An Informal Formal Method for Systematic JUnit Test Case Generation

David Stotts, Mark Lindsey, and Angus Antley

Dept of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

stotts@cs.unc.edu

April 2, 2002

An Informal Formal Method for Systematic JUnit Test Case Generation

David Stotts, Mark Lindsey, and Angus Antley

Dept. of Computer Science
Univ. of North Carolina at Chapel Hill
stotts@cs.unc.edu

Abstract. The JUnit testing tool is widely used to support the central XP concept of "test first" software development. While JUnit provides Java classes for expressing test cases and test suites, it does not provide or proscribe *per se* any guidelines for deciding what test cases are good ones for any particular class. We have developed a method for systematically creating complete and consistent test classes for JUnit. Called *JAX* (for *Junit Axioms*), the method is based on Guttag's algebraic specification of abstract data types. We demonstrate an informal use of ADT semantics for guiding JUnit test method generation; the programmer uses no formal notation other than Java. Experiments show that informal JAX-based testing finds more errors than an *ad hoc* form of JUnit testing. We also discuss how JUnit test method generation can be automated for developers who don't mind using partial or full formal specs.

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 it's introduction in the XP community. Given this audience, we will not go into any detail about its structure and usage. JUnit and its supporting documentation are available at http://www.junit.org .

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. 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 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 early experiments have been with this informal application of JAX; in the last section we discuss methods of employing more directly the formal notations of ADTs to gain more automation in test case generation, and we discuss the advantages and tradeoffs in automation.

1.1 Related work on formal spec-based testing

The method we describe 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.

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 or 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.

1.2 Structure of the report

In the following sections we first present a brief explanation of algebraic specifications of abstract data types, and why they are appropriate as formal semantics for objects. Following that we show how to use ADT specs manually to systematically develop a consistent and complete JUnit test class. We then discuss our preliminary experiments with several ADT implementations and the comparison of the number of errors found with JAX versus without. We end with a discussion of two ways JAX can be partially automated, generating JAX-based JUnit test suites from the formal specs directly.

2 Algebraic Specification of ADTs

The *raison d'etre* of JAX is to apply a systematic discipline to the otherwise informal (and possibly haphazard) process of creating effective test methods in JUnit test classes. A class to be developed is treated as an ADT (abstract data type), and the formal algebraic specification [5,6,7] of this ADT is then used as a guide to create a *complete and consistent* set of test methods. Guttag's work is fundamental and, we think, underused in modern software methodologies; for this contribution he was recently recognized as a software pioneer along with Dijkstra, Hoare, Brooks, Wirth, Kay, Parnas, and others [16]. We present here a brief summary of algebraic ADT semantics before we more fully explain JAX.

> *Note that the ADT formalism explained here is used as an intellectual guide for developing tests. Effective use of these ideas requires only Java, so the reader should not be overly concerned with the specific formalisms and notations shown.*

An ADT is specified formally as an algebra, meaning a set of operations defined on a set of data values. This is a natural model for a class, which is a collection of data members and methods that operate on them. The algebra is specified in two sections: the syntax, meaning the functional signatures of the operations; and the semantics, meaning axioms defining the behavior of the operations in an implementation-independent fashion. For example, consider BST, an ADT for a bounded stack of elements (of generic type E). The operation signatures are

```
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
```

In object-oriented terms, operation signatures correspond to the *interfaces* of the methods for a class. In Guttag's notation, ADT operations are considered functions in which the state of the object is passed in as a parameter, and the altered object state is passed back if appropriate. The operation *push(S,a)* in Guttag's ADT notation would thus corresponds to a method `S.push(a)` for a specific object S of this class.

In this example, *new* is a constructor that takes an integer (the maximum size of the bounded stack) and creates a BST that will hold at most that many elements. The *push* operation takes an E element and a BST (an existing stack) and adds the element to the top of the stack; though it has not yet been so specified, we will define the semantics to reflect our desire that pushing an element on a full stack does nothing to the state of the stack – effectively a *noOp*. Operation *maxSize* returns the bound on the stack (which is the parameter given to *new*); operation *getSize* tells how many items are currently in the stack. Operation *isFull* exists for convenience; it tells if there is no room in the stack for any more elements and can be done entirely with *maxSize* and *getSize*. The remaining operations behave as would be expected for a stack.

The semantics of the operations are defined formally with axioms that are systematically generated from the operation signatures. To do so, we first divide the operations into two sets: the *canonical operations[1]*, and all others. Canonical operations are those needed to build all elements of the type; these necessarily are a subset of the operations that return an element of the type. In this example, there are only three operations that return BST (*new, push, pop*), and the canonical operations are {*new, push*}. This implies that all elements of type BST can be built with *new* and *push* alone. Any BST that is built using *pop* can also be built with a simpler sequence of operations consisting only of *new* and *push*; for example, the stack produced by `push(pop(push(new(5),a)),b)` using *pop* is equivalent to the stack produced by `push(new(5),b)` not using *pop*.

