# Dynamic Analysis of Algebraic Structure to Optimize Test Generation and Test Case Selection

Anthony J H Simons and Wenwen Zhao
*Department of Computer Science, University of Sheffield*
*a.simons@dcs.shef.ac.uk, zhaoww18@hotmail.com*

## Abstract

*Where no independent specification is available, object-oriented unit testing is limited to exercising all interleaved method paths, seeking unexpected failures. A recent trend in unit testing, that interleaves dynamic analysis between each test cycle, has brought useful reductions in test-set sizes by pruning redundant prefix paths. This paper describes a dynamic approach to analyzing the algebraic structure of test objects, such that prefix paths ending in observer or transformer operations yielding unchanged, or derived states may be detected and pruned on-the-fly during testing. The fewer retained test cases are so close to the ideal algebraic specification cases that a tester can afford to confirm or reject these cases interactively, which are then used as a test oracle to predict many further test outcomes during automated testing. The algebra-inspired algorithms are incorporated in the latest version of the JWalk lazy systematic unit testing tool suite, which discovers key test cases, while pruning many thousands of redundant test cases.*

## 1. Overview

Systematic software unit testing methods fall into two categories. Code-based testing methods seek to exercise all paths through the software, identifying unexpected unit failures. Specification-based testing methods seek to validate the software unit completely against a formal specification, which serves as a test oracle. Recently, these approaches have started to converge, particularly in the *lazy systematic unit testing* method [1, 2], which combines semi-automatic inference of the test unit's specification with systematic conformance testing from the specification. The power of this method depends critically on an automated dynamic analysis to identify the most important test cases, whose outcomes must be confirmed by the tester. These key test cases then constitute the test oracle, used as a benchmark in fully automated testing.

This paper reports on a series of improvements to the dynamic analysis algorithms used by the *JWalk* lazy systematic unit testing tool suite [3]. These algorithms are deployed between each test cycle, using feedback from the previous test cycle to inform the test engine about which paths to extend in the following cycle. Starting from a baseline in which no test paths are pruned, rules of increasing sophistication are deployed to eliminate redundant test sequences. These strategies include eliminating all prefix paths that:

- terminate in exceptions;
- terminate in observations;
- terminate in re-entrant states;

and require a fine-grained ability to judge the algebraic properties of methods on a call-by-call basis, rather than simply partition all methods into *constructor*, *transformer* or *observer* categories. They also depend on the ability to judge object state equivalence in a flexible way, especially where their defining classes do not provide any consistent measure of equality.

In the rest of this paper, section 2 describes the increasing use of dynamic analysis during testing, to profit from feedback about the testing process. Section 3 describes the *JWalk* tool suite [3], highlighting the use of feedback-based code exploration to learn the algebraic specification of a test class, with hints from the programmer. Section 4 describes the algorithms deployed to detect the algebraic structure of unseen test classes in more detail. Section 5 demonstrates the effectiveness of algebra-motivated pruning rules for test-set reduction, comparing three different pruning rules. Section 6 considers how the retained test cases may constitute the "ideal" test set, to be confirmed by the tester, and reused as an oracle to predict many thousands of test outcomes in fully automated testing. The paper concludes with some observations on the properties of the algebraic analysis technique.

## 2. Dynamic analysis in unit testing

In systematic object-oriented unit testing, the focus is on exercising all interleaved method combinations. The testing assumption is that failures result mainly from unexpected states, caused by invoking methods in orders that ignore the expected protocols for the class in question. Since this is a laborious task to perform manually, automated approaches have been preferred. One of the earliest tools that generated all interleaved method paths was *JCrasher* [4]. This benefited from the Java programming language's facility for meta-analysis via the *reflection* API, a mechanism whereby compiled classes may be interrogated at run time to discover their public method interface. This was used to generate a breadth-first exploration of the test class's method invocation tree, using random techniques to populate each method-call with actual argument values. The focus of *JCrasher* was on forcing the test class to raise exceptions, expecting to identify code faults. However, the failures discovered were as much due to violated method preconditions, as they were to faulty or non-robust code. Later tools *DSD-Crasher* [5] and *Jov* [6] tended to confirm this finding. By contrast, *JWalk* does not assume that exceptions are faults; the tester has the chance to accept or reject such outcomes [1].

Other approaches concentrated on reducing the size of the breadth-first test-set. The *Rostra* tool [7] filtered the brute-force "whole method sequences" to yield "modifying method sequences", more selective paths consisting solely of state-modifying methods. These could be identified approximately from type signatures (typically, *observer* methods returned a result and state *modifying* methods returned *void*). In principle, this yielded smaller test sets that covered the state space of the test object, by eliminating sequences with *observers* in their prefix. To be more accurate in judging the equivalence of object states, *Rostra* required the user to supply explicit state-equality testing predicates. By contrast, *JWalk* does not require intrusive predicates or any kind of code instrumentation [1].

Another approach merged test paths by identifying common concrete states. The *Java Pathfinder* tool [8, 9] operated at a lower level, performing a *partial order reduction* analysis on sets of execution traces obtained directly from the Java bytecode interpreter. The testing strategy was to generate all interleaved method sequences, then identify equivalence-classes into which test sequences fell, so that the tester (or testing tool) could preserve single exemplars from each equivalence class for future testing. This generate-and-filter approach was expensive. By contrast, the first tools to deploy dynamic analysis and test-path pruning *during* the actual test-generation process were *JWalk* [1] and *Randoop* [10], which interleaved test generation and execution cycles. The advantage of this was that redundant prefix paths could be detected earlier, and pruned from the active test set before these were extended in the next cycle. Prefix sequences ending in an exception were pruned, based on the intuition that any path extending the prefix would always fail at exactly the same point, so not execute to completion. For example, the following pair of test sequences for a bounded *Stack* always fail at the same call to *pop(),* raising an *EmptyStackException*, making the longer test sequence redundant:

*new().pop()*
*new().pop().push(Object#1)*

*JWalk* also pruned prefix paths ending in an *observer*-method, on the basis that this would not modify the state of the test object. For example, *JWalk* treated the following pair of sequences as equivalent, by determining empirically that neither *size()* nor *isEmpty()* modified the *Stack* object in question:

*new().push(Object#1)*
*new().size().isEmpty().push(Object#1)*

*JWalk* used Java's reflection API to compare the shallow states of the test-object before and after each method execution, to detect side-effects. This was a more accurate way of determining *observer*-methods than a static analysis of signatures and worked whether or not the test class defined an *equals()* method.

