Universities of Leeds, Sheffield and York
http://eprints.whiterose.ac.uk/

JWalk: a Tool for Lazy, Systematic Testing of Java Classes by Design Introspection and User Interaction©

ANTHONY J. H. SIMONS   a.simons@dcs.shef.ac.uk

Department of Computer Science, University of Sheffield, United Kingdom

Abstract. Popular software testing tools, such as *JUnit*, allow frequent retesting of modified code; yet the manually created test scripts are often seriously incomplete. A unit-testing tool called *JWalk* has therefore been developed to address the need for systematic unit testing within the context of agile methods. The tool operates directly on the compiled code for *Java* classes and uses a new lazy method for inducing the changing design of a class on the fly. This is achieved partly through introspection, using *Java*'s reflection capability, and partly through interaction with the user, constructing and saving test oracles on the fly. Predictive rules reduce the number of oracle values that must be confirmed by the tester. Without human intervention, *JWalk* performs bounded exhaustive exploration of the class's method protocols and may be directed to explore the space of algebraic constructions, or the intended design state-space of the tested class. With some human interaction, *JWalk* performs up to the equivalent of fully automated state-based testing, from a specification that was acquired incrementally.

Keywords: agile methods, unit testing, state-based testing, algebraic testing, lazy specification, lazy systematic testing, operational abstraction, *JWalk*, *JUnit*

1.   Introduction

One of the strengths of agile software development methods, such as *eXtreme Programming* (XP), is the renewed emphasis on rigorous testing (Beck, 2000; Beck 2005). XP advocates test-driven development, in which the intended specification of the software is expressed as tests, which the software must pass. The tests are created before any production code is written; and all the tests must be passed before each coding cycle is considered complete. The *JUnit* tool (Beck, 2004; JUnit, 2007) automates the re-testing process; and passing the saved tests is treated as a kind of conformance testing.

While encouraging testing is commendable, a potential weakness is that the test cases are only selected manually, according to the programmer's intuition, and are therefore usually incomplete. Tests written by hand have a tendency to confirm expected behaviour, or diagnose anticipated fault classes. They cannot be relied upon to validate the complete functional behaviour of the software, since they do not usually anticipate all the possible ways in which the software's operations might

---

interact with each other (Holcombe, 2003). Achieving test-completeness is made more difficult in object-oriented languages by the mechanism of inheritance, which militates against reusing saved test suites in conformance testing. *JUnit*'s saved tests fail even to cover the original state-space of the parent class in the child class, because of the state partitioning in the refinement (Simons, 2006).

1.1   Agile Testing and Completeness

The motivation for the current work was to find a more rigorous class unit-testing method suitable for the agile software development community. The testing method has to satisfy several goals: firstly, testing should be demonstrably complete, under some weakening assumptions about the regularity of the intended design; secondly, the testing process should be automated, in as far as this is possible; thirdly, any saved test sets should evolve smoothly with the constantly-changing design of the target class; and fourthly, the testing method should be accessible to a community that eschews any kind of formal design process.

This is quite a difficult combination of goals to achieve together. For example, most software testing methods that aspire to *functional test completeness* are based on automated test generation from formal specifications. These include state-based approaches, particularly the X-Machine testing method (Ipate and Holcombe, 1997; Holcombe and Ipate, 1998), and algebraic approaches, particularly the TACCLE algebraic testing method (Chen et al., 1998; 2001). What makes these formal approaches superior to their peers is the focus on completeness for conformance testing purposes. X-Machine test generation is based on Chow's W-method (Chow, 1978), which guarantees deterministic levels of confidence even in the presence of duplicated states and transitions, something which weaker transition- or switch-coverage methods fail to obtain; see also (Jard and von Bochmann, 1983; McGregor, 1994; Binder, 2006) for contrasting approaches. Likewise, the systematic test generation algorithm in TACCLE creates the minimal test-set that fully covers the algebra, by selecting all canonical test exemplars of increasing length, in contrast to random sequence generation from axioms in other approaches, such as (Bernot et al, 1991; Doong and Frankl, 1991; 1994).

To improve the quality of test-case selection and test coverage in agile methods, various attempts have been made to include lightweight specifications, such as simple X-Machines derived from XP story cards (Holcombe, et al., 2001), which were used to generate complete system acceptance tests. Elsewhere, X-Machines were synthesized from UML use cases and used to generate complete test sets for the lightweight UPEDU method (Dranidis, et al., 2004; Robillard and Kruchten, 2002). When X-Machines are used for class unit testing, the generated test sets are smaller and detect more faults than the handcrafted tests produced by programmers (Holcombe, 2003). Similarly, while *JUnit* regression tests cannot confirm the preservation of behaviour in extended or modified classes (Simons, 2005), this can be achieved by regenerating the test-sets completely from a refined specification that is proven compatible with the base specification (Simons, 2006).

1.2   Lazy Systematic Unit Testing

While there is now considerable evidence that the addition of even very simple specifications permits the generation of efficient, complete test sets for agile methods, it is equally clear that aficionados of XP and similar approaches are quite unwilling to adopt any kind of up-front specification. Partly, this is due to the overall philosophy, which rejects all "big design up-front" (Beck, 2000; 2005), considering even simple specifications as too burdensome in a lightweight approach.

Therefore, the current work seeks to provide an exhaustive testing approach by different means, based partly on the structure of the tested code and partly on information elicited from the programmer during the testing process. The novel testing method is known as *lazy systematic unit testing*, which combines the two notions of *lazy specification* and *systematic testing*. *Lazy specification* (a term used by analogy with *lazy evaluation*) is intended to convey an approach in which there is no need to commit to a stable specification until as late in the design process as the programmer wishes. *Systematic testing* means a deterministic approach (in contrast to random test-case execution), in which the software unit is either exercised completely up to some criterion, or tested exhaustively for complete conformance to a specification.

In the *lazy systematic unit testing* method, the implicit specification of the class-under-test is inferred incrementally, using a combination of automatic analysis, predictive rules and hints supplied by the programmer. As more of the specification is acquired, the testing mode moves seamlessly from bounded exhaustive *exploration*, satisfying the bounded inter-method coverage criterion (Binder, 1996), to bounded exhaustive *testing*, equivalent to full conformance-testing to an independent finite state specification (Chow, 1978; Holcombe and Ipate, 1998); and the generated test reports become shorter and more focused until full test automation is achieved.
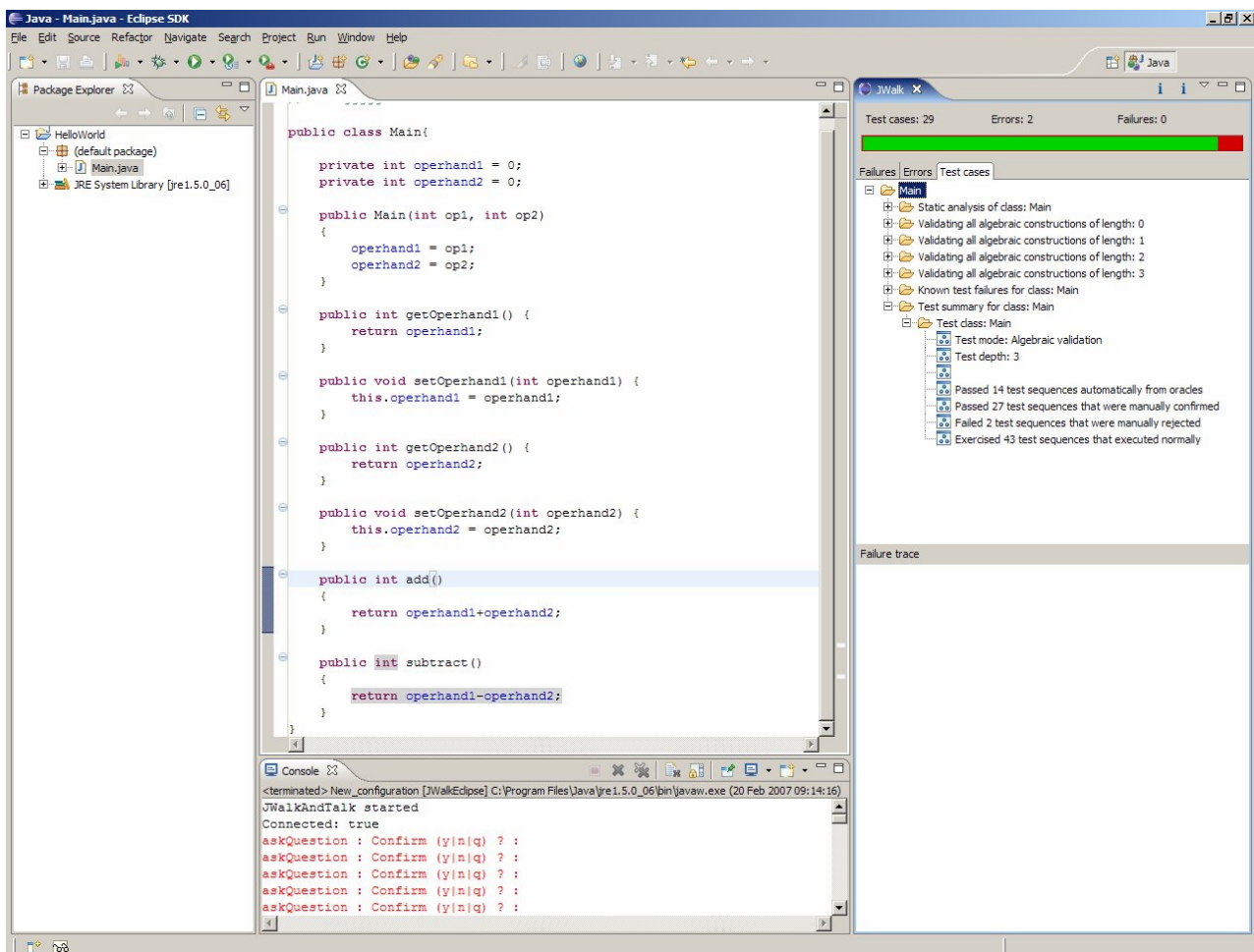
Fig 1. JWalk in the Eclipse IDE – tracing a fault to source

The testing method is exemplified in a tool called *JWalk* (Simons, 2007), which performs lazy systematic unit testing on classes compiled in the *Java* programming language. Currently, the tool

exists in two versions, as a command-line utility program, which generates test reports on standard output, and as an API that can be integrated with third-party programs, using the standard *Java* event-model to communicate between *JWalk* and the third-party user interface. *JWalk* has been successfully integrated as a plug-in for the *Eclipse IDE* environment (Eclipse Foundation, 2007), for use by a second-year class of undergraduate students (sophomores) in external public service and business development projects. In this configuration, it is used both for conformance testing and source debugging (see figure 1).