Once a set of canonical operations is identified[2], one axiom is written for each *combination* of a non-canonical operation applied to the result of a canonical operation. This is a form of induction. We are defining the non-canonical operations by showing what they do when applied to every element of the type; we obtain every element of the type by using the output of the canonical operations as arguments. For example, considering the bounded stack ADT, one axiom is for `pop(new(n))` and another is for `pop(push(S,a))`.

With 2 canonical constructors and 6 non-canonical operations, we write a total of 6*2 = 12 axioms. These combinations generate mechanically the left-hand sides of the axioms; specifying the right-hand sides is where design work happens and requires some thought. An axiom is thought of as a re-writing rule; it says that the sequence of operations indicated on the left-hand side can be replaced by the equivalent (but simpler) sequence of operations on the right-hand side. For example

```
top(push(S,x)) = x    // ok for normal stack but not for bounded stack
pop(push(S,x)) = S    // ok for normal stack but not for bounded stack
```

---

[1] Guttag uses the term *canonical constructor*. We use *operation* instead of *constructor* to avoid confusion with the method that is invoked when an object is first created. That constructor method is called *new* in Guttag's vocabulary.

[2] Though it is not evident from this simple example, there can be more than one set of canonical operations for an ADT. Any valid set will do.

are two axioms for normal stack behavior. The first specifies that if an element x is pushed on top of some stack S, then the top operation applied to the resulting stack will indicate x is on top. The second says that if an element x is pushed onto some stack S, then the pop operation applied to the resulting stack will return a stack that is equivalent to the one before the push was done. For a bounded stack the proper behavior is slightly more complicated:

```
top(push(S,x)) = if isFull(S) then top(S) else x
pop(push(S,x)) = if isFull(S) then pop(S) else S
```

This says that if we push an element x onto a full stack S, then nothing happens, so a following pop will be the same as popping the original stack S; if S is not full, then x goes on and then comes off via the following pop; similar reasoning applies to top.

For those readers interested in more details, the full set of ADT axioms for BST is given in Appendix A. For JAX, the lesson to carry into the next section is that class methods can be divided in two groups (canonical and non-canonical) and that combining one non-canonical method applied to one canonical will define one axiom… and hence one test method for a JUnit test class, as will see.

3 Informality Level One: Manual JAX

Once the ADT axioms have been specified for the target class, we can systematically construct a corresponding JUnit test class. In summary, 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 the non-canonical methods on the canonical ones
4) write an *equals* function that will compare two elements of the target class (to compare two BSTs in our example)
5) write one test method in the test class for each *axiom* left-hand side, using the abstract *equals* where appropriate in the JUnit assert calls.

The last two steps are the keys. The papers explaining 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*.

The first level of informality enters the method here. 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-had side behavior *en passent* 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. For example, consider this BST axiom:

```
pop(push(S,e)) = if isFull(S) then pop(S) else S
```

The right-hand side gives a behavior definition by showing the expected outcomes when the method calls on the left-hand side are executed. The "=" equality that is asserted between the method sequence on the left and the behavior on the right is an abstract equality; in this case, it is equality between two stacks. To check this equality the programmer must supply a function for deciding when two stacks are equal, which is the function from item (4) in the list above. In this case, we get a method called *testPopPush()* in the JUnit test class for BST:

```
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 a section for each part of the right-hand side behavior. 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, 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].

A Note on the Importance of 'Equals'

In addition to writing the test classes for the axiom combinations, the tester must write an "equals" function to override the built in method on each object. This must be an abstract equality test, one that defines when two objects of the class are thought of being equal without respect to unimportant internal storage or representational issues. This gives us something of a chicken-and-egg problem, as this abstract *equals* is used with JUnit *assert* methods to indicate the correctness of the other methods of the target class. It is important, therefore, that the tester have confidence in the correctness of equals before trying to demonstrate the correctness of the other methods. We follow a procedure where the test methods are generated for equals like for other methods, by applying it to objects generated by the various canonical operations of the target class. In this example we wrote and populated the JUnit test methods