The ability to map longer test sequences onto shorter sequences was used in *JWalk* to predict test outcomes dynamically for the longer sequences from known outcomes for the shorter sequences [1]. This was the first time that *test prediction* had been deployed during testing. At the time, it was foreseen that a more thorough *algebraic* classification of all methods (see below) might yield an even greater test set reduction and much greater predictive power. For example, if it could be determined that *pop()* were a *transformer*-method, undoing the effect of an earlier *push(),* returning the *Stack* object to a prior visited state, then the following sequences could be predicted to yield identical results:

*new().push(Object#1).size()*
*new().push(Object#1).push(Object#2).pop().size()*

Overall, if prefixes containing both *observer* and *transformer* methods could be mapped onto shorter prefixes, many more cyclic paths could be pruned during test generation; and outcomes for the longer sequences could also be predicted with certainty.

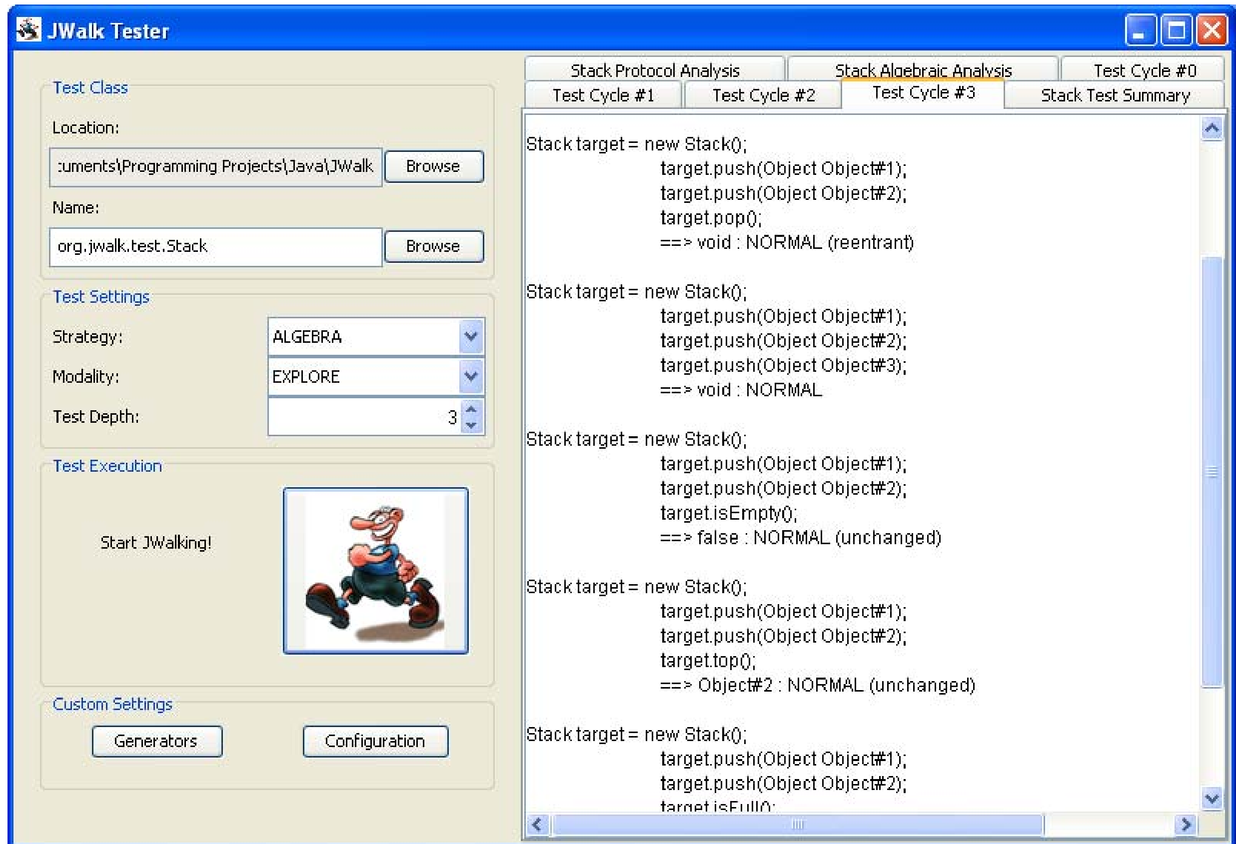


**Figure 1: JWalkTester performing an algebraic exploration of a Stack class to depth 3**

An *algebraic data type* is a structure consisting of operation signatures, typed in basic *sorts* (sets), whose semantics are defined using *axioms* (equations). The axioms are constructed after identifying all operations as belonging to one of the categories: *constructor, transformer* or *observer*. *Constructors* are primitive, returning all unique instances of the given data type. *Transformers* and *observers* are derived, defined by the axioms in terms of the constructors. Below, the term *primitive* is used instead of *constructor*, since the latter has a restricted sense in object-oriented programming: the *primitives* of a *Stack* include the *push* method as well as the *new Stack* constructor.

