

# AJAX: Automating an Informal Formal Method for Systematic JUnit Test Suite Generation

David Stotts

Dept. of Computer Science  
Univ. of North Carolina at Chapel Hill  
[stotts@cs.unc.edu](mailto:stotts@cs.unc.edu)

## Abstract

The JUnit testing tool is widely used to support the central XP concept of “test first” software development. We previously introduced an informal use of formal ADT semantics for guiding JUnit test method generation [16]. Called JAX (for *JUnit AXioms*), the method does not require the programmer to learn and use any formal notation in order to gain some benefits from a formal method; no notations are involved other than Java. Experiments showed that manual application of the JAX method produces JUnit test suites that expose more coding errors than ad hoc JUnit test case development.

The previous research emphasized a *manual* programming procedure as a way to work the formal benefits into practice more easily. In this paper we discuss AJAX (*Automated JAX*), extensions we have made to the JUnit Java classes to *automate* the application of JAX to various degrees. The tradeoff is that for the highest degree of automation, the programmer will need to learn and develop formalisms (ADT axioms); to ease this potential burden we use a programming notation (ML) for the formalism. Our JUnit extensions will be available for download.

## 1 Motivation and background

Regression testing has long been recognized as necessary for having confidence in the correctness of evolving software. Programmers generally do not practice thorough tool-supported regression testing, however, unless they work within a significant industrial framework. JUnit [1,2,3] was developed to support the “test first” principle of the XP development process [4]; it has had the side effect of bringing the benefits of regression testing to the average programmer, including independent developers and students. JUnit is small, free, easy to learn and use, and has obtained a large user base in the brief time since its introduction in the XP community. JUnit and its supporting documentation are available at <http://www.junit.org>. The tool is useful in any software development process (not just in XP), and since it is available for more than 20 source languages, it is useful to more than just Java programmers.

The basic JUnit testing methodology is simple and effective. However, it still leaves software developers to decide if enough test methods have been written to exercise all the features of their code thoroughly. The documentation supporting JUnit does not prescribe or suggest any *systematic* methodology for creating complete and consistent test suites. Instead it is designed to provide automated bookkeeping, accumulation, and execution support for the manner in which a programmer is already accustomed to developing test suites.

We have developed and experimented with a systematic test suite generation method we call JAX (for *JUnit AXioms*), based on Guttag’s algebraic semantics of Abstract Data Types (ADTs) [5,6,7]. Following the JAX method leads to JUnit test suites that completely cover the possible behaviors of a Java class. We refer to JAX as an *informal formal method* because, while it is based in the formal semantics of abstract data types, the Java programmer and JUnit user need use no formalisms beyond Java itself to take advantage of the guidance provided by the method. Our approach is simple and systematic. It will tend to generate more test methods than a programmer would by following the basic JUnit practice, but our preliminary experiments show this extra work produces test suites that are more thorough and more effective at uncovering defects.

We will presume the reader has some familiarity with both axiomatic ADT semantics as well as the JUnit testing tool, and will not cover them in detail here.

## Summary of Manual JAX

We briefly summarize the manual JAX procedure described in [16]. The steps are:

1. Design the method signatures for the Java class to be written (the target class)
2. Decide which methods are canonical, dividing the methods into 2 categories
3. Create the left-hand sides (LHS) of the axioms by crossing non-canonical methods on canonical ones
4. Write an *equals* function that will compare two elements of the target class
5. For each *axiom* left-hand side, write one test method in the test class, using the abstract *equals* where appropriate in the JUnit *assert* calls.

The first 3 steps are the basic process of creating ADT axioms, except we stop before requiring the designer to create the axiom right-hand sides. Instead, we expect that behavior to get expressed in the Java code produced in step 5. The last step is the key to the effectiveness of the method. The documentation distributed with JUnit provide examples where each method in the target class causes creation of a corresponding method in the test class. JAX calls for creation of one test class method *for each axiom*; it is this crossing, and testing method behavior in combinations, that makes the process systematic, thorough, and complete compared to *ad hoc* development of JUnit test suites.