```
testNewNewEquals()        testNewPushEquals()
testPushNewEquals()       testPushPushEquals()
```

and run them first. This has the effect of checking the canonical operations for correct construction of the base elements on which the other methods are tested, as well as checking equals for correct comparisons. Doing the extra work to structure these tests and including them in the regression suite allows changes in the constructors to be checked as the target class is evolved. Given the introspective nature of the *equals* tests, the programmer may wish to create them as a separate but related JUnit test class rather than bundling them into the same test class as the other test methods. The JUnit *suite* can then be used to bundle the two together into a full JAX test for the target class.

6

3.3 Preliminary Experimental Results

We have run several experiments to gauge the effectiveness of the JAX methods. While these early studies were not fully controlled experiments, we think the results are encouraging enough to continue with more extensive and controlled experiments. We coded several non-trivial ADTs in Java and tested each two ways:

- basic JUnit testing, in which one test-class method is generated *for each method* in the target class
- JAX testing, in which one test-class method is generated *for each axiom* in the ADT specification.

Each of the preliminary experiments involved two students (author and tester). For each ADT tested, the tester wrote axioms as specifications of the target class. The axioms were given to the author, who wrote Java code to implement the class. While the author was writing the source class, the tester wrote two test classes -- one structured according to basic JUnit testing methodology, and the other structured according to the JAX methodology. After the author completed the source code for the target class the tester ran each JUnit test collection against the source class, and recorded the number of errors found in each. In each case the source class was the same, so the sole difference was the structure and number of the test methods in the test class. Over the course of the studies we used 2 different testers and 2 different authors.

We examined in this way 5 Java classes, starting with some fairly simple ones to establish the approach and progressing to a pair of cooperating classes with a total of 26 methods between them[3]. In the non-trivial cases, the JAX test class uncovered more source class errors than did the basic JUnit test class. As a warm up we wrote a JAX test class for the `ShoppingCart` example distributed with JUnit. The JUnit test class that comes with it containe no errors, and none were found via JAX either. Another of the classes tested was the bounded stack (BST) discussed previously. We implemented it twice, with two different internal representations; in each case, JAX testing found one error and basic JUnit test uncovered none.

The most complicated example we studied was a pair of cooperating classes implementing a simple library of books. The interface is defined as follows:

```
public book( String title, String author )
public void addToWaitList(int pid)
public void removeFromWaitList(int pid)
public void checkout(int pid)
public void checkin(int pid)
public int getNumberAvailable()
public String getTitle()
public String getAuthor()
public boolean isInCheckoutList(int pid)
public boolean isInWaitList(int pid)
public int getPidWaitList(int index)
public void addACopy(int)
public void removeACopy(int)
public boolean equals(book b)
public int getSizeOfWaitList()
public int getSizeOfCheckoutList()
public int NextWaiting()
public int getNumberCheckedOut()

public library()
public book getBook(String title, String author)
public void addBook(String title ,String author)
public void removeBook(String title ,String author)
public void checkoutBook(String title, String author,int pid)
public void checkinBook(String title, String author,int pid)
public boolean equals(library l)
public boolean isAbleToBeRemoved(String title, String author)
```

---

[3] All Java code for these classes, and the JUnit test classes corresponding to them, can be obtained online at http://rockfish-cs.cs.unc.edu/JAX/

```
public boolean isAvailable(String title, String author)
public boolean isAbleToBeCheckedIn (String title, String author,int pid)
public int getNumberOfBooks()
```

This test involved two different classes interacting. In particular, methods in *Library* invoke methods in *Book*; to manage this, we developed and tested *Book* first, then developed and tested *Library*. We repeated this two-step process once for normal JUnit tests, and once for the JAX methodology. Obviously, if there are errors in *Book* we would expect to see the error propagate into errors in the *Library* test as well. Here are the results for each test class when run in JUnit:

```
Test              Failures
----------------------------
BookTest            0
BookAxiomTest       13
LibraryTest         0
LibraryAxiomTest    15
```

We have been using the term "error" a bit loosely. This table shows counts of failed JUnit tests; because of the interactions being tested in JAX, a single code error (bad method) is likely to cause several JUnit failures. Moreover, we suspected that the 13 failures found in class *Book* would be due to errors that would then propagate into many of the failures seen in class *Library*. We decided to fix the errors in *Book* and retest. On examination, we found 3 distinct flaws in the implementation of *Book*. On rerunning JUnit, the 15 failures found by the JAX test class still remained:

```
Test              Failures
----------------------------
BookTest            0
BookAxiomTest       0
LibraryTest         0
LibraryAxiomTest    15
```

Implications of these numbers

Following JAX manually will require a programmer to write more test methods in a test class than he would with a normal JUnit discipline for the same test class. This is because the JAX approach requires one test method per axiom. For example, in the ADT bounded stack (BST), there are 8 methods, with 2 being canonical. Normal JUnit testing produced 8 test methods (or 7, as the constructor was not tested); JAX application produced $2 * 6 = 12$ methods from axioms alone; creation of the equals function and dealing directly with constructors brought the total to 14 methods.

In our early experiments, we are finding about 70% more test methods are being written in a JAX test suite for small classes (those with on the order of 8 methods). For the larger *Book* class, there are 17 methods, of which 4 are canonical (*book, checkOut, addToWaitList, addACopy*). This means we wrote $13*4 = 52$ test methods for it.

Given the combinatorial nature of axiom creation, the rough estimate for number of test methods in a JAX test class will be on the order of $(n^2)/4$ where *n* is the number of target class methods. Since good OO design encourages smaller classes, the quadratic factor should not be a major issue. An IBM fellow who was listening to an earlier talk on our work commented that IBM systems typically had 10 times as much test code as application code, so we don't find the overhead of applying JAX to be excessive compared to that industrial benchmark. The goal is to test thoroughly, not to economize on test writing. Since tests will be run hundreds of times once written, and will help uncover errors in code beyond the class they are developed for, economizing at the test writing phase is false economy.

4 Informality Levels Two and Three: Automation of JAX

For this forum, space prohibits an extensive discussion of automating JAX. However we will present a summary of the two main approaches we are developing to leverage the error-finding capabilities of our methodology. We are developing tools to support two forms of automation, both based on an earlier testing project for C++ [9]. In full automation, a developer must write full specs (as in Appendix A) for a class, and also 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.

4.1 Full automation

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; we then compare the two action-sequence results for equality (either with a built-in "=" for a base type, or using the programmer-supplied *equals* for objects) since the axioms declares the left side to be equal to the right. For example, consider these 2 axioms from 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) ;
```

This formally encodes the semantics that if you try to remove something from an empty queue it is like a no-op. 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.

   The test 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 ivoke *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 the tools. The operation *add* needs arguments that are queues and elements in queues. Our tools must allow the specification of test points to fill these roles; these test points will get translated into the test scaffolding created in the *setUp()* and *tearDown()* methods of a JUnit *TestCase* class. Thus there will be a known collection of variable names that can be systematically substituted into the calls to *add* so that the RHS/LHS structures can be computed and compared. Such specs would look like ML data added to the functions defining the axioms:

```
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 would tell the tool 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. JAX tools give the possibility of large numbers of tests being done without having to write manually all the combinations into the test methods. JUnit then makes these large numbers of tests permanent in a regression suite.

4.2 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. 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. The automation tool 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 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.

The full automation level requires a tool that can parse formal specs for the ADTs and will generate JUnit test classes as output. The partial automation level we are achieving with nothing more that extensions to the JUnit *TestCase* class.

5   Discussion and Conclusions

We have presented 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 formal*isms;* effective use of JAX requires only Java. The method is automatable in at least two ways for designers who don't mind writing formal specifications.

Our preliminary studies show that JAX-based test classes find more errors in the target class than the basic method we compared it with (writing one test class method for each method in the target class). These studies were initial investigations only and were not blind controlled experiments. Though the results were encouraging, more thorough and controlled experiments need to be performed for the findings to be considered conclusive. We are pursuing such studies with large Java applications (30K – 50K lines) obtained from the EPA which were developed with JUnit testing. Our follow-on studies involve writing JAX-based test classes to compare to the JUnit classes supplied with the EPA applications. One thing to note is that the JUnit classes provided with these applications are not strictly written according to the simple one-test-method-per-target-method approach; rather, they are supplemented in an *ad hoc* fashion with test methods deemed needed by the programmers but not corresponding directly to any target method. We think comparing JAX tests to these JUnit tests will be a more effective study of the thoroughness of the JAX procedure.

We are not suggesting that all classes be tested this way. However, we think that certain forms of classes lend themselves well to the approach. Classes that are highly algorithmic, for example, or have

little GUI interactivity are especially applicable. We do not wish to rob agile methods of their flexibility and responsiveness; however, thorough testing is needed for agile development as much as for traditional processes. JAX is a *mostly mechanical* procedure for producing thorough collections of test methods.

Another issue we have not studied is the impact JAX would have on refactoring. More test methods means more code to move or alter as one refactors target classes. It is not clear how difficult it is to refactor JAX tests compared to other JUnit tests. Automation could help here, as axioms/specs might be easier to move than full tests; the tests could be regenerated once the specs were relocated in the refactored code.

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] Proceedings of the Software Design and Management Conference 2001: *Software Pioneers*, Bonn, Germany, June 2001; audio/video streams of the talks at can be viewed at
http://www.sdm.de/conf2001/index_e.htm

Appendix A

The full set of axioms for the bounded set BST is given here, 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.

```
(*
   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 ;
```