In its *exploring* modes, the tool systematically exercises the class-under-test, reporting the results of test-sequence evaluations to the programmer and highlighting any exceptions raised, c.f. (Csallner and Smaragdakis, 2004). These modes include *protocol exploration*, *algebraic exploration* and *design state exploration* (see section 3), which progressively abstract over the notion of state, c.f. (Xie, et al, 2004). In its *testing* modes, which include *algebraic testing* and *state-based testing* (see section 3), the tool interacts with the programmer, asking him or her to confirm the correctness, or otherwise, of certain key test results. These oracle values are used in predicting further test outcomes, constantly raising the level of automated testing. Predictions are based on a conservative partial order reduction that maps test sequences into equivalence classes that yield the same outcome. The set of oracle values may be updated dynamically, as the design of the class-under-test evolves, and the testing process adapts gracefully to changes in the implicit specification.

The tool exhibits several novel features, including: the ability to move seamlessly between exploring and testing modes; the ability to analyse test results while testing is in progress, in order to control dynamically the growth of test sequences by selective pruning; the ability to detect the abstract ("high level") *design states* (McGregor, 1994) of the class-under-test, using an algorithm based on simple predicates that naturally form part of the class's interface; and the ability to learn a specification incrementally. The chief benefit of lazy systematic unit testing is that it brings the full power of specification-based testing to non-formal, agile software development approaches.

1.3   Overview of the Paper

In the rest of this paper, section 2 presents an overview of object-oriented unit testing, describing the context in which *JWalk* was developed. Section 3 presents the key features of *JWalk*, focusing on the test sequence generation algorithms, the state-space induction algorithms, the use of generators for arguments and stubbed objects and the test result prediction strategy used with oracles. Section 4 describes practical experiences in using *JWalk* on a number of examples, introducing the notion of *oracle efficiency*, a metric expressing the degree of automation achieved during successive test cycles. Section 5 compares and contrasts *JWalk* in more detail with other recent tools identified in section 2; and explains how *JWalk*'s lazy systematic unit testing method converges with the proven *X-Machine* testing method (Holcombe and Ipate, 1998). Section 6 summarises the paper's main findings in the conclusion.

2.   Object-Oriented Unit Testing

The motivation behind *JWalk* is to improve the quality of quick-turnaround unit testing available for developers in the agile software development community. The benchmark for this is *JUnit* and its derivatives (Beck, 2004; JUnit, 2007). The main advantage of *JUnit* is its ability to save test scripts created by programmers, in parallel with evolving production code, and repeatedly re-run the saved tests, during regression testing. The test-driven development philosophy of XP (Beck, 2000; 2005) requires programmers to create tests before coding is complete; and to run and re-run the saved tests until the modified or extended code passes all the tests.

The test cases used in *JUnit* are handcrafted. Programmers write test cases, with names such as "testX" for some desired property X, which become the methods of a test-harness class. The method body of a test case asserts some arbitrary equivalence, inequality or other relationship that is supposed to hold. The *JUnit 4.x* tool recognises and extracts such test cases automatically by code introspection (by parsing *Java*'s special code *annotations*). While test-sets are automatically collated and repeatedly executed on demand, it is possible to argue that this still does not make the best use of automation during class unit testing.

Programmers find it hard to foresee problems caused by unexpected interleavings of methods, which affect the state of the test object, especially where inherited methods are interleaved with the new methods defined in a subclass (Simons, 2005). For this, a systematic exploration of the class's state-space is required (Buy, et al., 2000; Xie, et al., 2004). So, it makes sense for testing tools to seek to explore systematically what programmers habitually fail to discover through handcrafted tests. Ideally, systematic testing is driven from specifications (Ipate and Holcombe, 1997; Chen et al., 2001), yet agile approaches like XP forego any formal specification stage. A common reason cited for this is that the prototype software system inevitably evolves faster, in response to changes in user requirements, than any specification can be sensibly maintained (Beck, 2000).

2.1 Generating Exhaustive Tests from Code

An obvious approach would therefore be to generate tests systematically from the code, which could at least assure that the code had been exhaustively exercised. White-box testing tools, such as *Cantata++* (IPL, 2007), instrument the tested code, so that the tool can determine whether decision-points, branches or statements have been covered. In general, the complexity in class unit testing depends less on nested branching, and arises more from the interaction of simple, short methods with the class's encapsulated state variables. A number of *Java* testing tools, such as *JTest* (Parasoft, 2007) and *JCrasher* (Csallner and Smaragdakis, 2004), exploit Java's reflection capability to infer the method signatures of a compiled class, then construct random test sequences consisting of chains of method invocations, for which suitable arguments are randomly generated.

A disadvantage of exploration-based tools is that they cannot determine whether a raised exception represents a program fault, or merely a violated precondition. Likewise, they cannot tell whether a non-exceptional result is correct, or a semantic fault. This would require an independent specification of the expected results. *JCrasher* and similar tools are fault-finders, rather than conformance testing tools. They typically provide no guarantee about the coverage of randomly generated test sets. According to Xie, et al. (2005), neither *JCrasher* nor *JTest* satisfy the branch-coverage criterion (Bezier, 1990), let alone the stronger bounded intra-method path coverage criterion (Ball and Larus, 2000). For these, a more systematic approach to test-case generation is required.

2.2 Inferring Abstract States from Code

Class unit testing should at least explore the class's state-space systematically. While (Buy, et al., 2000) propose a number of techniques to improve the state coverage of test sequences, including dataflow analysis, symbolic execution and automated deduction, the complete unit testing approach favoured in this paper bears a greater affinity with the state abstraction approach used in *Rostra* (Xie, et al., 2004) and *Symstra* (Xie, et al., 2005), although a different method is eventually used here to identify high-level *design states*. Xie, et al. (2004) propose up to five different abstractions over state (see section 5.2.1). In this, and the later work (Xie, et al., 2005), which also includes systematic symbolic execution of the object, the emphasis is on filtering randomly generated

JCrasher or *JTest* sequences to preserve those exemplar sequences that uniquely exercise each identified state. A significant reduction in test-set size is achieved for no loss of coverage.

Reverse-engineering approaches to state induction seek to cluster the atomic states hypothesized for every program statement. Variations of the *k*-tail algorithm may be used to infer states from execution traces (Lorenzoli, et al., 2006). Another example of low-level state clustering is *Java Pathfinder* (Visser, et al., 2003; Lerda and Visser, 2001). This runs *Java* bytecode programs in a specialised virtual machine that is used as an explicit state model checker. The tool explores all potential execution paths, clustering equivalent states, finding deadlocks and unhandled exceptions. The tool is highly effective in detecting bugs in multithreaded programs. The "state matching" ability of *Java Pathfinder* has also been applied to improve the coverage of test sequences, increasing their state-related discrimination (Visser, et al., 2006). This test-filtering goal is shared by (Xie, et al., 2004; 2005; Marinov and Kurshid, 2001).

Partial order reduction on execution traces is the main technique used by *Java Pathfinder* to reduce the state-space. State clustering may be improved by also requiring explicit branch coverage (Yuan and Xie, 2005) and the granularity of states may be raised even higher through user-supplied abstraction functions (Grieskamp et al., 2002; Xie, et al., 2004). A less intrusive "state matching" is obtained through observational equivalence (Bernot et al., 1991; Doong and Frankl, 1994; Henkel and Diwan, 2003). The latter algebraic approach is potentially expensive, because of the growth of observer-sequences needed to characterise each state.

2.3   Testing with Partial Specifications

The handcrafted tests of XP (Beck, 2000; Beck, 2005) constitute an informal kind of specification, since the tester supplies oracle values ("expected results") for each test. In some ways this is better than blind exploration, as each test seeks to assure that the software conforms to some intended design property; but the handcrafted tests are never exhaustive. A prerequisite for automating *complete* functional testing is a complete and consistent formal specification.

While programmers find simple state machines relatively easy to comprehend (Holcombe, et al., 2001; Holcombe, 2003), these only specify the modal properties of a system. States can be supplemented by more precise observations, such as control flow (Ural, et al., 2000) or observations on internal variable values (Petrenko, et al., 2004). However, in the envisaged agile testing context, it is not really tractable to accommodate more complex multi-level specification and testing for each class, since there is already some resistance to adopting simple state-based specifications. To specify the exact properties of a class most succinctly requires the power of an inductively defined data type algebra (Goguen and Malcolm, 1997), and knowledge of subtle recurrence relations. This is more in the province of mathematicians than of agile developers (but see (Henkel and Diwan, 2003), discussed below).

It is worth considering whether automatic tool support might be offered to help agile developers induce a complete formal specification from their code. The state-of-the-art is constantly improving here. One important strand of research is the kind of property mining carried out by the *Daikon* tool (Ernst, 2000; Ernst et al., 2001; 2007; Perkins and Ernst, 2004) and its derivatives, such as *Agitator* (Boshernitsan, et al., 2006; Agitar, 2007). *Daikon* establishes the input space of the tested unit from an analysis of signatures and program constants, then repeatedly executes the tested unit on random input vectors and draws inferences about invariant relationships that seem to exist between program variables. The *Agitator* tool allows the tester to choose interactively whether to promote each observation to an axiom of the system; or to ignore it as an over-generalisation; or to flag it as a fault in the system's implicit specification (and implementation).

Inferred observations are also known as "operational abstractions" in the *Jov* tool (Xie and Notkin, 2003), which uses *Daikon* as a filter on random tests generated by *JTest*, to establish test equivalence classes. *Daikon* has been incorporated in other tools, such as *DSD-Crasher* (Csallner and Smaragdakis, 2006a), a hybrid bug-finder, which combines dynamic invariant detection with static program analysis and dynamic testing; and *Eclat* (Pacheco and Ernst, 2005), a test refinement tool, which uses *Daikon* to induce a model from successful test-runs, then generates more discriminating tests that are predicted to be "operational violations".

2.4   Inferring Complete Specifications from Code

While *Daikon* has successfully been shown to re-learn the known invariants of a system (Ernst, et al., 2001), it cannot guarantee that its learned specification is complete. It can help to identify tests for "operational violations"; however, these are more often precondition violations, than indicative of genuine software faults (Xie and Notkin, 2003). *Daikon* may have trouble distinguishing the properties of an abstract interface from the properties of some satisfying implementation (Csallner and Smaragdakis, 2006b). Perhaps the most ambitious work on the automatic induction of specifications is the semi-automated inference of data type algebras (Henkel and Diwan, 2003). This approach maps a *Java* class to an algebraic signature, then generates and evaluates many ground terms, proposing equations, typically equivalences between ground terms. The important generalisation step induces quantified axioms, which succinctly capture many ground term equations. While this approach is not guaranteed to be complete, because of dynamic inference, it seems to discover relevant axioms.

The approach has also been embedded in a tool to help programmers write and debug algebraic specifications (Henkel and Diwan, 2004). The tool maps the candidate specification to a prototype *Java* class and simulates its behaviour, which can be compared with the behaviour of some hand-coded concrete *Java* class. Discrepancies inform the programmer about faults in the algebra, or possibly in the concrete class. Eventually, this kind of supported, interactive development of specifications may prove attractive to the agile community, if it can be persuaded that algebras are not mathematically too difficult. Parameterizing specifications for testing purposes has received support elsewhere (Tillmann and Schulte, 2005a; 2005b).