Our approach was partly inspired by the work of Henkel and Diwan, who induced the algebraic structure of Java classes semi-automatically by probing the behavior of test instances [11]. They derived an abstract data type signature from a concrete Java class through reflection, then generated and evaluated many ground terms, which were grouped into equivalence classes. Thereafter, an important generalization step induced quantified axioms, which succinctly captured many ground term equations. They also embedded this approach in a tool to help programmers write and debug algebraic specifications [12]. Our interest was mainly in the technique used to determine when objects had re-entered previously visited concrete states. This involved converting objects into their *serialized* format (a binary encoding used to transfer objects to persistent storage or across distributed systems) and then hashing to yield a single code representing the object's state. We found this approach unsuitable, for two reasons. Firstly, not every Java class declares that it supports serialization; and secondly, serialization offers no control over the depth to which object states are compared. Our alternative solution is presented in section 4, below.

## 3. The *JWalk* family of testing tools

The current work relates to the latest version of the *JWalk* tool suite, which comprises a number of tools, including *JWalkTester*, a GUI-based testing tool in the spirit of *JUnit* [13], *JWalkUtility*, a command-line version that prints all results to standard output, and *JWalkEditor* [2], an integrated Java editor, compiler

and testing tool, with Java-sensitive syntax highlighting in the style of *jEdit* [14]. All of these incorporate the common *JWalker* test engine, which is also offered as a component toolkit API for integration with other editors or testing tools. For this paper, *JWalkTester* was used to generate all the examples and statistics below (see figure 1 for an example).

All of the *JWalk* tools were conceived with a vision to support agile software development methods, such as XP [15], in which test-driven development is the cornerstone. The goal was to bring together the rigor of formal specification-based testing methods and the flexibility of constant code refactoring. Earlier work from our research group had highlighted how even simple finite-state specifications could greatly improve the selection of tests written for XP [16]; and also how re-using saved tests in regression testing was not as secure as previously assumed [17, 18]. Nonetheless, XP and similar methods remained wary of lightweight specifications, requiring a different approach.

The *lazy systematic unit testing* method was devised, based on the two notions of *lazy specification*, the ability to infer the evolving specification of a unit on-the-fly by dynamic analysis, and *systematic testing*, the ability to explore and test the unit's state space exhaustively to bounded depths [1]. *Lazy specification* refers to a delayed approach to software specification, in which the specification evolves rapidly in parallel with frequently modified code [2]. The specification is inferred by a semi-automatic analysis of a prototype software unit, with some user-interaction. *Systematic testing* refers to a complete, conformance testing approach, in which the tested unit is shown to conform exhaustively to a specification, up to the testing assumptions, so providing guarantees of correctness once testing is over [18].

The featured *JWalkTester* tool supports three test strategies, which are protocol-, algebra-, and state-based. In the *protocol* strategy, all interleaved methods are executed on test instances in a breadth-first manner. In the *algebraic* strategy, all algebraic constructions are explored, driving test instances into all their distinct concrete states. In the *state-based* strategy, the high-level (or abstract) states of the test class are discovered by exploration, and test instances are driven through all their high-level states and transitions. Dynamic analysis is critical in detecting actual state changes empirically, rather than relying on a static analysis of variable assignments, or method signatures, since some updates are conditional on particular argument values. The algebraic exploration technique uses only *primitive* algebraic constructions to extend test sequences. This also reduces the search space when seeking high-level states, found by evaluating the reached concrete states using the natural state predicates of the test class. In this way, the dynamic analysis techniques reported here optimize both low- and high-level state exploration.

The *JWalkTester* tool may be executed in three modalities, to inspect, explore and validate the test class. In the *inspect*-modality, it extracts the public constructor and method interface of the test class, including public methods inherited from superclasses. It may also probe the test class by dynamic analysis, to discover its algebraic structure (a new feature, from *JWalk v1.0*), or its high-level state-space [1]. In the *explore*-modality, the tool constructs and executes test sequences according to the chosen test strategy and displays the results, sorted by test path length, in a tabbed output pane for the tester to examine. Figure 1 illustrates exploring *all algebraic constructions* of a *Stack* class, to depth 3. In the *validate*-modality, the tool also interacts in a limited way with the tester, who must confirm or reject certain key test outcomes, which are compiled in an oracle and used to predict further test outcomes. Eventually, over 90% of testing is fully automated using saved, or predicted outcomes [1, 2].

Dynamic analysis has a role to play in determining when a particular test outcome should be identified as significant and presented to the tester for confirmation; and also when that same test result could be used to predict further test outcomes. The whole benefit of *lazy systematic unit testing* is to minimize the user interaction required to create a complete test oracle. The goal of dynamic analysis is therefore to identify, in some sense, the "ideal" test cases for presentation to the tester. In the context of this paper, this is interpreted as all observations on the leaves of the tree of all novel primitive algebraic constructions.

## 4. Dynamic analysis of algebraic structure

Previously, the old version 0.8 of the *JWalk* toolset had a rudimentary ability to classify *observer* methods (see section 2) and so prune redundant paths whose prefix contained *observers*. The current work improves on this in two ways: by pruning redundant paths containing both *observers* and *transformers* in the prefix; and by applying the dynamic state analysis and test prediction rules *per method invocation*, which allows further predictions to be made when states are not modified by methods that might, at other times and for other arguments, update state.

