator* and *CMutator*. Each Mutator class implements the *mutate* method (of interface *Mutator*) that takes an object and returns a mutated object.

The *ObjectMutationEngine* is the heart of the application. It maintains a table of registered mutators (*fRegisteredMutators*). Every *Mutator* registers with the *ObjectMutationEngine* using the *register(Mutator)* method. The *register()* method determines the class name of the *Mutator* using the Java reflection API. A mutator may be removed from the table using the *withdraw()* method.

ObjectMutationEngine.mutate(Object) is passed an instance of class *A*, *B*, or *C* as a parameter. The mutate method determines the appropriate mutator key by appending the string *Mutator* to the parameter's class name, and uses this key to find the appropriate mutator class from the *RegisteredMutators* table. If there is a match, the mutation request is forwarded to the matching mutator. Otherwise, the search continues up the inheritance hierarchy to find the nearest applicable mutator.

The *ObjectMutationEngine* has a fairly simple design, yet it can support a wide variety of mutations. The design of libraries of Mutator classes that match Java library classes and interfaces is an ongoing activity.

5. Conclusions and Future Work

Our technique for performing mutation analysis on object-oriented programs injects faults into objects. Reusable libraries of mutation components can inject plausible faults into objects that instantiate items from common Java libraries. Since Java library items are heavily used in commercial software, the technique should be effective for evaluating the real-world software testing suites. An object mutation engine implements the technique. We are now testing the effectiveness of this technique.

Currently, each mutator class has one mutate method. After we define additonal mutation operators, the mutator classes will have multiple mutate methods.

The design of the *ObjectMutationEngine* is very flexible. It can inject a wide variety of mutations into running programs, and can be extended to support new fault models.

References

[1] H. Agrawal, R. DeMillo, R. Hathaway, W. M. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue Univ., 1989.

[2] M. E. Delamaro, J. C. Maldonado, M. Jino, and M. L. Chaim. mutantes PROTEUM: A test tool based on mutation analysis. In *Software Tools Proc. VII Brazilian Symp. on Software Engineering*, 1993.

[3] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Integration Testing Using Interface Mutation. *Proc. ISSRE '96*, pp. 112–121, 1996.

[4] R.-K. Doong and P. G. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Trans. Software Engineering and Methodology*, 3(2):101–130, April 1994.

[5] S. Ghosh and A. P. Mathur. "Interface Mutation". *Proc. of MUTATION 2000*, pp. 112–123, 2000.

[6] S. Kim, J. A. Clark, and J. A. McDermid. "Class Mutation: Mutation Testing for Object Oriented Programs". *Proc. FMES*, 2000.

[7] S. Kirani and W. T. Tsai. "Method Sequence Specification and Verification of Classes". *Object-Oriented Programming*, pp 28–38, 1994.

[8] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. "Change Impact Identification in Object Oriented Software Maintenance". *Proc. IEEE Int. Conf. Software Maintenance*, pp. 202–211, 1994.

[9] A. P. Mathur. *Encyclopedia of Software Engineering, J. Marciniak, Editor*, chapter Mutation Testing, pp. 707–713. Wiley Interscience, 1994.

[10] A. J. Offutt, J. Voas, and J. Payne. Mutation Operators for Ada. Technical Report ISSE-TR-96-09, Information & Software Systems Engineering, George Mason Univ., 1996.

[11] R. A. DeMillo et al. An Extended Overview of the MOTHRA Testing Environment. *Proc. Workshop of Software Testing, Verification and Analysis*, 1988.

[12] J. M. Voas. PIE: A Dynamic Failure-based Technique. *IEEE Trans. Software Engineering*, 18(8):717–727, 1992.