3.   The Design of the *JWalk* Tool

Lazy systematic unit testing with *JWalk* requires no initial formal specification. Instead, aspects of the intended design are acquired incrementally, and may also be changed as the programmer edits the class-under-test (CUT) in an iterative design-and-test cycle (see figure 2). The signatures of the CUT's public operations are acquired by static analysis and observer/mutator properties are learned by dynamic analysis. If the CUT offers any *Boolean*-valued state predicates, then these will be used dynamically to identify significant design states. Finally, the programmer may also supply oracle values when running the *JWalk* tool in its interactive testing modes, which accumulate over several testing cycles and adapt gracefully to changes in the design. At any time, systematic testing from the specification obtained so far may be carried out.

The *JWalk* lazy systematic unit-testing tool exists in two versions, as a command-line utility program; and as an API toolkit for integration with third-party programs and user interfaces, such as the *Eclipse IDE* (Eclipse Foundation, 2007). A stable version of the toolkit is freely available for download at the *JWalk* website (Simons, 2007) and there are plans to release a version with an integrated graphical user interface at a future date.
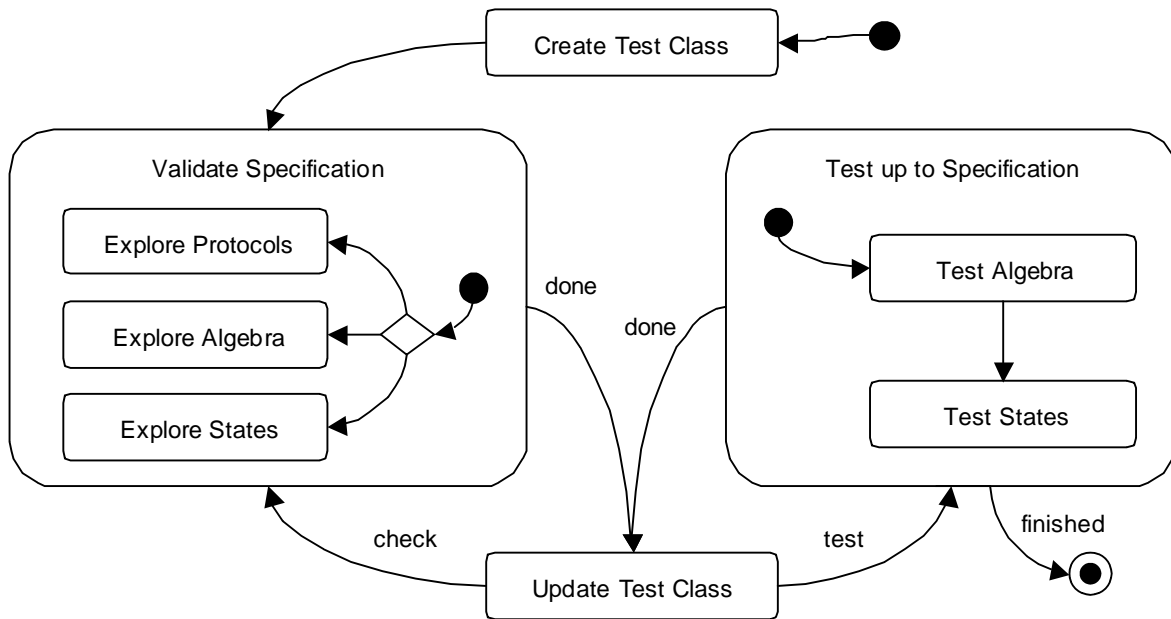
Fig 2. Workflow in the *JWalk* design-and-test cycle

Executing the command line utility in the *Java JDK* environment is extremely simple:

```
java org.jwalk.JWalk MyClass
```

will invoke the *JWalk* tool on the compiled *Java* class file *MyClass.class* and will perform, by default, bounded exhaustive protocol exploration of *MyClass* to a depth of three, generating a test report on standard output. The behaviour of the tool is controlled by further command-line arguments, indicating the test mode and test depth. The different test modes include:

- bounded exhaustive protocol exploration – in which all interleaved method paths (including inherited methods) are explored to the given depth (see sections 3.2.2 and 4.1.1);

- bounded exhaustive algebraic exploration – in which all observations on all interleaved constructor/mutator-method paths are explored to the given depth (see sections 3.3.1 and 4.1.2);

- bounded exhaustive state-based exploration – in which all interleaved method transitions are fired from each of the abstract design states of the class, to the given depth (see sections 3.3.3 and 4.1.3);

- bounded exhaustive algebraic testing – in which oracle values are acquired interactively to automate the confirmation of tests generated by bounded exhaustive algebraic exploration (see sections 3.4 and 4.2.1); and

- bounded exhaustive state-based testing – in which oracle values are acquired interactively to automate the confirmation of tests generated by bounded exhaustive state-based exploration (see sections 3.4 and 4.2.2).

The test-depth represents the maximum length to which interleaved method paths are constructed and exercised, which, in the case of the state-based styles of exploration, does not include the prefix sequence required to reach each state. Other terms such as "protocol", "mutator", "observer" and "design state" have particular senses, which are explained in more detail below.

3.1 Open Architecture of the *JWalk* Tool

The *JWalk* tool is designed in an object-oriented style, to facilitate future extensions, and it exploits certain design patterns, such as the *Strategy*, *Observer* and *Template Method* patterns (Gamma, et al., 1995), as well as specific features of the *Java* programming language, such as reflection. The architecture of the *JWalk* tool is illustrated in figure 3, showing the organisation of its components. The specialised functions of some of the major components are described below.
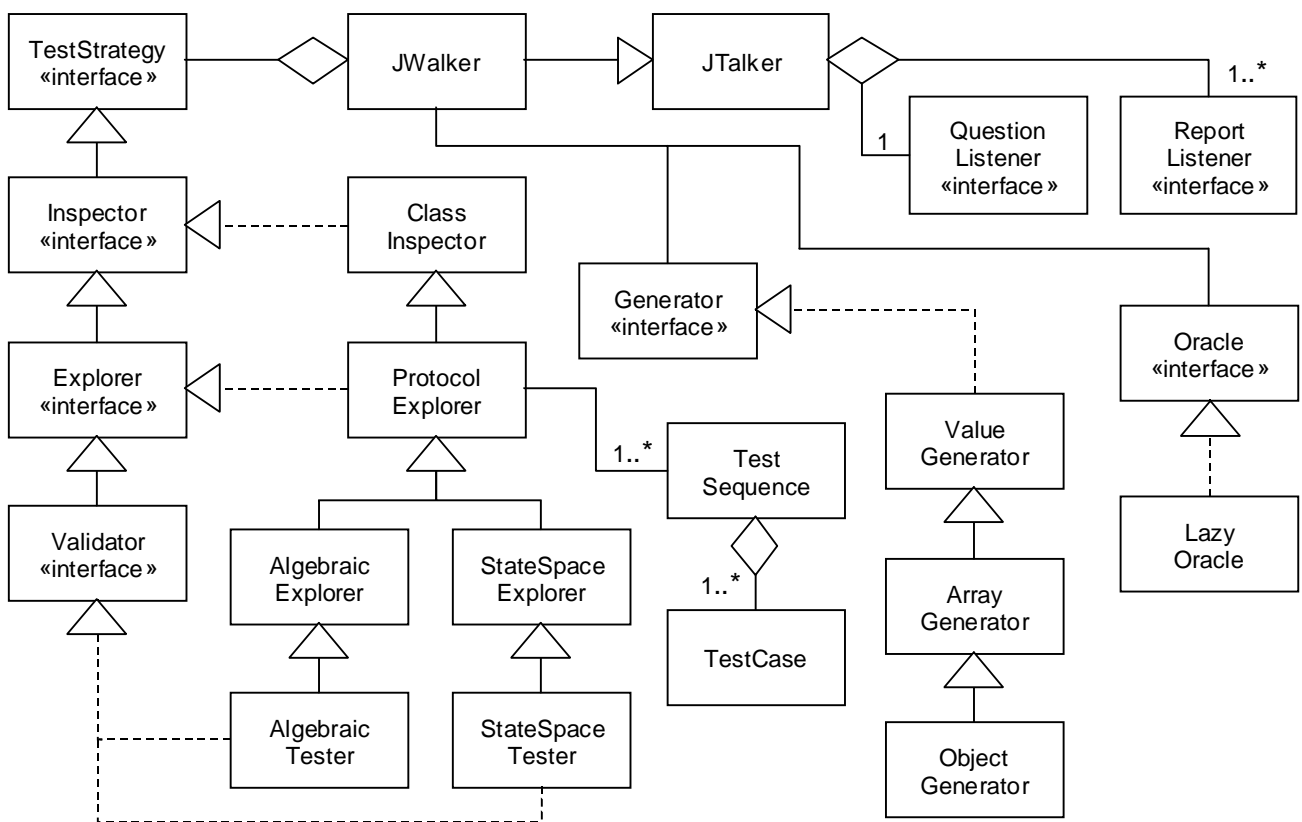
Fig.3. Architecture of the *JWalk* testing toolkit

3.1.1 The *TestStrategy* Hierarchy

TestStrategy is an interface, the abstract root of a tree of different components that perform static and dynamic analysis of the class-under-test (CUT). The subinterfaces *Inspector*, *Explorer* and *Validator* describe extended capabilities appropriate to each static or dynamic strategy. Various concrete components implement these interfaces; and may satisfy more than one. Subclass components extend the testing capabilities of the immediately dominating superclass in figure 3, for example:

- *ClassInspector* – performs a static analysis of the CUT by reflection, to extract its constructors and methods, and their type signatures;

- *ProtocolExplorer* – performs a static analysis, and then constructs and executes test sequences for a bounded exhaustive protocol exploration;

- *StateSpaceExplorer* – performs a static analysis, then executes a state-space discovery search, before constructing and executing test sequences for a bounded exhaustive state-based exploration; and

- *StateSpaceTester* – performs a static analysis and state-space discovery search, before constructing and executing test sequences for a bounded exhaustive state-based test, whose results are confirmed against oracle values.

The top-level *JWalk* program interprets the command line parameters and from these constructs a suitable subclass of the abstract *TestStrategy*, according to the kind of testing to be carried out, following the *Strategy* design pattern (Gamma, et al., 1995). The *TestStrategy* object accepts the CUT and proceeds with a static analysis and the eventual dynamic analysis.

Other important components include *TestCase* and *TestSequence*. *TestCase* models a single invocation of either a constructor or method of the CUT, while *TestSequence* models a chain of such test cases, starting with a constructor and continuing with a unique combination of methods. The *TestStrategy* object builds up test sequences of increasing length for later evaluation, according to its particular strategy. Each *TestCase* is supplied with controlled arguments, manufactured by a *Generator* object, which by default is an *ObjectGenerator*. A different generator may be requested by supplying a different *Generator*-class name on the command line.