The old algorithm compared shallow state vectors taken from the test object, before and after each method invocation, to identify and classify *observer*-methods. In the improved algorithm, we wanted to compare the concrete state after each method invocation with *every earlier state* in the same test

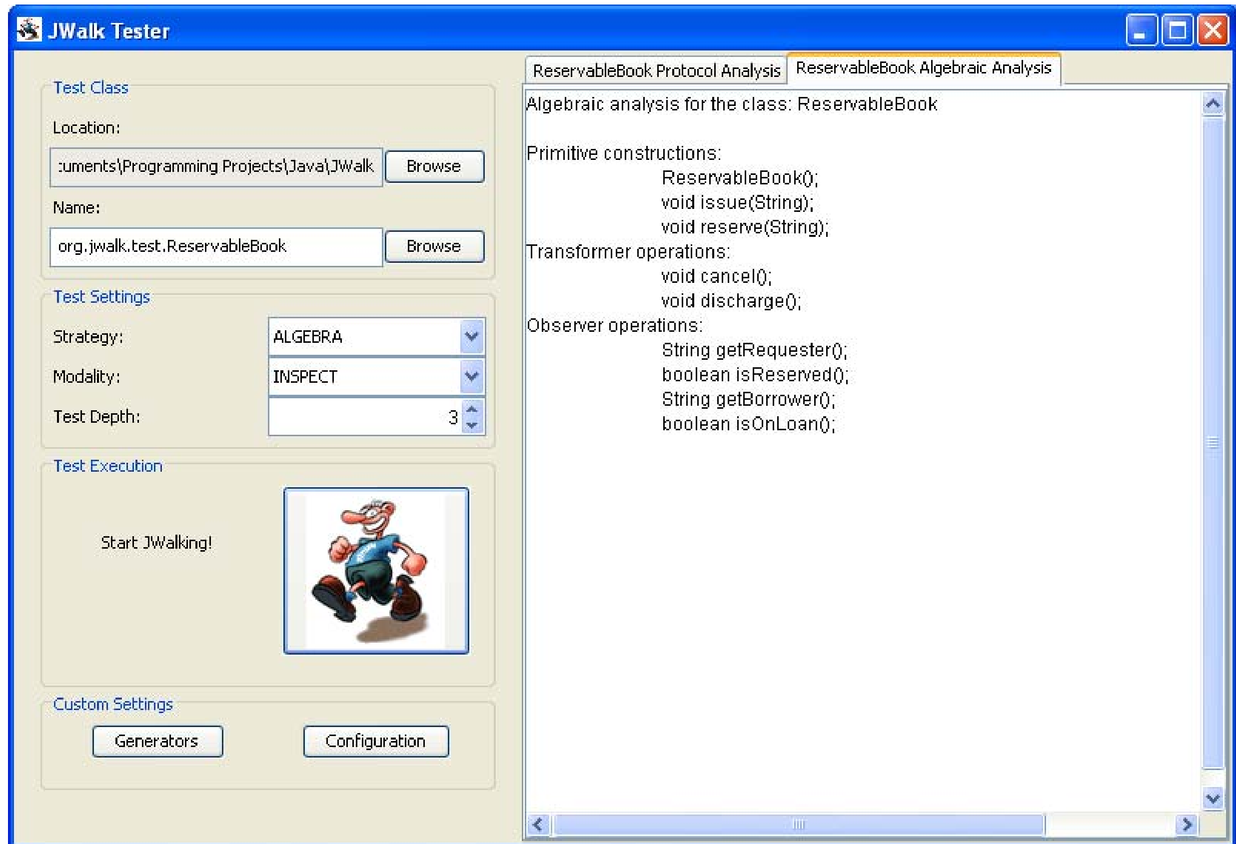


**Figure 2: JWalkTester analyzing the algebraic properties of a ReservableBook class**

sequence, to identify re-entrant methods that returned the test object to some prior visited state. For this, a more compact encoding of state was desirable.

When discussing the algebraic nature of object states, the semantic issue of equality arises. Comparing two objects might make use of an *equals()* method naturally provided by their class; but then, some classes might not define such a method (in Java, they would inherit *Object*'s method by default, which compares object references for identity). Supporting mixed notions of *reference*, *shallow* and *deep* equality might be considered inconsistent. Furthermore, the behavior of any user-defined *equals()* method might be faulty, or might conflict with the proper algebraic notion of equality [19], which is defined as all *observations* on the data type yielding (recursively) equivalent results. For this reason, we wanted to find a more consistent and repeatable means of determining state equality, which could nonetheless be controlled by the tester.

The approach we eventually adopted replaced the old strategy of extracting object state vectors, which might consume memory resources, by the computation of a single hash code to represent the whole state of the object. This is similar to Henkel and Diwan [11], except that the hash value is not based on Java's serialized format, which is not always available. An internal release 0.9 of the *JWalk* toolset [20] computed hash codes from the persistent *oracle value* string representation [1] of each object, which the *JWalk* tools compute for all types. However, the processing time and storage required to generate the oracle strings repeatedly were unacceptably high. Also, the benefits we were seeking did not actually require persistent states to be compared across different test runs. So it was feasible to compute hash codes directly from objects and values in memory.

Primitive types, such as *int*, and "boxed" Java types, such as *Integer*, and types with a natural hash code based on their value, such as *String*, use their natural *hashCode()* method. The hash code for any other kind of object is obtained by combining the hash codes for its fields, where the combined code reflects both the order and value of each field (using a prime multiplier for the position). Fields are extracted by reflection, bypassing the usual visibility restrictions. Where a

field is an object reference, a choice exists to apply the hashing algorithm recursively, or simply return a code based on the memory address. This is controlled by a depth parameter supplied by the tester, denoting the object tree-depth to which state comparisons should be conducted (specifying shallow, or deeper equality).

Given this compact encoding of object state, it was relatively easy to incorporate the extra information into the core test engine. This constructs *TestSequence* objects, consisting of many *TestCase* objects, each of which exercises a single constructor or method. The state of the target object is encoded immediately after executing each *TestCase*, and cached locally. Once a *TestSequence* has fully executed, it is possible to query the sequence to find if the final state was *unchanged*, or *re-entrant* (see figure 1, where these indicators are appended to certain test outcomes). This is a fast algorithm, which compares the final state code with the penultimate one; or with all earlier state codes in the sequence. The chances of accidental hash collisions are remote, especially since sequences are short (up to low tens of *TestCases*), and all injected test input values are already quasi-unique, thanks to the monotonic test input generation strategy [1, 3].