As an example, consider the method signatures of this ADT for a bounded stack:

```
new:      int      --> BST
push:    BST x E  --> BST
pop:     BST      --> BST
top:     BST      --> E
isEmpty: BST      --> bool
isFull:  BST      --> bool
maxSize: BST      --> int
getSize: BST      --> int
```

Here are some axioms defining its behavior, expressed in ML as a formal spec notation (they are executable specs... download an ML interpreter and give them a try):

```
(*
  Algebraic ADT specification
  full axioms for BST (bounded set of int)
*)

datatype BST =
  | New of int
  | push of BST * int ;

fun isEmpty (New(n)) = true
  | isEmpty (push(B,e)) = false ;

fun maxSize (New(n)) = n
  | maxSize (push(B,e)) = maxSize(B) ;

fun getSize (New(n)) = 0
  | getSize (push(B,e)) = if getSize(B)=maxSize(B)
                        then maxSize(B) else getSize(B)+1 ;

fun isFull (New(n)) = n=0
  | isFull (push(B,e)) = if getSize(B)>=maxSize(B)-1
                        then true else false ;

exception topEmptyStack;

fun top (New(n)) = raise topEmptyStack
```

```

| top (push(S,e)) = if isFull(S) then top(S) else e ;

fun pop (New(n)) = New(n)
| pop (push(S,e)) = if isFull(S) then pop(S) else S ;

```

The informality of this formal method application enters here. For manual JAX, we do not need to write out the axioms *completely* in the ML formalism (or any other formal notation). Rather, all we need is the left hand sides of the axioms – the combinations of the non-canonical methods applied to the canonical ones. The programmer will create the right-hand side behavior *en passant* by encoding it in Java directly in the methods of the corresponding JUnit test class. The formal ADT semantics tell us which method combinations to create test for, but we flesh out the axiomatic behavior directly in JUnit and Java.

To continue the example, consider the final axiom above, with LHS being an application of pop to the stack remaining after a push method invocation. In our JUnit test class we would generate a test method *testPopPush()* like this:

```

protected void setUp() { // establish the stack objects used in testing
    stackA = new intStackMaxArray(); // defined as max 2 for ease of filling
    stackB = new intStackMaxArray();
}

public void testPopPush() {
// axiom: pop (push(S,e)) = if isFull(S) then pop(S) else S

    // do the not(isFull(S)) part of the axiom RHS
    int k = 3;
    stackA.push(k);
    stackA.pop();
    assertTrue( stackA.equals(stackB) ); // use of our abstract equals function

    //now test the isFull(S) part of the axiom RHS
    stackA.push(k); // 1 element
    stackA.push(k); // 2... now full
    stackB.push(k); // 1 element
    stackB.push(k); // 2.. now full
    assertTrue(stackA.equals(stackB)); // expect true

    stackA.push(k); // full... so push is a noop
    stackA.pop(); // now has 1 elt
    stackB.pop(); // now has one elt
    assertTrue( stackA.equals(stackB) ); // expect true
}

```

Note that this test method has Java code to exercise each of the two situations described in the RHS of the axiom. In this example, the axiom was written first for illustrative purposes. In a fully informal JAX application, the programmer may well have invented appropriate behavior as she wrote the JUnit code and never expressed the right-hand side of the axiom any other way.

JAX is clearly going to generate more test methods that ad hoc use of JUnit, but they are systematically generated and together cover all possible behaviors of the ADT. Consistent and complete coverage of the target class behavior is guaranteed by the proof that the axioms formally define the complete ADT semantics [5].

## 2 AJAX: Automating JAX

Automating JAX will leverage its error-finding capabilities. The tradeoff is a decrease in informality, and an increase in formality of our use of ADT semantics. We are developing Java classes that extend the classes of JUnit to allow both full and partial automation of JAX. We call this tool AJAX (*Automated JAX*).

Both automation approaches grow out of an earlier testing project for C++ [9]. In full automation, a developer must write full specs for a class (as with our ML example for BST), and also must provide descriptions of data points

for the test scaffold; a tool will then create the Java source for a JUnit test class directly from the specs. In partial automation, a developer will manually write JAX-based JUnit test classes as discussed above, and supply a list of test points for each axiom so that each test methods can be invoked numerous times on the appropriate data points. Partial automation allows programmers to write more compact test methods that nonetheless have more extensive testing power. We now expand on each concept.

## Related Research

ASTOOT [10] is a related system for testing based on formal specifications. ASTOOT uses a notation similar to Parnas' trace specs [11,12] instead of algebraic ADT semantics. This work was presented in the context of the language Eiffel, and has not been carried forward into commercial quality tools. We also think the use of the functional notation of algebraic ADT axioms is an advantage over the trace spec approach; such axiom can be expressed in a functional programming language (we give our examples in ML) giving executable specs.