3.1.2   The *Generator* Hierarchy

JWalk's main testing assumption is that it is the unexpected interactions of an object's methods with its state that give rise to faults (see the earlier discussion in section 2.1). This is different, for example, from the category-partition testing assumption (Irvine and Offutt, 1995) that faults arise from unexpected combinations of values in the input space. The main goal for test sequence generation is therefore to make it possible to exercise the CUT's methods in all possible combinations, under repeatable conditions. For this, a relatively simple strategy may be used to synthesize method arguments.

Generator is an interface, the abstract root of a tree of test input vector synthesizers. These play an important role in generating controlled arguments for the *TestCases*, ranging from unique primitive values and arrays of values, to symbolic objects standing for equivalence-classes of inputs and custom stubbed objects designed to force certain kinds of behaviour under test. Three standard generators are provided, in a specialisation hierarchy (see figure 3):

- *ValueGenerator* – can synthesize a sequence of quasi-unique values for any of *Java*'s basic types *boolean, byte, char, short, int, long, float* and *double*;

- *ArrayGenerator* – in addition to basic types, can synthesize primitive arrays of increasing lengths of any type (basic types; and object types); and

- *ObjectGenerator* – in addition to the above, can synthesize controlled exemplar instances of *any* class type with a default constructor (and of certain *Java* library classes with other kinds of constructor).

While *ObjectGenerator* is adequate for standard purposes (for example, the generated symbolic instances allow the comparison of objects by reference across different test cycles; the default sequence of generated integers allows testing for all of the normal, underflow and overflow indexing cases of collections), it is expected that developers will provide further custom generators, satisfying the *Generator* interface, when they wish to generate non-standard input sequences, or supply particular stubbed objects or controlled instances. One reason to do this might be if a category-partition style of testing was desired, where it is anticipated that different partitions in the input space for method arguments will trigger significantly different behaviour. Instructions for customising generators are provided in (Simons, 2007).

If one of the *testing* modes is selected (by executing an *AlgebraicTester* or a *StateSpaceTester*), a *LazyOracle* is also deployed, to collect oracle values interactively and reuse oracles to confirm test results automatically. The architecture is designed around the *Oracle* interface, so that other kinds of oracle could eventually be deployed. One future possibility might be an *AlgebraicOracle* that could predict test results from a supplied algebraic specification, for example.

3.2   Static and Dynamic Analysis Techniques

JWalk seeks to infer as much of the intended design of the CUT as possible from the compiled *Java* code, before it asks the programmer for extra information. While this might require full source code analysis in some languages, identifying the signatures of operations is made much easier in *Java* by the *Java Reflection API*, a feature also exploited by some other tools (Csallner and Smaragdakis, 2004).

3.2.1   Static Analysis by Reflection

Reflection refers to the facility provided in certain object-oriented languages, whereby objects and classes may introspect at runtime about their own definitions. Reflection is available in languages that have a strong metaobject facility (Kiczales, et al., 1991), in which classes exist also as runtime concepts, in order to allow programmatic manipulation of their own definitions.

The information extracted by reflection includes: the type of the CUT, the names and type signatures of its public constructors and methods (it is assumed that non-public features cannot be invoked directly, but only indirectly through public methods). The extracted methods include all inherited public methods for the CUT, since systematic testing must exercise all possible interleavings of inherited and local methods (Simons, 2006). However, an optional filter is provided to exclude the standard built-in methods inherited from *Object*, the root of the Java class hierarchy, which deal with the internals of the *Java* kernel, such as the synchronization of threads.

Further to this basic analysis, the CUT's methods are partitioned into *procedures* (which return a void result) and *functions* (which return a non-void result). These are taken as first approximations to the *observers* (pure functions) and *mutators* (side-effecting operations) of the CUT, which are later determined empirically by dynamic analysis (see section 3.3.1). The functions are filtered further to extract any *state predicates*, *Boolean*-valued functions with no argument. These are used later in the dynamic discovery of abstract design states (see section 3.3.2).

3.2.2 Dynamic Analysis by Test Execution

All dynamic analysis is performed by constructing test sequences and executing these on new instances of the CUT, confirming the result either semi-automatically (in the testing modes) or by visual inspection of a test report (in the exploring modes). In some modes, the CUT may be placed into a known starting state, before the main test sequence is executed. The basic sequence generation strategy is supplied by *ProtocolExplorer* and inherited by *StateSpaceExplorer*, where it is applied after the CUT has been driven into a particular state; and is adapted in *AlgebraicExplorer* to remove observers from the prefix of test sequences.

The basic sequence generation strategy of *ProtocolExplorer* seeks to exercise the CUT's protocol completely. The term "protocol" here refers to any interleaved sequence of the CUT's public methods, prefixed by one of its public constructors. *Bounded exhaustive protocol exploration* includes paths beginning with every possible constructor, continuing with all possible interleavings of methods, up to some upper bound on the path length. The test set includes all initial constructor sequences, treated as length 0, then all method invocation sequences of length 1, length 2, length 3 and so on, up to a maximum depth k. Mathematically, this can be expressed as the following formula for the "protocol-walking" test set T_{Pk}:

$$T_{Pk} = C \otimes \{1 \cup M_1 \cup M_2 \cup \ldots \cup M_k\}$$

where C is the set of unit constructor sequences, M_1 is the set of unit method sequences, M_2 is the set of all method pair sequences and so on; and $\otimes$ is the concatenated product, appending every sequence in the right-hand set onto every sequence in the left-hand set. Listing 1 illustrates in pseudo-code how the same test set might be constructed by an imperative algorithm (in which `extend` and `insert` are understood to modify the sequence, or set in question):

```
test_set := empty
last_set := empty
for i := 0 to max_depth_k
   if i == 0 then
      for c := 1 to max_constructors
         test_seq := empty
         extend(test_seq, constructor[c])
         insert(test_set, test_seq)
      last_set := test_set
   else
      next_set := empty
      for j := 1 to size(last_set)
         if executed_ok(last_set[j])
            for m := 1 to max_methods
               test_seq := copy(last_set[j])
               extend(test_seq, methods[m])
               insert(next_set, test_seq)
               insert(test_set, test_seq)
      last_set := next_set
```

Listing 1. Algorithm for Exhaustive Protocol Exploration

This algorithm relies on copying every test sequence from the previous cycle (those in *last_set*) and extending each copy by every possible method to yield the sequences in the next cycle (those in *next_set*). Test sequences will therefore contain every single constructor, every constructor followed by every method, and longer sequences which extend every existing sequence by every single method. This means that sequences will eventually contain repetitions of methods, including combinations that are not usually anticipated by human testers, such as repeated calls to the same

access method, or access methods embedded in the prefix, evaluated only for their possible side-effects. These exhaustive sequences are required to satisfy the coverage criteria for the complete state-based testing approach supported by *JWalk* (Chow, 1978; Holcombe and Ipate, 1998) – (see section 5.3).

The algorithm from listing 1 assumes that test *generation* and test *evaluation* are interleaved activities. In each test cycle, a filtering step is applied to the executed tests of the previous cycle, to remove edges from the graph that raised exceptions in the previous cycle. This is because any longer test sequence grown from a halting edge would automatically halt at the same point in the next cycle. After reporting the exception, *JWalk* prunes halting sequences, to reduce the size of the search space. The tool nonetheless keeps a tally of all longer (now virtual) paths that would have halted at the same point.

3.3   Exploring Different Abstractions over State

JWalk's main testing assumption (see section 3.1.2 above) is that it is primarily the unexpected interactions of interleaved methods with an object's state that reveal faults in the object's implicit design. This is similar in philosophy to (Xie, et al., 2004; 2005; Grieskamp, et al., 2002; Buy et al., 2000), but not identical in approach (see detailed comparisons in section 5.2). The tool directly supports up to three different abstractions over state:

- *all interleaved methods* – where every distinct sequence of methods is presumed to give rise to a potentially different state;

- *all algebraic constructions* – where every distinct sequence of constructor and mutator methods is presumed to give rise to a potentially different state; and

- *all abstract design states* – where every valid and unique combination of predicate observations over an object's variables is presumed to give rise to a potentially different design state.

The *all interleaved methods* abstraction represents the finest-grained notion of state. It assumes that any method invocation may create a unique object state (for each argument equivalence-class); and even simple access methods are anticipated to have possible state-modifying side effects. The default mode known as "protocol-walking" constructs sequences with this granularity, presenting an exhaustive test report to the user (see section 4.1.1), which can grow to become lengthy for classes with many methods, or for longer bounded path-lengths.

3.3.1   Approximating Algebraic States

The *all algebraic constructions* abstraction represents an intermediate-grained notion of state. In standard functional treatments of algebra (Goguen, et al., 1993; Goguen and Malcolm, 1997) the primitive operations of a datatype are known as (algebraic) *constructors* and these may be used to create every unique instance of the datatype. The constructors of a *Stack* would therefore include both *newStack* and *push* (in the functional universe), because these two operations alone may be used to construct every conceivable *Stack* instance. So, the notion of algebraic states corresponds to all (syntactically legal) interleavings of the type's primitive constructor functions.

In an object-oriented universe (with side-effects), algebraic constructions are mapped onto a class's constructors and some of its mutator methods. It is impossible to distinguish automatically from code analysis alone which of the class's mutator methods should be counted as primitive (like *push*,

an algebraic constructor) and which are derived operations (like *pop*, an algebraic transformer). This would require independent information about the algebraic category of each operation. Instead, an approximation of the class's primitive operations is taken to include its constructors and *all* of its mutator methods. This over-estimates the number of primitive operations, but is guaranteed to include all of them. The set of all algebraic constructions is therefore approximated conservatively by constructing all sequences beginning with a constructor, followed by all possible interleavings of the CUT's mutator methods. This is guaranteed (by the above argument) to contain more sequences than strictly necessary to reach all distinct algebraic states.

Bounded exhaustive algebraic exploration includes paths beginning with every possible constructor, continuing with all possible interleavings of mutator methods, and terminating with a single observer or mutator method, up to some upper bound on the method path length k. Mathematically, this can be expressed as the following formula for the "algebra-walking" test set T_{Ak}:

$$T_{Ak} = C \otimes \{1 \cup N_1 \cup N_2 \cup \ldots \cup N_{k-1}\} \otimes M_1$$

where C is the set of unit constructor sequences, $N_1 \subseteq M_1$ is the set of unit mutator sequences, $N_2 \subseteq M_2$ is the set of all mutator pair sequences and so on; and the final term M_1 is the set of unit method sequences, as before. The concatenated product operator $\otimes$ is used twice to extend all constructors with all mutator-paths, then again with all single methods. Listing 2 illustrates in pseudo-code how the same test set might be constructed by an imperative algorithm.

```
test_set := empty
last_set := empty
for i := 0 to max_depth_k
   if i == 0 then
      for c := 1 to max_constructors
         test_seq := empty
         extend(test_seq, constructor[c])
         insert(test_set, test_seq)
      last_set := test_set
   else
      next_set := empty
      for j := 1 to size(last_set)
         if modified_ok(last_set[j])
            for m := 1 to max_methods
               test_seq := copy(last_set[j])
               extend(test_seq, methods[m])
               insert(next_set, test_seq)
               insert(test_set, test_seq)
      last_set := next_set
```

Listing 2. Algorithm for Exhaustive Algebraic Exploration

This algorithm generates a smaller test set than listing 1 (see also section 4.1.2). The only difference is in the way that edges are filtered before they are grown in the iteration step. Only those paths terminating in a normally-executing *constructor* or *mutator* method are extended in the following cycle. Paths terminating in an *observer* method, or which raised an exception (see 3.2.2 above) are pruned on the fly, before the next generation cycle. The similarity between these two algorithms suggested the obvious implementation strategy using the *Template Method* design pattern (Gamma, et al., 1995); and in fact, variations of the same basic algorithm are used to discover design states and then extend transition paths from these states (see section 3.3.2 below).