The latest version 1.0 of the *JWalk* toolset infers the algebraic structure of the test class by successive conservative approximations, probing the dynamic behavior of the class. Figure 2 shows *JWalkTester* discovering automatically the algebraic structure of a *ReservableBook*. The operations of the class are classified into the categories: {*primitive, transformer, observer*}. All object constructors are assumed to be *primitive*, unless it can be proven that they are derived, creating the identical object from fewer supplied initial parameters, in which case they are reclassified as *transformers*. All methods are initially assumed to be *observers*, until they are found to modify state, in which case they are first classified as *primitive*; but if later they are found to drive the target object into previously visited states, their category is revised to *transformer*.

## 5. The role of algebra in test pruning

Algebra-inspired analysis adds to the growing set of sophisticated measures that allow a testing tool, with interleaved test generation, execution and analysis phases, to prune redundant test sequences. The size of the baseline test set (constructors and all interleaved methods) may be calculated algorithmically from the size of the test class's public API. A class with $c$ constructors and $m$ methods has $cm^k$ test sequences of length $k$, therefore $\Sigma\, cm^k$ sequences altogether, for any bounded depth $0 \leq k \leq n$. For example, a bounded *Stack* with just one public constructor and the six methods *{push, pop, top, size, isEmpty, isFull}* has 1 + 6 + 36 + 216 +… sequences. For bounded depth $n = 3$, we would expect a maximum of 259 sequences.

The first pruning rule drops prefix paths ending in raised exceptions. The effect of this can be observed by running any *JWalk* tool in *protocol exploration* mode. Test cycle 0 creates the *Stack* instance. Out of the six sequences exercised in test cycle 1, two raise exceptions (*pop()* and *top()* called on an empty *Stack*). This causes 12 paths to be pruned in test cycle 2 (all 6 extensions of each failed path). The remaining 4 paths from test cycle 1 are extended to yield 24 paths in test cycle 2. Of these, another 6 paths terminate with exceptions (mostly consisting of an *observer*, followed by *pop()* or *top()*), causing 108 sequences to be pruned from test cycle 3. Cumulatively, 120 test sequences are pruned during 3 test cycles, leaving 139 test sequences that were actually constructed and executed. This is a useful saving, compared to the 259 sequences that could have been attempted (see table 1).

The second pruning rule drops prefix paths ending in algebraic *observers*, which do not modify the state of the target object. In this case, for the same *Stack*, 30 paths were pruned in test cycle 2 and a further 204 paths in test cycle 3. Cumulatively, 234 test sequences were pruned over 3 cycles, leaving only 25 tests that were actually constructed and executed. This compares even more favorably with the original 259 sequences!

When the third pruning rule is added to drop prefix paths ending in algebraic *transformers*, which re-visit earlier states, a similar picture is seen. As before, 30 paths are pruned in test cycle 2, but a further 210 paths are pruned in cycle 3 (an increase of 6). Cumulatively, 240 test sequences were pruned over 3 cycles, leaving only 19 tests that were actually constructed and executed. This is a tiny fraction of the 259 sequences that could have been attempted (see table 1). When looking at the retained test sequences, these are almost exclusively paths of the form:

*new().size()*
*new().push(Object#1).size()*
*new().push(Object#1).push(Object#2).size()*

that is, paths which force the test object through all algebraic constructions, then make all *observations* (or exercise all *transformers*) at the leaves of the algebraic tree. In terms of state exploration alone, this is close to the ideal test set that a programmer might have wished to create manually, since it tests all fine-grained properties of the class. Yet, since it was created by algorithm, we can be confident that it is complete, up to the chosen bounded depth.

**Table 1. Pruning applied to Stack**

| *Stack* | *base* | *exc* | *obs* | *alg* |
|---|---|---|---|---|
| *0* | 1 | 1 | 1 | 1 |
| *1* | 7 | 7 | 7 | 7 |
| *2* | 43 | 31 | 13 | 13 |
| *3* | 259 | 139 | 25 | 19 |
| *4* | 1555 | 667 | 43 | 25 |
| *5* | 9331 | 3391 | 79 | 31 |

**Table 2. Pruning applied to Wallet**

| *Wallet* | *base* | *exc* | *obs* | *alg* |
|---|---|---|---|---|
| *0* | 1 | 1 | 1 | 1 |
| *1* | 6 | 6 | 6 | 6 |
| *2* | 31 | 31 | 11 | 11 |
| *3* | 156 | 156 | 16 | 16 |
| *4* | 781 | 781 | 26 | 21 |
| *5* | 3906 | 3906 | 41 | 31 |

To evaluate the power of the new algebraic pruning rules under known conditions, a series of experiments were conducted using the standard set of test classes used to develop the *JWalk* toolset. These include a *Stack* (which exhibits obvious state-like behavior; and has a well-known abstract data type algebra); a *Wallet* (whose behavior depends more on the values supplied as method arguments); and a basic *LibraryBook* and its subclass, *ReservableBook* (both with re-entrant states and abstract algebras to discover; these were included to verify *JWalk's* ability to detect all novel interleaved method combinations to test, after extending a class by inheritance). Stress testing was also carried out by exercising the standard *Java* library classes *Character* and *String* (both known to have very large APIs).

Tables 1-4 indicate the cumulative sizes of the *retained* test sets, after pruning rules 1-3 are successively applied, for test cycles of increasing length. The numbers of *pruned* sequences may be obtained by subtracting the size of the featured test set from the size of the baseline set, in each case.

In the tables, the leftmost column gives the test cycle index number (where cycle 0 is construction; cycle 1 is method paths of length 1, etc). The column *base* indicates the size of the baseline test set (all interleaved methods). The column *exc* gives the test set size after pruning rule 1 is applied (drop *exception* prefixes). The column *obs* likewise gives the test set size after pruning rule 2 is also applied (drop *observer* prefixes). The final column *alg* gives the test set size after pruning rule 3 is also applied (drop *transformer* prefixes). This last column reflects the algebraic filtering strategy used in the current version 1.0 of the *JWalk* toolset.