Larch [13,14,15] is another research effort in which formal program specs are used to gain leverage over software problems. In Larch, program specifications have a portion written in Guttag's functional style, along with a second portion written to express semantic-specific details for a particular programming language. A significant body of work exists on using Larch for program verification, supported by automated theorem provers. We do not know of any testing methodologies based on Larch specs however.

We described experiments in using JAX in a recent paper [16]. The method we describe here for automating JAX is similar to those of DAISTS [8] and Daistish [9]. The DAISTS system used Guttag's algebraic specification of abstract data types to write a test oracle for an ADT implemented in a functional language (SIMPL-D). Daistish expanded the work of DAISTS into the object-oriented domain (for C++) by solving problems related to object creation and copying that were not found in functional languages. Daistish automated the creation of the test oracle, leaving the programmer to write axiomatic specifications and test points.

## Full automation

Complete automation requires full ADT specs for the class under development. The full set of axioms for the bounded set BST was given previously, using the functional language ML as our formal specification notation: These specs are fully executable; we encourage you to download a copy of SML 93 and try them. The `datatype` definition is where the canonical constructors are defined. The axioms are ML function definitions, using the pattern-matching facility to create one pattern alternative for each axiom.

We use the left and right sides of the axioms as definitions of two different sequences of actions to be performed by the Java code in a JUnit test method. Each axiom is a declaration of the equality of the LHS to the RHS, so we compute the results specified by each side and then compare them for equality (either with a built-in "=" for a base type, or using the programmer-supplied *equals* for objects).

For example, consider this ML function; it gives 2 axioms of an ADT defining the behavior of a FIFO Queue:

```
fun remove (new()) = new()
  | remove (add(Q,e)) = if size(Q)=0 then Q else add(remove(Q),e) ;
```

The first axiom formally encodes the design decision that if you try to remove something from an empty queue it is like a no-op. The second axiom specifies two situations related to removing an item from a queue that has just had something added to it. If you add something to an empty queue and then remove an element, the result is an empty queue; if you add something to a queue that is *not* empty and then remove an item, the result is the same as if you do the remove first, then the add.

AJAX creates JUnit test methods from these axioms. The method *testRemoveNew()* would be generated for the first axiom, and its Java code would do several things. First, it would run the operations indicated by the LHS: *remove(new())*. That is, it would create an empty queue with *new()*, and then invoke *remove()* on that queue. Next, the test method would perform the operations indicated by the RHS: *new()*. It would then compare the RHS result to the LHS result for equality. In this example, the code would most likely look like

```
(new().remove()).equals(new())
```

The second axiom is a bit more complicated and points out the need for more structure in AJAX. The operation *add* needs arguments that are queues and elements in queues. AJAX allows the specification of collection of test points to fill these roles; test points will get translated into the test scaffolding created in the *setUp()* and *tearDown()* methods of a JUnit *TestCase* class. Thus there is a known collection of variable names that are systematically substituted into the calls to *add* so that the RHS/LHS structures can be computed and compared. Such specs look like ML data added to the functions defining the axioms. For example, here are queue data points:

```
val q1 = new() ;
val q2 = add(new(),4) ;
val q3 = add(add(q2,7),12) ;
val e1 = 8 ;
val e2 = 19 ;
```

These ML data values tell the AJAX (JUnit) test runner to create Java objects in the test framework. Now we can create test method *testRemoveAdd()* and invoke the methods called for in the axiom, applying them one time for each *combination* of appropriately typed data values. So for one test we do *remove(add(q1,e1))* as one LHS and compare that for equality to *q1* (if *size(q1)=0*) or to *add(remove(q1),e1)* if *q1* is not empty. We then do the comparison again, but with *q2* and *e1* as parameters; then again with *q3* and *e1*, etc. until all possible parameter combinations have been used in the RHS/LHS method calls. AJAX gives the possibility of large numbers of tests being done without having to write manually all these combinations into the test methods. JUnit then makes these large numbers of tests permanent in a regression suite.