During algebraic exploration, *JWalk* distinguishes *observer* and *mutator* methods. These are determined empirically by dynamic analysis, after each test execution phase and before the next test generation phase. Initially, methods were partitioned into *functions* and *procedures* according to their signatures (see section 3.2.1 above). The *functions* may qualify as *observers* (pure functions) if they have no side-effects. To detect side-effects, a state vector is extracted from the CUT instance, before and after the execution of each *function*, using *Java*'s *Reflection API* to bypass the *private* and *protected* visibility of its attribute declarations. If the prior and posterior state vectors are identical, then the *function* is deemed an *observer*. Otherwise, it has a side-effect and is deemed a *mutator*, along with all other *procedures*.

3.3.2   Inferring Abstract Design States

The *all abstract design states* level of abstraction represents the coarsest-grained notion of state. The term "design state" comes from McGregor (1994) and denotes any state that has a meaning for the CUT's designer. *Design states* are complete partitions of the CUT's attribute space (McGregor, 1994) but may also be defined more abstractly as partitions of its observer-space (Simons, 2005). In practice, design states correspond to modes in which the CUT's behaviour changes in some significant way and may encode the context for illegal calls (exceptions), ignored events (null operations) or different execution (branching). *JWalk*'s method for discovering high-level design states has evolved across different versions of the tool, in favour of increasingly less intrusive design-for-test requirements on the CUT.

Early versions of *JWalk* (0.3 onward) required the CUT to provide a complete set of state predicates, *Boolean*-valued methods with no argument, that exhaustively partitioned the CUT's attribute space, c.f. (D'Souza and Wills, 1998). For example, a *Stack* would supply state predicates to report its *Empty* and *Default* states. This had the advantage that an automatic state discovery procedure could rapidly identify every reached state. Furthermore, it was possible to check empirically that the predicates were mutually exclusive and exhaustive. *JWalk* could search up to some bounded depth, or up to memory-exhaustion, to determine that all design states had been found and that none were overlapping.

Later versions of *JWalk* relaxed the requirement to provide total predicate coverage. From version 0.6, *JWalk* could infer at most one *Default* state for each CUT, being the state found when no other state predicate returned true. The advantage was a greater economy of expression, and the ability to infer at least one state for every object (objects without predicates have a *Default* state). State completeness was defined less strictly, as finding all explicit states and optionally the *Default* state. Searching proceeded to the limit if the *Default* state was not found. Overlapping predicates could still be detected as ill-formed.

A further sophistication was added in *JWalk* 0.6 to cater for the partitioning of state caused by subclassing (Simons, 2005; 2006), which legitimately entails overlapping predicates. For example, an abstract *Stack* interface might be realised by a concrete *BoundedStack*, which partitions the *Stack*'s *Default* state into distinct *Normal* and *Full* states. An object could therefore simultaneously satisfy the *Full* and *Default* state predicates. This required a relaxation of the mutual exclusivity rule. The state induction algorithm was amended to take account of the *most specific* predicates, in cases of partitioning, which was distinguished from ill-formed cases of overlapping.

Eventually, an analysis of programming styles showed that programmers tended to introduce quasi-orthogonal states in each subclass. For example, a *LibraryBook* may exist in the states: {*Available*, *OnLoan*}. A *ReservableBook* subclass may introduce the orthogonal states: {*Unreserved*, *Reserved*}, however, these states interact with the original states, resulting in a state product, which

completely partitions the original states. These states can all be represented in a *Boolean* matrix of two variables:

OnShelf = ¬onLoan ∧ ¬reserved	*OnLoan = onLoan*
PutAside = ¬onLoan ∧ reserved	*Available = ¬onLoan*
Borrowed = onLoan ∧ ¬reserved	*Reserved = reserved*
Recalled = onLoan ∧ reserved	*Unreserved = ¬reserved*

and the programmer need only provide the predicates *isOnLoan()* and *isReserved()* to represent these states. We believe that supporting this kind of coding style will prove most habitable to programmers, since it imposes the fewest design-for-test restrictions.

From version 0.8 onwards, *JWalk* identifies abstract design states from the *Boolean* product of state predicate observations. Searching attempts to find all truth-value combinations of atomic observations, but may still succeed without having found the full product. This may occur if predicates are *not* orthogonal, either because they are mutually exclusive, or because a superclass predicate includes one or more refined subclass predicates. In the *ReservableBook* example, all truth-value combinations are meaningful states, and will be found, since the atomic predicates are orthogonal. In the *BoundedStack* example, no logical state corresponding to *empty ∧ full* exists, since the associated predicates are mutually exclusive.

The guarantees of complete state discovery are not as obviously strong. In general, no assumption can be made about predicate dependency (one way or the other). The posterior check for complete state discovery is a heuristic, based on every atomic predicate having returned both true and false at some point. This correctly identifies redundant predicates (states that are never reached) and lack of progress (states that are never left). In practice, state discovery seems to be extremely effective. This can only be explained by an analogy with the anthropic principle ("the universe is the way it is, because we are here to view it"). If a CUT provides state predicates, then it expects the design states derived from these to be found within a reasonable space of constructions. (See also the discussion in section 5).

3.3.3 Exploring Design States

When exploring all design states, *JWalk* first computes the state cover (see below), then drives the CUT into each of its abstract design states, using each prefix from the state cover, then exercises all possible interleavings of methods, up to some chosen path depth k. The exhaustive exploration ensures that all combinations of transitions are attempted from each design state. For $k = 1$, the generated tests correspond to the *transition cover*. For $k = 2$, they correspond to the *switch cover*. For $k \geq 3$, test sequences can even validate ill-formed implementations with duplicated states and transitions, in the style of X-Machine testing (Holcombe and Ipate, 1998).

Let S stand for the state cover set that will drive the CUT into each of its intended design states. Each sequence in S already consists of a constructor, followed by zero or more mutator methods. Test sequences are constructed by extending each prefix sequence in S by increasingly longer sequences of methods in M_1, M_2, etc. This can be expressed as a mathematical formula for the "state-walking" test set T_{Sk}:

$$T_{Sk} = S \otimes \{1 \cup M_1 \cup M_2 \cup \ldots \cup M_k\}$$

where M_1, M_2, etc. are sets of method sequences of increasing length, containing all possible singletons, pairs, and so on. This "state-walking" set is calculated algorithmically in a similar

manner to the "protocol-walking" test set, described in listing 1, but with the state cover S substituted in place of the set of constructors C.

The state cover is discovered by a prior exploration of the CUT's state-space. *JWalk* constructs probe sequences, consisting of a constructor followed by interleaved mutator methods and ending in a predicate, to determine whether the predicate holds for that particular sequence. Mathematically, this can be expressed as a formula for the probe set Q_k:

$$Q_k = C \otimes \{1 \cup N_1 \cup N_2 \cup \ldots \cup N_k\} \otimes P$$

where C is the set of unit constructor sequences, P is the set of unit predicate sequences and N_1, N_2, etc. are sets of state-modifying sequences of increasing length, containing all possible singletons, pairs, and so on, of mutator methods (*mutators* were defined in section 3.3.1). In each exploratory cycle, the state-space of the CUT is explored in a breadth-first way and a predicate vector is evaluated for each path. One of the decision rules (from section 3.3.2) is applied to determine whether a new design state has been detected. If so, the exploratory sequence is added to the state cover. In this way, the state cover S always consists of the shortest sequences that reach each new state. The depth criterion *k* is a safety cut-off point. This can be set at less than the memory-exhaustion limit, to reduce searching time (which is typically around 3-5 seconds), but can be extended up to memory-exhaustion. *JWalk* recovers gracefully from this condition, with all discovered information intact.

3.4   Lazy Systematic Testing with Oracles

The *algebraic testing* and *state-based testing* modes of *JWalk* seek progressively to automate the execution of complete unit test-sets. The programmer is invited to confirm or reject particular test results that *JWalk* detects as being in some sense "novel"; and the tool repeatedly applies each learned judgement, unless told that it no longer applies. As more information is gleaned, the size of the test report generated for human inspection is reduced. At the limit, testing becomes completely automatic, producing a summary report.

Automated validation compares each actual test result against a known reference value, or *oracle*. In many other testing methods, including *JUnit* (Beck, 2004; JUnit, 2007), the programmer is expected to provide oracle values in each of the handwritten tests. For a moderate number of unit tests, this is tractable, but in the context of *JWalk*, which performs bounded exhaustive testing, the prospect of having to input many such oracle values by hand would be daunting.

3.4.1   Minimal Intervention Strategy

As a first effort-saving strategy, *JWalk* saves the programmer from having to input *any* reference data at all, by suggesting the result first. The programmer need only confirm whether this is correct or not, with a keystroke. There is no requirement to input any reference data, since the correction strategy is always to fix the software until *JWalk* can predict all of the desired results. This feature is designed especially to suit the agile development philosophy (Beck, 2000; 2005), in which the only durable artifacts are the production code and the saved tests. The second effort-saving strategy is to avoid presenting the programmer with every unseen test case. *JWalk* need only select certain *TestSequences* for interactive confirmation by the programmer, since it can predict many other expected results from information already known (see section 3.4.2 below).

During a validation cycle, the programmer is asked to confirm each presented result and may key *Enter* to accept (quicker than typing in "Y"); or type in "N", to reject the result; or type in "Q" to

exit early from the validation process, if an error was found that would have consequential effects in later tests. After one such pass, the set of confirmed oracles are saved and any failed test-cases are printed in a test report. The programmer may then fix the CUT and re-run the validation process, which re-presents all the modified cases for confirmation, but checks all unchanged cases automatically, whether these are known to be correct, or incorrect. The programmer edits the CUT until all predictions are judged to be correct.

From small-scale experiments, it was found that programmers learn quickly how to read and judge the presented test cases (see section 4). It is reasonable to expect a trained programmer to confirm 10-12 oracle values per minute. The actual number of confirmations requested depends on the size of the CUT's public interface and the depth to which testing is carried out. If editing and testing are carried out in an iterative fashion, the burden of oracle creation is amortized over the development cycle. The time taken ranges from a few seconds (5 confirmations) to around 7 minutes (80 confirmations).