The results indicate an impressive ability to rule out redundant test sequences by automatic analysis. For test classes whose methods have preconditions relating to state, such as *Stack*, a useful reduction is achieved by the first pruning rule. For test classes such as *Wallet*, whose method preconditions are rarely violated, no reduction may be seen until the *observer*-prefix pruning rule is applied. *Wallet*'s state is affected more by the choice of amounts credited and debited than anything else, so rarely re-visits earlier states. On the other hand, the *LibraryBook* cycles through two states in response to the *issue()* and *discharge()* methods. Here, not only does the complete algebraic filtering strategy (dropping *exceptional*, *observer* and *transformer* prefixes) prune more redundant cases, but the test set reaches a stable state of 9 ideal test cases, from test cycle 2 onward.

Similarly, the *ReservableBook* subclass (see figure 2 for its API) cycles through four states in response to the *issue()*, *discharge(), renew()* and *cancel()* methods. It eventually stabilizes on 41 ideal test cases after cycle 4. (Note that in cycle 5, *protocol exploration* exhausted the available memory; but the more aggressive pruning in *algebraic exploration* allowed testing to continue).

**Table 3. Pruning applied to LibraryBook**

| *LibBook* | *base* | *exc* | *obs* | *alg* |
|---|---|---|---|---|
| *0* | 1 | 1 | 1 | 1 |
| *1* | 5 | 5 | 5 | 5 |
| *2* | 21 | 21 | 9 | 9 |
| *3* | 85 | 81 | 13 | 9 |
| *4* | 341 | 301 | 17 | 9 |
| *5* | 1365 | 1101 | 21 | 9 |

**Table 4. Pruning applied to ReservableBook**

| *ResBook* | *base* | *exc* | *obs* | *alg* |
|---|---|---|---|---|
| *0* | 1 | 1 | 1 | 1 |
| *1* | 9 | 9 | 9 | 9 |
| *2* | 73 | 73 | 25 | 25 |
| *3* | 585 | 561 | 49 | 33 |
| *4* | 4681 | 4185 | 97 | 41 |
| *5* | 37449 | **mx** | 169 | 41 |

The effectiveness of algebraic analysis can be seen in the huge reductions in test-set sizes, focusing on the most important test cases. *Stack* retained the best 0.33% paths (pruning 9,300 redundant paths), *Wallet* retained the best 0.79% paths (pruning 3,875 paths), *LibraryBook* retained the best 0.66% paths (pruning

1,356 paths) and finally *ReservableBook* retained the best 0.12% paths (pruning 37,408 paths). When stress testing using *java.lang.Character* (1 constructor and 78 methods), *protocol exploration* exhausted memory in cycle 3 (a baseline of 480,715 paths), but *algebraic exploration* stabilized on 79 test cases from cycle 1. Similarly, for *java.lang.String* (13 constructors and 64 methods) *protocol exploration* exhausted memory in cycle 3 (a baseline of 54,093 paths), but *algebraic exploration* stabilized on 845 test cases from cycle 1 (using a custom index generator [3] to avoid out-of-bounds *char* array access during *String* construction).

## 6. The role of algebra in test prediction

The utility of algebraic analysis can be measured in another way, while interacting with the *JWalk* tools in the *algebraic validation* mode, when inferring the test oracle for a given class. The first benefit of the new algorithms is that fewer test cases are presented to the tester for manual confirmation or rejection than before. This is partly due to the extra pruning rule (see section 5) and partly due to the improved capability for *test prediction*, which is still not quite complete, an interesting finding that we explain below.

**Table 5. Algebraic confirmations per cycle**

| *Class : cycle* | *0* | *1* | *2* | *3* | *4* | *5* |
|---|---|---|---|---|---|---|
| Stack 0.8 | 1 | 5 | 4 | 9 | 12 | 26 |
| Stack 1.0 | 1 | 5 | 4 | 4 | 4 | 4 |
| Wallet 0.8 | 1 | 4 | 4 | 4 | 8 | 12 |
| Wallet 1.0 | 1 | 4 | 4 | 4 | 4 | 8 |
| LibBook 0.8 | 1 | 2 | 3 | 2 | 3 | 2 |
| LibBook 1.0 | 1 | 2 | 3 | - | - | - |
| ResBook 0.8 | 1 | 2 | 8 | 12 | 30 | 40 |
| ResBook 1.0 | 1 | 2 | 8 | 6 | 6 | - |

The power of the *JWalk* test engine comes from its predictive rules, especially predictions about sequence equivalence-classes. Whereas version 0.8 only mapped sequences containing *observers* in the prefix onto shorter sequences, version 1.0 also maps sequences, whose prefix contains *transformers*, onto equivalent shorter sequences. Table 5 shows the improvement gained, which starts to become noticeable from test cycle 3, where sequences with prefixes ending in *transformers* start to be pruned. Note that the tally of confirmations for *ReservableBook* is reduced even further, because this class inherits a partial oracle from its parent *LibraryBook* and only asks the tester to confirm novel interleaved combinations of methods.

Summing the counts of confirmations across a row in table 5 gives the cumulative confirmations to depth 5. By test cycle 5, the tool is presenting significantly fewer cases to the tester for validation than before. For example, in the current *algebraic validation* mode, the tester confirmed 22 cases over 5 cycles for the *Stack*, whereas in the old version of the same mode, the tester had to confirm 57 cases (35 more confirmations).