### Partial automation

For partial automation, we exploit the combinatorial power of specifying multiple data points, but we use manual JAX and do not use full formal axioms. A user writes a collection of data points as shown above (*q1*, *q2*, *e1*, etc.) but in Java notation rather than ML. This is similar to what would normally be done in the *setUp()* method of a JUnit test class. Instead of then manually (and repetitively) writing code in each appropriate JUnit test method to manipulate these data points by global name, the programmer creates the test methods that manual JAX dictates (one per axiom), with one difference: the methods are parameterized instead of referring to global variable names for the test objects being operated on. An AJAX class then wraps each parameterized test method in a similarly named method that fits the normal JUnit naming scheme (no arguments); the wrapper sets up loops and invokes the user's manually-generated test method on all possible combinations of the appropriate data points. In this way, a little hand coding can generate a huge number of tests, making it easier to test a wide range of the possible values in any particular data type.

## 3 Discussion and Conclusions

We have reviewed JAX, a systematic procedure for generating consistent and complete collections of test methods for a JUnit test class. JAX is based on a formal semantics for abstract data types and exploits *method interactions* rather than keying on individual methods. Theoretical results from formal semantics indicate that the interactions generated by JAX cover the possible behaviors of a class. JAX is unique in that formal methods are used for *guidance*, but the programmer is not saddled with formalisms; effective use of JAX requires only Java.

We presented AJAX, a collection of classes that extend the JUnit regression testing tool to automate (or partially automate) application of the JAX method. Automation of JAX with AJAX gives compact expression of extensive test coverage at the cost of giving up some of the informality of JAX.

### Acknowledgements

This work was supported by a grant from the United States Environmental Protection Agency, project # R82 – 795901 – 3.

## References

- [1] Beck, K., and Gamma, E., "JUnit Test Infected: Programmers Love Writing Tests," *Java Report*, July 1998, Volume 3, Number 7. Available on-line at: <http://JUnit.sourceforge.net/doc/testinfected/testing.htm>
- [2] Beck, K., and Gamma, E., "JUnit A Cook's Tour," *Java Report*, 4(5), May 1999. Available on-line at: <http://JUnit.sourceforge.net/doc/cookstour/cookstour.htm>
- [3] Beck, K., and Gamma, E., "JUnit Cookbook.." Available on-line at: <http://JUnit.sourceforge.net/doc/cookbook/cookbook.htm>
- [4] Beck, K., "Extreme Programming Explained," Addison-Wesley, 2000.
- [5] Guttag, J.V., and Horning, J.J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica* 10 (1978), pp. 27-52.
- [6] J. Guttag, E. Horowitz, D. Musser, "Abstract Data Types and Software Validation", *Communications of the ACM*, 21, Dec. 1978, pp. 1048-1063.
- [7] J. Guttag, "Notes on Type Abstraction", *IEEE Trans. on Software Engineering*, TR-SE 6(1), Jan. 1980, pp. 13-23.
- [8] Gannon, J., McMullin, P., and Hamlet, R., "Data Abstraction Implementation, Specification, and Testing," *IEEE Trans. on Programming Languages and Systems* 3( 3). July 1981, pp. 211-223.
- [9] Hughes, M., and Stotts, D., "Daistish: Sytematic Algebraic Testing for OO Programs in the Presence of Side Effects," *Proceedings of the 1996 International Syposium Software Testing and Analysis (ISSTA)* January 8-10, 1996, 53-61.
- [10] R.-K. Doong, P. Frankl, "The ASTOOT Approach to Testing Object-Oriented Programs", *ACM Trans. on Software Engineering and Methodology*, April 1994, pp. 101-130.
- [11] D. Hoffman, R. Snodgrass, "Trace Specifications: Methodology and Models", *IEEE Trans. on Software Engineering*, 14 (9), Sept. 1988, pp. 1243-1252.
- [12] D. Parnas, Y. Wang, "The Trace Assertion Method of Module Interface Specification", Tech. Report 89-261, Queen's University, Kingston, Ontario, Oct. 1989.
- [13] J. Guttag, J. Horning, J. Wing, "The Larch Family of Specification Languages", *IEEE Software*, 2(5), Sept. 1985, pp. 24-36.
- [14] J. Wing, "Writing Larch Interface Language Specifications", *ACM Trans. on Programming Languages and Systems*, 9(1), Jan. 1987, pp. 1-24.
- [15] J. Wing, "Using Larch to Specify Avalon/C++ Objects", *IEEE Trans. on Software Engineering*, 16(9), Sept. 1990, pp. 1076-1088.
- [16] Stotts, D., M. Lindsey, and A. Antley, "An Informal Formal Method for Systematic JUnit Test Case Generation," XP Universe 2002, Chicago, August 4-7, 2002; Lecture Notes in Computer Science 2418 (Springer), pp. 131-143.