3.4.2   Rule-Based Prediction of Test Results

In the testing modes, *JWalk* uses previously saved oracles to make predictions about future test results, in an effort to automate more of the validation process. This uses *strong* and *weak* predictive rules. Strong predictions allow *JWalk* to confirm test results automatically, and are guaranteed (see below). Weak predictions allow *JWalk* to assume results, unless these are contradicted, but they are not guaranteed. Where the predictions hold, *JWalk* checks the result automatically. Only novel test results (which contradict expectations) are presented for interactive confirmation.

An example of a *weak* prediction, from a static analysis of signatures, is that a sequence ending in a *procedure* is expected to return a *void* result. If this weak assumption holds, then validation is automatic. If it is violated, for example, because the procedure call raises an exception, then this unexpected result is noticed and is presented to the human tester for confirmation. There is little to be lost if the prediction fails to hold, since *JWalk* will eventually learn the exceptions to the rule (and from these, may construct new predictions). Weak predictions are only valid in one direction. For example, if a *procedure* is supposed to raise an exception, but does not, then this will not be detected automatically. However, a subsequent observation on this sequence will unexpectedly be presented, and so draw attention.

An example of a *strong* prediction, based on a dynamic analysis of states, is that *observer* methods have no effect upon state. This allows *JWalk* to map many test sequences containing prefix *observers* into equivalence-classes. So, a sequence whose prefix contains a function, which has previously been found empirically to have no side-effects, may be mapped onto a shorter sequence from which the function is eliminated. *JWalk* uses the existing oracle value for the shorter sequence to confirm the longer sequence. Sequences are eventually mapped onto equivalent sequences with no prefix observers. A strong prediction is always guaranteed to be valid. If a test sequence contained a function with unwanted side-effects in its prefix, which were later reversed by another faulty method, then the reduction rule would simply not apply, since the faulty prefix function would be classed as a *mutator*. If the prefix function were accidentally side-effect free (assuming this was not intended), then the prediction made by the reduction rule would still be valid (irrespective of the faulty function).

These predictive rules still work in degenerate circumstances. For example, if the CUT instance has an illegal path, *JWalk* will present the first observation on the illegal path for confirmation (e.g. a *ReservableBook* that is first *reserved*, then wrongly *issued* to a different person, who is

unexpectedly returned as the *borrower*). If the CUT instance has a duplicated path leading to a redundant state that randomly produces different observations, the first observation will be confirmed on first sight (as a *pass*, or *fail*) and will abnormally reappear for confirmation in a subsequent cycle (the cycle in which the random value first changes), so drawing attention.

When using *JWalk* in its *testing* modes, the assumptions made by the predictive rules are in practice highly useful. For higher-coverage state-based tests, the oracles learned during algebraic testing allow over 90% of the state-transition paths to be confirmed automatically (see section 4.3). In general, a balance should be maintained between using the *exploring* and *testing* modes, since the former present all cases for visual inspection. *Testing* assumes that the inferred specification is mostly valid (but it could be inconsistent in places). *Exploring* assumes that the implicit specification is still being validated.

3.4.3   Reuse of Symbolic Oracle Values

The storage of oracle values for reuse in subsequent test cycles poses non-trivial problems. While simple types, such as *int* or *char*, have printed representations from which the original values can be reconstructed by reading, object types do not in general have readable representations. *Java*'s built-in *serialisation* mechanism permits the saving and restoring of equivalent (in the sense of isomorphic) graphs of objects, provided that the class of each implements the *Serializable* interface. Enforcing this as a design-for-test condition proved somewhat restrictive and prevented testing any CUTs that referenced non-serialisable classes from the *Java* kernel.

However, the more serious problem was trying to judge when the original and reconstructed objects should be counted as pair-wise equal. By default, objects are compared by identity, which does not yield a useful measure of equality after reconstruction, since a reconstructed object is always non-identical to the original. The *equals()* method was unreliable, since this uses a mixture of deep, shallow and identical comparison, according to which version of the method is obtained. For this reason, serialisation was abandoned as the storage policy for oracle values.

Instead, the *JWalk* tool constructs symbolic oracle values for every object type, based on its type name and the point in its *TestSequence* when it was created. By default, every *Generator* class constructs a map from objects created in each test cycle to their canonical oracle representations. Objects constructed from the same argument values at the same point in a test-run will map to the same oracle. This means that, on subsequent test-runs, newly generated instances, which normally would be treated as non-identical and therefore not equal to the original instances, are judged to be in the same oracle equivalence class. This notion of equality is mediated by the *Generator* classes, so may be adapted by the programmer in more specialised generators, if this is required.

3.4.4   Code Evolution and Graceful Degredation

Lazy systematic testing allows the programmer to be more or less relaxed about the amount of specification-related information supplied, before systematic testing can begin. This is helpful, since it allows continued experimental evolution of the unit's coding in the early development stages. Even if the design of the CUT is modified, the *JWalk* tool adapts to any implicit changes in the specification; and the validity of any saved oracle values degrades gracefully. This is an important property for the tool to support the goals of agile software development.

Initially, the programmer may choose to exercise the CUT in one of the exploring modes. Later, he or she may switch to one of the testing modes. Oracle values are acquired in a cumulative way, up to the path depth requested. Retesting in either of the testing modes will reuse oracle values from

the existing saved set, confirming known results automatically and only interacting with the programmer when longer, or novel test sequences are first encountered.

If the result of a test is recognised as incorrect, the programmer can confirm this provisionally as a fault. *JWalk* will remember the negative result in exactly the same way as positive results, confirming all retest failures automatically, until the programmer edits the faulty code, such that it generates a different result. This is useful if there are many faults to fix, which cannot be addressed in a single editing pass. Eventually, the code will be fixed to generate the correct result, which will require confirmation in the next test cycle.

If the design of the CUT is now changed, by adding a new method to the class's interface, retesting will confirm automatically all previously encountered sequences, but will seek confirmation for novel interleaved sequences (one exemplar from each new equivalence class). If a method is edited and this affects the value of its result, *JWalk* will recognise that one oracle has been invalidated and will seek confirmation of the new result (once for each equivalence class). In this way, the set of oracles is constructed incrementally and converges with the desired behaviour of the CUT.

4.   Testing Experiences using *JWalk*

During early development of the *JWalk* tool, test examples consisting of simple abstract data types, such as *Stack, Queue* and *Vector*, were used to evaluate the tool's performance. These experiences helped to focus the reporting style, promote the reuse of oracle values across different testing modes and develop the tool's recovery strategy after precondition violations. Later evaluation concentrated on testing merged properties of classes after inheritance. The *AbstractStack* and *BoundedStack* pair were used to validate the interleaving of local and inherited methods; and likewise to demonstrate the detection and partitioning of inherited states. These and the *LibraryBook* and *ReservableBook* pair were used to evaluate the product algorithm for state inference; and to extend saved oracles to test subclasses.

To demonstrate its ability to scale, *JWalk* was used to exercise some of the library classes from the standard *Java* distribution, such as *Object*, *Character, Integer* and *String*. This was to ensure that the static and dynamic analyses could handle much larger units, for which only the compiled library code was available. Other third-party classes from the *nanoxml* project (De Scheemaecker, 2007) were tested. A weakness was found in *ObjectGenerator*'s failure to synthesize instances of *Streams* without default constructors. An interim solution was found using custom generators, but further work is needed for a general solution.

Finally, the effectiveness and efficiency of oracle-based learning was evaluated. An interactive workshop invited participants to make arbitrary modifications to test classes, for which oracles had previously been trained. Retesting found all mutation faults, within an exploration depth of 1-3. Data was also collected on the ratio of new manual confirmations to automatic validations, for cumulative testing up to different depths in the two testing modes.

4.1   Bounded Exhaustive Exploration

JWalk produces several different kinds of report, depending on the test mode chosen by the programmer. In the early stages of development, it is anticipated that one of the *exploring* modes will be used. These generate one of:

- the *protocol-walking* report – which details the full results of all interleaved method invocations (see sections 3.2.2 and 4.1.1);

20

- the *algebra-walking* report – which details a subset of the above, showing only observations on interleaved constructor/mutator sequences (see sections 3.3.1 and 4.1.2); or

- the *state-walking* report – which details exhaustive transition paths explored from each discovered design state (see sections 3.3.3 and 4.1.3).

JWalk version 0.8 was used to generate sample exploration reports for a test class *BoundedStack*, from which excerpts are displayed below. Complete versions of these reports, as well as source code for the tested class, are available on the *JWalk* website (Simons, 2007).

4.1.1 The Protocol-Walking Report

Listing 3 shows excerpts from the test report generated for the "protocol-walking" exploration of a *BoundedStack* class that is derived by inheritance from an *AbstractStack* class. The notion here is to view all interleavings of methods, especially the interaction of local and inherited methods, as a means of validating the possible behaviours of the class. Listing 3 illustrates sequences that interleave inherited methods, such as *isEmpty(),* with locally-defined methods, such as *push()*, in all combinations.

```
Exploring all method protocols of length: 0

BoundedStack target = new BoundedStack();
        ==> BoundedStack#0


Exploring all method protocols of length: 1

BoundedStack target = new BoundedStack();
  target.pop();
        ==> EmptyStackException#0  *exception*

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
        ==> void

BoundedStack target = new BoundedStack();
  target.top();
        ==> EmptyStackException#0  *exception*

BoundedStack target = new BoundedStack();
  target.isFull();
        ==> false

BoundedStack target = new BoundedStack();
  target.size();
        ==> 0

BoundedStack target = new BoundedStack();
  target.isEmpty();
        ==> true
...

BoundedStack target = new BoundedStack();
  target.isEmpty();
  target.isEmpty();
  target.isFull();
        ==> false
```

```
    BoundedStack target = new BoundedStack();
      target.isEmpty();
      target.isEmpty();
      target.size();
            ==> 0

    BoundedStack target = new BoundedStack();
      target.isEmpty();
      target.isEmpty();
      target.isEmpty();
            ==> true


    Test summary for class: BoundedStack

      Test class: BoundedStack
      Test mode: Protocol exploration
      Test depth: 3

      Exercised 111 test sequences that executed normally
      Terminated 28 test sequences that raised an exception
      Discarded 120 test sequences whose prefixes had failed
```

Listing 3. Excerpts from the "Protocol-Walking" Report

All executed *TestSequence*s are presented for human inspection, grouped in order of increasing path depth. The program-style layout and visualisation make the reports familiar and easy to read. Note how the result of a test may be *void*, a value, an object or an exception. Exceptions are flagged, c.f. (Csallner and Smaragdakis, 2004), but the tester must decide whether these indicate faults, or merely broken preconditions. Semantic faults are as likely to be found in the non-exceptional results. The protocol-walking mode is useful for investigating unexpected interactions between methods in the early stages of design.

A test summary is given at the end of the report, detailing how many sequences completed normally or terminated with exceptions. The summary also reports how many other sequences were discarded, due to the prefix having already halted. *JWalk* prunes these dead-ends on-the-fly, but keeps a total tally of how many interleaved sequences might eventually terminate with exceptions.