**Table 6. Residual protocol confirmations**

| *Class : cycle* | *0* | *1* | *2* | *3* | *4* | *5* |
|---|---|---|---|---|---|---|
| Stack 0.8 | - | - | - | - | - | - |
| Stack 1.0 | - | - | - | - | 4 | 8 |
| Wallet 0.8 | - | - | 1 | 12 | 99 | 691 |
| Wallet 1.0 | - | - | 1 | 12 | 100 | 704 |
| LibBook 0.8 | - | - | - | - | - | - |
| LibBook 1.0 | - | - | - | - | 3 | - |
| ResBook 0.8 | - | - | - | - | - | **mx** |
| ResBook 1.0 | - | - | - | - | 20 | **mx** |

Note also that the total number of confirmations is strictly less than the number of paths actually explored. For example, whereas the tester confirmed 22 cases (over 5 cycles) for the *Stack*, the tool explored 31 cases altogether (see table 1), making 9 further automatic validations, based on predictions about test outcomes. On examining these cases, they are all found to be *opportunistic* predictions that methods with a *void*-result type are expected to yield no result (see the discussion in section 7). All the other cases presented to the tester were of observations made on the leaves of the tree of all algebraic constructions. We may conclude with some confidence that the *algebraic validation* mode is the most economical way to present the tester with significant test cases that define (most of) the behavior of the test class.

This is borne out in table 6, where the majority of the remaining test outcomes in the brute-force *protocol validation* mode are predictable from the oracles that were trained in *algebra validation* mode. Recall from tables 1-4 how *protocol exploration* had many thousands of test cases. To be able to validate most of these automatically, having confirmed only a few tens of cases by hand, is an outstanding success and the hallmark of the *lazy systematic unit testing* method.

However, the benefits gained by pruning the extra *transformer*-prefixes in the newer version of the *JWalk* toolset are not completely carried forward when using the smaller trained oracles to predict all test outcomes in *protocol validation* mode. Some of the extra cases that were pruned by the more aggressive algebraic rule had to be confirmed in *protocol* mode, indicating that the revised prediction rule, while more powerful than

the old version, is not quite as effective as hoped. For example, in the current *protocol validation* mode, the tester had to confirm 12 residual cases over 5 cycles for the *Stack*, compared to no residual cases in the old version of the same testing mode.

Nonetheless, this is still a net gain, since in the old version, the tester had previously confirmed 35 extra cases (see table 5), so in fact the new version still saved the tester 35 – 12 = 23 confirmations for this example. Likewise, for the *LibraryBook* example, the new version saved 7 – 3 = 4 cases and the *ReservableBook* saved 70 – 20 = 50 cases. The *Wallet* example proved somewhat resistant to re-entrant state prediction in both tool versions. The states of this object were influenced more by argument values (which were quasi-unique on each call) and states tended not to be re-entered.

How may we explain the mixed performance of test prediction rules associated with re-entrant states? The reason for this has to do with the exact concrete states of the objects that are being compared. For example, the tools can detect that the following two sequences leave the target *Stack* object in the same state:

*new()*
*new().push(Object#1).pop()*

and from this fact, they can predict some test outcomes for the longer sequence, based on extending such pairs of sequences with simple observations:

*new().size()*
*new().push(Object#1).pop().size()*

which map onto the same outcome. However, the tools cannot always make use of re-entrant state information in other kinds of extension, which do not leave the target object in the same concrete state:

*new().push(Object#1)*
*new().push(Object#1).pop().push(Object#2)*

That is, the two *Stacks* produced by these sequences are not equal, because their *top* elements are different, as a consequence of the value generator synthesizing a new, distinct instance of *Object* as the argument to the second *push()* in the longer sequence.

In hindsight, this kind of situation will occur frequently in *transformer*-related cycles, because the *JWalk* toolset always synthesizes quasi-unique input values for each method argument, so that these values may be distinguished when later observed (note that it is significant, in figure 1, that the result of the *top()* method is *Object#2*, rather than *Object#1*). The only way to improve the tools' reasoning ability further would be to add some kind of symbolic generalization over concrete argument values, perhaps in the spirit of axiom induction [11, 12].

## 7. Conclusions

Algebra-inspired analysis has been shown to have a hugely beneficial influence on the reduced generation of test sequences and the selection of "ideal" test cases. Determining the space of *all algebraic constructions* for a class-under-test is the best way to focus a test generator on the most relevant test cases that will exercise all the distinct, fine-grained behaviors of the class. In the above examples, the test sets were eventually reduced to a few tens of key test cases, which could be confirmed manually and reused in predictive testing. This was compared against the many thousands of test cases that might be generated in any baseline, or *protocol exploration* strategy.

Perhaps the single most important aspect is having a tool, such as *JWalkTester*, to guide the tester through all the relevant cases. Our previous experiments have shown that trying to achieve similar complete test coverage by manual test case selection will yield much poorer results [2, 21]. The *JWalk* tools take control of the selection of test cases, here guided by algebraic analysis, relieving the tester of this burden. Another key feature of the *JWalk* tool suite is the ability to interleave cycles of test generation, execution and analysis. That is, the generation of each new test cycle is informed by an analysis of the results from the previous cycle. This allows the tool to select which paths to expand in the next cycle, rather than over-generating all test paths and filtering these afterwards [4-9]; and is also quite different from prioritizing and re-ordering test suites [22, 23]. *JWalk* avoids generating any redundant paths in the first place, yet seeks to identify all the distinct test cases (not a subset) in the test class's algebra, or high-level state space.

Regarding the influence of algebraic analysis techniques on test outcome prediction, the results are still very good, but sometimes mixed. Many thousands of test outcomes were predicted for the *protocol validation* mode, after training up an oracle in the *algebra validation* mode. It is clear that mapping sequences with *observers* in the prefix onto shorter sequences (with known outcomes) has a strong predictive power. However, mapping sequences with *transformers* in the prefix onto shorter sequences without cycles may not yield quite as many predicted gains. This was found to be due to an interaction between the predicted cyclic behavior and the injection of different input values, resulting in objects whose states were not in fact equivalent.