4.1.2 The Algebra-Walking Report

Listing 4 shows excerpts from the test report generated for the "algebra-walking" exploration performed on the same *BoundedStack* class. The notion here is to focus more closely on constructions that are more likely to reflect the normal use of the CUT. In this mode, *JWalk* builds all algebraic constructions, consisting of interleaved constructor/mutator prefixes, terminating in either an observer or a mutator, whose behaviour is the real point of interest.

```
    ...

    Exploring all algebraic constructions of length: 2

    BoundedStack target = new BoundedStack();
      target.push(Object Object#0);
      target.top();
            ==> Object#0

    BoundedStack target = new BoundedStack();
      target.push(Object Object#0);
```

```
      target.isFull();
            ==> false

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.size();
            ==> 1

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.isEmpty();
            ==> false


Exploring all algebraic constructions of length: 3

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.pop();
            ==> EmptyStackException#0  *exception*

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.top();
            ==> EmptyStackException#0  *exception*

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.isFull();
            ==> false

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.size();
            ==> 0
...

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.size();
            ==> 2

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.isEmpty();
            ==> false


Test summary for class: BoundedStack

  Test class: BoundedStack
  Test mode: Algebraic exploration
  Test depth: 3

  Withheld 6 void results predicted from signatures
```

```
     Exercised 21 test sequences that executed normally
     Terminated 4 test sequences that raised an exception
     Discarded 234 test sequences whose prefixes were pruned
```

Listing 4. Excerpts from the "Algebra-Walking" Report

The "algebra-walking" test report is shorter than the "protocol-walking" test report in two respects. Firstly, it avoids generating sequences containing observer methods in the prefix (see section 3.3.1). Secondly, the report is filtered, so that it only displays the results of *observations* made on sequences. A *void* procedural sequence will not be displayed, unless it raises an exception. "Algebra-walking" mimics a certain style of developer-testing, in which the CUT is modified, then the update is observed. It is most useful during build-and-test cycles, as a first confirmation that the CUT behaves as anticipated, before one of the more rigorous testing modes is chosen.

The test summary indicates, along with the tally of normal and exceptional sequences, how many *void* results were predicted automatically; and finally how many sequences were pruned, whose prefixes either contained observers, or raised exceptions. These are considered non-unique in algebraic mode. This exploratory mode generates the shortest reports, for a given depth of exploration, so is sometimes used to investigate behaviour to a greater depth.

4.1.3   The State-Walking Report

Listing 5 shows excerpts from the test report generated for the "state-walking" exploration, performed on the same *BoundedStack*. The notion here is to discover the CUT's intended abstract design states, in order to identify places where the behaviour of its methods might change in some significant way. For example, the *BoundedStack* will react differently in its *Empty* and *Full* states, from when it is in a *Default* state.

```
     State space of class: BoundedStack
       found state: Empty
       found state: Default
       found state: Full


     Empty state: Exploring all state transitions of length: 0

     BoundedStack target = new BoundedStack();
             ==> BoundedStack#0


     Empty state: Exploring all state transitions of length: 1

     BoundedStack target = new BoundedStack();
       target.pop();
             ==> EmptyStackException#0  *exception*

     BoundedStack target = new BoundedStack();
       target.push(Object Object#0);
             ==> void
     ...

     Default state: Exploring all state transitions of length: 0

     BoundedStack target = new BoundedStack();
       target.push(Object Object#0);
             ==> void
```

```
Default state: Exploring all state transitions of length: 1

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
        ==> void

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
        ==> void

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.top();
        ==> Object#0
...

Full state: Exploring all state transitions of length: 0

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.push(Object Object#2);
  target.push(Object Object#3);
  target.push(Object Object#4);
  target.push(Object Object#5);
  target.push(Object Object#6);
  target.push(Object Object#7);
  target.push(Object Object#8);
  target.push(Object Object#9);
        ==> void


Full state: Exploring all state transitions of length: 1
...

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.push(Object Object#2);
  target.push(Object Object#3);
  target.push(Object Object#4);
  target.push(Object Object#5);
  target.push(Object Object#6);
  target.push(Object Object#7);
  target.push(Object Object#8);
  target.push(Object Object#9);
  target.push(Object Object#10);
        ==> void

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.push(Object Object#2);
  target.push(Object Object#3);
  target.push(Object Object#4);
  target.push(Object Object#5);
  target.push(Object Object#6);
```

```
        target.push(Object Object#7);
        target.push(Object Object#8);
        target.push(Object Object#9);
        target.top();
              ==> Object#9
    ...


    Test summary for class: BoundedStack

        Test class: BoundedStack
        Test mode: State space exploration
        Test depth: 3


        Exercised 601 test sequences that executed normally
        Terminated 44 test sequences that raised an exception
        Discarded 132 test sequences whose prefixes had failed
```

Listing 5. Excerpts from the "State-Walking" Report

There are two interesting aspects to highlight about state discovery. The first is that *JWalk*'s prior exploration of the *BoundedStack*'s state space correctly identifies the three distinct states {*Empty, Default, Full*}, even though these were distributed over two classes. The inherited {*Empty, Default'*} states are correctly partitioned by splitting the *Default'* state into {*Default, Full*}. The second is that *JWalk* finds the shortest sequences to reach the *Empty*, *Default* and *Full* states, bypassing longer constructions that fall into the *Default* category to reach the *Full* state, even though this is at some distance from the others. *JWalk* also reports if any anticipated state is not found, or if no progress is made out of a given state (see section 3.3.2 above).

All executed *TestSequence*s are presented, grouped by starting state and increasing path depth. The state cover is indicated by transition paths of length 0, while longer paths of length 1, 2, 3 above indicate transition sequences of all interleaved methods, explored from each state. This achieves identical coverage to complete state-based testing approaches (Chow, 1978; Holcombe and Ipate, 1998). The resizing behaviour of the *BoundedStack* can be observed in the *Full* state, as well as the exceptional behaviour in the *Empty* state. The "state-walking" mode is useful for confirming that the CUT behaves correctly at the limits of its representation. Design states encode important conditions affecting branches in a CUT's implementation, so are useful for testing the major branches of the code, sometimes achieving branch coverage.

4.2   Bounded Exhaustive Testing

JWalk produces two more kinds of report for the interactive testing modes. In these modes, the tool interacts with the programmer to elicit oracle values, as described above (see section 3.4). These are quickly confirmed or rejected by a keystroke. When an interactive testing cycle is completed, *JWalk* generates one of:

- the *algebra-testing* report – which validates observations on interleaved constructor/mutator sequences, after interactive confirmation (see sections 3.4 and 4.2.1); or

- the *state-testing* report – which validates bounded exhaustive transition paths from each design state, after interactive confirmation (see sections 3.4 and 4.2.2).

JWalk version 0.8 was used to generate sample test reports for the same test class as used above, *BoundedStack*, from which excerpts are displayed below. Complete versions of these reports are available on the *JWalk* website (Simons, 2007).

4.2.1   The Algebra-Testing Report

Listing 6 shows excerpts from the report generated for the "algebra-testing" mode, performed on the same *BoundedStack*. The test sequence generation strategy is exactly the same as for the "algebra-walking" mode (see section 3.3.1), but this time, the results of each test must be confirmed, either manually (see section 3.4.1), or automatically by rule-prediction (see section 3.4.2). The interest here is in building up a test oracle for all the key observations on the CUT for as little effort as possible. Every sequence presented is an observation on a unique constructor/mutator prefix, whose result could be used to predict the outcome of further tests in the state-based testing mode (see section 4.2.2). Some *void*-results are predictable.

```
...
Validating all algebraic constructions of length: 2

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.top();
        ==> Object#0
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.isFull();
        ==> false
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.size();
        ==> 1
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.isEmpty();
        ==> false
Confirm (y|n|q) ? : y


Validating all algebraic constructions of length: 3

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.pop();
        ==> EmptyStackException#0  *exception*
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.top();
        ==> EmptyStackException#0  *exception*
Confirm (y|n|q) ? : y
```

```
BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.isFull();
        ==> false
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.size();
        ==> 0
Confirm (y|n|q) ? : y
...

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.size();
        ==> 2
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.push(Object Object#1);
  target.isEmpty();
        ==> false
Confirm (y|n|q) ? : y

Test summary for class: BoundedStack

  Test class: BoundedStack
  Test mode: Algebraic validation
  Test depth: 3

  Passed 6 test sequences automatically from oracles
  Passed 19 test sequences that were manually confirmed
  Exercised 21 test sequences that executed normally
  Terminated 4 test sequences that raised an exception
  Discarded 234 test sequences whose prefixes were pruned
```

Listing 6. Excerpts from the "Algebra-Testing" Report

In addition to the usual statistics on the number of normal, exceptional and discarded test sequences, the test summary reports how many tests *passed* or *failed*. *Passed* tests are those, whose results were confirmed as correct; while *failed* tests (none in this example) are those, whose results were rejected as incorrect. Results are further split into those that were *manually* confirmed by the tester and those that were *automatically* confirmed, in the current cycle, which gives a measure of test-oracle reuse. Out of the 259 possible sequences, 234 were discarded as non-unique (see above), 19 unique observations were confirmed manually and 6 *void*-results were predicted. In the above example, the test cycle began with *no* prior saved oracle values; and an oracle was constructed to depth 3 in one pass.

Typically, oracles are constructed over several test cycles, of increasing depth, as the design of the CUT gradually stabilizes, in which case the value of oracles increases during retesting. *JWalk* remembers both *positive* and *negative* confirmations, so that both passes and fails may be automatically reconfirmed. Any failed test cases (none arising here) would be listed for inspection

before the test summary. On test completion, the behaviour of the *BoundedStack* has been validated for all algebraic constructions up to length 3. This would be satisfactory coverage of an abstract algebraic specification, under the usual assumption of regularity (Doong and Frankl, 1991; Chen, et al. 1998), but has not yet properly tested the behaviour of the concrete *Full* state, which is a feature of the implementation.