One aspect of the test prediction strategy worth highlighting is that the decision to map onto a known shorter sequence is made on a case-by-case basis, and is not based on any global classification of all methods into *primitive, transformer* or *observer* operations. This is because a given method may, on subsequent invocations, choose to modify the state of the target object, or leave it unchanged. The *Wallet* test case demonstrated this behavior, in that sometimes the debiting-action was blocked (if the requested amount exceeded the balance) and sometimes it succeeded. However, whenever the state context for a predictive rule applies, the rule may always be applied safely, because states are always compared empirically.

For this reason, we call this kind of prediction a *strong*, or *conservative* assumption, which is always guaranteed to hold, unlike the *weak*, or *opportunistic* assumption made when expecting a *void*-method to return no result. The latter prediction is potentially vulnerable to false positives (viz. a *void*-method, which should raise an exception, but does not, and is passed by default). In practice, such fault cases are identified in the following test cycle [1, 2], when the tester observes extra paths that should have failed.

**Acknowledgment:** Thanks are due to Arne-Michael Toersel (at TaicPart '07) for the *Wallet* test case; and to Mihai-Gabriel Glont, for inspiring the GUI design of the *JWalkTester* tool.

## 8. References

[1] A. J. H. Simons, "JWalk: a tool for lazy systematic testing of Java classes, by design introspection and user interaction", *J. Auto. Softw. Eng.*, *14 (4)*, 2007, pp. 369-418.

[2] A. J. H. Simons, N. Griffiths and C. D. Thomson, "Feedback based specification, coding and testing", *Proc. 3rd Test. in Acad. and Indust. Conf.: Pract. and Research Tech.*, eds. M. Roper, G. M. Kapfhammer and L. Bottaci, IEEE, Windsor, UK, 2008, pp. 69-73.

[3] A. J. H. Simons, "JWalk: lazy systematic unit testing", *http://www.dcs.shef.ac.uk/~ajhs/jwalk/*, 2009.

[4] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness checker for Java", *Software: Practice and Experience, 34(11),* 2004, pp. 1025-1050.

[5] C. Csallner and Y. Smaragdakis, "DSD-Crasher: a hybrid analysis tool for bug finding", *Proc. 5th ACM Sigsoft Int. Symp. on Softw. Testing and Analysis,* ACM, Portland, Maine, 2006, pp. 245-254.

[6] T. Xie and D. Notkin, "Tool-assisted unit test selection based on operational violations", *Proc. 18th IEEE Int. Conf. Automated Softw. Eng.,* IEEE, Montreal Canada, 2003, pp. 40-48.

[7] T. Xie, D. Marinov and D. Notkin, "Rostra: a framework for detecting redundant object-oriented unit tests", *Proc. 19th IEEE Conf. Automated Softw. Eng.*, IEEE, Washington DC, 2004, pp. 196-205.

[8] W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda, "Model checking programs", *Automated Softw. Eng. J., 10(2)*, 2003, pp. 203-232.

[9] W.Visser, C. S. Pasareaunu and R. Pelánek, "Test input generation for Java containers using state matching", *Proc. 5th ACM Sigsoft Int. Symp. Softw. Testing and Analysis*, ACM, Portland, Maine, 2006, pp. 37-48.

[10] C. Pacheco, S. K. Lahiri, M. D. Ernst and T. Ball, "Feedback-directed random test generation", *Proc. 29th Int. Conf. Softw. Eng.*, IEEE Computer Soc., Minneapolis, USA, 2007, pp. 75-84.

[11] J. Henkel and A. Diwan, "Discovering algebraic specifications from Java classes", *Proc. 17th Europ. Conf. Obj.-Oriented Progr., LNCS 2743*, Springer, Darmstadt, Germany, 2003, pp. 431-456.

[12] J. Henkel and A. Diwan, "A tool for writing and debugging algebraic specifications", *Proc. 26th Int. Conf. Softw. Eng.*, IEEE Computer Soc., 2004, pp. 449-458.

[13] K. Beck, *The JUnit Pocket Guide*, O'Reilly, Beijing, 2004.

[14] S. Pestov et al., "jEdit programmer's text editor", http://www.jedit.org/, 2009.

[15] K. Beck, *Extreme Programming Explained: Embrace Change, 2nd edn*. New York: Addison-Wesley, 2005.

[16] W. M. L. Holcombe, "Where do unit tests come from?", *Proc. 4th Int. Conf. on Extreme Progr. and Flexible Proc. in Softw. Eng., LNCS 2675*, Springer, Genova, Italy, 2003, pp. 161-169.

[17] A. J. H. Simons, "Testing with guarantees and the failure of regression testing in eXtreme Programming", *Proc. 6th Int. Conf. on Extreme Progr. and Flexible Proc. in Soft. Eng., LNCS 3556*, Springer, Sheffield, UK, 2005, pp. 118-126.

[18] A. J. H. Simons, "A theory of regression testing for behaviourally compatible object types", *Softw. Test., Verif. Reliab., 8(2)*, 2006, pp. 133-156.

[19] H. Y. Chen, T. H. Tse and T. Y. Chen, "TACCLE: a methodology for object-oriented software testing at the class and cluster levels", *ACM Trans. Softw. Eng. Meth., 10(1),* 2001, pp. 56-109.

[20] W. Zhao, "Test prediction in revisited states of JWalk", *MSc dissertation, Department of Electronic and Electrical Engineering,* University of Sheffield, 2008.

[21] A. J. H. Simons and C. D. Thomson, "Benchmarking effectiveness for object-oriented unit testing", *Proc. 1st Software Testing Benchmark Workshop, IEEE 1st Int. Conf. Softw. Testing*, Lillehammer, 2008.

[22] A. Smith, J. Geiger, G. M. Kapfhammer and M. L. Soffa, "Test suite reduction and prioritization with call trees", *Autom. Softw. Test.,* 2007, pp. 539-540.

[23] Z. Li, M. Harman and R. M. Hierons, "Search algorithms for regression test case prioritisation", *IEEE Trans. Softw. Eng., 33,* 2007, pp. 225-237.