4.2.2 The State-Testing Report

Listing 7 shows excerpts from the report generated for the "state-testing" mode, performed on the same *BoundedStack*. The test sequence generation strategy is exactly the same as for the "state-walking" mode (see section 3.3.3). The results of each test were confirmed, either manually or automatically (see section 3.4). In this example, the "state-test" was carried out immediately following on from the "algebra-test" above (see section 4.1.1), in order to permit *JWalk* to reuse some of the oracle values already collected for the previous test. This eventually leads to a considerable increase in the level of test automation.

```
State space of class: BoundedStack
  found state: Empty
  found state: Default
  found state: Full

Empty state: Validating all state transitions of length: 0

Empty state: Validating all state transitions of length: 1

Empty state: Validating all state transitions of length: 2

Empty state: Validating all state transitions of length: 3

Default state: Validating all state transitions of length: 0

Default state: Validating all state transitions of length: 1

Default state: Validating all state transitions of length: 2

Default state: Validating all state transitions of length: 3

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.push(Object Object#1);
  target.top();
        ==> Object#1
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
  target.pop();
  target.push(Object Object#1);
  target.isFull();
        ==> false
Confirm (y|n|q) ? : y
  ...

Full state: Validating all state transitions of length: 1

BoundedStack target = new BoundedStack();
  target.push(Object Object#0);
```

```
     target.push(Object Object#1);
     target.push(Object Object#2);
     target.push(Object Object#3);
     target.push(Object Object#4);
     target.push(Object Object#5);
     target.push(Object Object#6);
     target.push(Object Object#7);
     target.push(Object Object#8);
     target.push(Object Object#9);
     target.top();
           ==> Object#9
Confirm (y|n|q) ? : y

BoundedStack target = new BoundedStack();
     target.push(Object Object#0);
     target.push(Object Object#1);
     target.push(Object Object#2);
     target.push(Object Object#3);
     target.push(Object Object#4);
     target.push(Object Object#5);
     target.push(Object Object#6);
     target.push(Object Object#7);
     target.push(Object Object#8);
     target.push(Object Object#9);
     target.isFull();
           ==> true
Confirm (y|n|q) ? : y
...

Test summary for class: BoundedStack

  Test class: BoundedStack
  Test mode: State space validation
  Test depth: 3

  Passed 605 test sequences automatically from oracles
  Passed 40 test sequences that were manually confirmed
  Exercised 601 test sequences that executed normally
  Terminated 44 test sequences that raised an exception
  Discarded 132 test sequences whose prefixes had failed
```

Listing 7. Excerpts from the "State-Testing" Report

The early part of this listing contains summary headings, indicating which sets of tests have been automatically checked without the need for user-interaction. These are state-transition paths for which the saved oracle values and predictive rules were sufficient to determine the result. Note that many previously unseen test cases, such as novel sequences that interleave observers in their prefix, are also automatically tested using predictive rules.

It is not until constructions of a previously unseen length are discovered (paths of length 3 in the *Default* state are equivalent to algebraic constructions of length 4, for example) that *JWalk* requests new confirmations. Naturally, all observations on paths leading to the distant *Full* state must be confirmed. Nonetheless, only one test case from each equivalence-class need be confirmed, since the rest are predicted. On test completion, the behaviour of the *BoundedStack* has been validated exhaustively under the assumption that the implementation contains unwanted duplicated paths of lengths less than 3 (Holcombe and Ipate, 1998; Simons, 2006).

The test summary indicates how many results in total were confirmed manually and automatically, in the current cycle. The overwhelming majority of results (605 out of 645) were predicted using the oracle. Test automation achieved more than this, since 777 unique paths were originally proposed, of which 132 were pruned (with halting prefixes) and only 645 were eventually constructed for testing. During this cycle, only 40 confirmations were requested (less than 4 minutes' work). Counting the 19 manual confirmations in the algebraic-testing cycle above, only 59 confirmations were requested, or 9.15% of the total number of executed tests. The complementary level of result prediction was therefore 90.85%; but the level of automated decision-making was 92.41%, when the filtering of pruned sequences is taken into account.

4.3   Efficiency of Oracle Reuse

It is interesting to gather statistics on the percentage of tests confirmed automatically, for different CUTs and for test cycles of different depths, at different stages during the development process. In the following tables, the oracles were grown in left-to-right order, first for "algebra-testing" and then for "state-testing", for paths of depth 1-3, following the recommended develop-and-test cycle. Algebraic testing is always performed first, since this populates the oracle with key values and maximises the opportunity for oracle reuse in state-based testing (see sections 4.2.1, 4.2.2).

In tables 1 and 2, the oracles were built from scratch for each CUT, such that the oracle values elicited during cycles 1..*k* could be reused predictively in cycle *k*. In tables 3 and 4, larger CUTs were constructed by inheritance, respectively extending the classes in tables 1 and 2. These automatically reused the oracles of their parent class as a starting basis; otherwise oracle values were elicited as above, to confirm novel interleaved sequences. The tables mostly illustrate the *incremental* cost of manual confirmation (per test cycle), amortized over the entire development. For comparison, the *cumulative* cost is given in the final column.

Table 1. Incremental Oracle Efficiency for the LibraryBook

LibraryBook Class	Algebra 1	Algebra 2	Algebra 3	State 1	State 2	State 3	Merged
Manual (cycle)	3	5	7	0	0	5	20
Predicted (cycle)	2	8	18	18	38	133	118
Manual (total)	3	8	15	0	0	5	20
Executed (total)	5	13	25	18	38	138	138
Discarded (total)	0	8	60	0	4	32	32
% Manual (cycle)	60.00%	38.46%	28.00%	0.00%	0.00%	3.62%	14.49%
% Predicted (cycle)	40.00%	61.54%	72.00%	100.00%	100.00%	96.38%	85.51%
% Automated (cycle)	40.00%	76.19%	91.76%	100.00%	100.00%	97.06%	88.24%

Table 2. Incremental Oracle Efficiency for the BoundedStack

BoundedStack Class	Algebra 1	Algebra 2	Algebra 3	State 1	State 2	State 3	Merged
Manual (cycle)	6	4	9	4	8	28	59
Predicted (cycle)	1	9	16	17	109	617	586
Manual (total)	6	10	19	4	12	40	59
Executed (total)	7	13	25	21	117	645	645
Discarded (total)	0	30	234	0	12	132	132
% Manual (cycle)	85.71%	30.77%	36.00%	19.05%	6.84%	4.34%	9.15%
% Predicted (cycle)	14.29%	69.23%	64.00%	80.95%	93.16%	95.66%	90.85%
% Automated (cycle)	14.29%	90.70%	96.53%	80.95%	93.80%	96.40%	92.41%

In the tables, the upper rows 1-2 indicate the *per cycle* number of new manual confirmations and predicted results (including all reconfirmed results of lesser depth). The middle rows 3-5 give the *cumulative* number of manual confirmations, executed and pruned sequences, for the given mode and depth. The lower rows 6-8 give the *per cycle* measures of oracle efficiency: the (complementary) ratios of manual and predicted outcomes to *executed* tests and the ratio of automated decisions (including pruning and predictions) made over the *total* number of proposed test sequences. The columns *Algebra 1–3* give the incremental efficiency for algebraic testing, likewise the columns *State 1-3* for the following incremental state-based tests. The final *Merged* column gives adjusted figures for when the testing is conducted in a single block, rather than incrementally. This is as if testing had been delayed until the end and conducted in one algebraic, and one state-based pass, to depth 3.

Table 3. Incremental Oracle Efficiency for the ReservableBook

ReservableBook Class	Algebra 1	Algebra 2	Algebra 3	State 1	State 2	State 3	Merged
Manual (cycle)	3	14	56	0	11	83	167
Predicted (cycle)	6	27	89	36	241	1649	1565
Manual (total)	3	17	73	0	11	94	167
Executed (total)	9	41	145	36	252	1732	1732
Discarded (total)	0	32	440	0	40	608	608
% Manual (cycle)	33.33%	34.15%	38.62%	0.00%	4.37%	4.79%	9.64%
% Predicted (cycle)	66.67%	65.85%	61.38%	100.00%	95.63%	95.21%	90.36%
% Automated (cycle)	66.67%	80.82%	90.43%	100.00%	96.23%	96.45%	92.86%

Table 4. Incremental Oracle Efficiency for the BoundedVector

BoundedVector Class	Algebra 1	Algebra 2	Algebra 3	State 1	State 2	State 3	Merged
Manual (cycle)	3	1	9	1	12	88	114
Predicted (cycle)	6	16	32	26	175	1219	1193
Manual (total)	3	4	13	1	13	101	114
Executed (total)	9	17	41	27	187	1307	1307
Discarded (total)	0	56	544	0	32	608	608
% Manual (cycle)	33.33%	5.88%	21.95%	3.70%	6.42%	6.73%	8.72%
% Predicted (cycle)	66.67%	94.12%	78.05%	96.30%	93.58%	93.27%	91.28%
% Automated (cycle)	66.67%	98.63%	98.46%	96.30%	94.52%	95.40%	94.05%

In table 1, the simple *LibraryBook* had 5 public operations, including a constructor. "Algebra-testing" covered much of the design state space, such that few manual confirmations were requested in "state-testing" mode. In table 2, the *BoundedStack* (described above) had 7 public operations, including a constructor. Further manual confirmations were requested in "state-testing" mode, because of the distant *Full* state that had not been visited before. In table 3, *ReservableBook* had 9 public operations, adding 4 and replacing 2 of *LibraryBook*'s operations, including the constructor. New confirmations were requested for novel observations on interleaved local and inherited methods in "algebra-testing" mode. The increase in confirmations for "state-testing" to depth 3 reflected deeper searches starting in the *OnLoan&Reserved* state. In table 4, *BoundedVector* also had 9 public operations, adding 2 operations to *BoundedStack* and replacing the constructor. The few additional confirmations requested in "algebra-testing" mode reflect the small incremental change. "State-testing" to depth 3 caused the first real increase in confirmations.

What is most significant about all of these cases is the high degree of test automation possible in the lazy systematic approach. The total number of executed test cases typically exceeds the number of

manually confirmed cases by more than an order of magnitude. Even when the number of manual confirmations is high, such as in the *State-3* columns in figures 3 and 4, this is only around eight minutes' work for the tester, which is quite tractable, given the quality of the testing result. The cumulative cost of building an oracle should be compared against the time taken to develop manual tests for *JUnit* (Beck, 2004; JUnit, 2007). *JWalk* can test literally thousands of test cases automatically, for just a few minutes' properly focused effort.

4.4   Detecting Seeded Faults

It is interesting to demonstrate the fault-finding capability of a fully trained test oracle. Further testing experiments were carried out by seeding faults deliberately in the CUT, after an oracle had been trained on the correct version. This was conducted as an interactive workshop, where participants were invited to suggest faults in the kinds of datatype described in the introduction to section 4. The suggested faults ranged from simple counter update failures and indexing errors, to subtle changes in the semantics of the datatype in question. Faults were independently proposed, by participants not involved in creating the original software.

The experimental procedure was to create a reference class, test it exhaustively in both testing modes, then archive the oracle data file. A copy was taken both of the class and the oracle file, for use in fault seeding and mutation testing. The class source code was modified by hand, to introduce semantic faults, and recompiled. When the faulty CUT was retested with *JWalk*, a copy of the fully trained oracle of correct observations was used to predict test outcomes.

4.4.1   Mutant Detection

The detection of mutants was signalled whenever *JWalk* asked the tester to confirm novel results that differed from the saved oracle values. The tester *rejected* each such discrepancy (as incorrect). *JWalk* stored the new incorrect result in the oracle, and used this to predict further test failures. At the end of the test, a report was generated listing all of the failed test cases. Subsequently, the re-trained oracle could be reused to detect all passed and failed test cases automatically. If the CUT was edited to fix the seeded faults, the same oracle could be used again to confirm the absence of faults, automatically. Conversely, if editing introduced further faults, these were signalled as new discrepancies.

Examples of faults introduced in a simple *Stack* class are reported on the *JWalk* website (Simons, 2007). Faults include failure to update the stack counter, failure to reallocate memory, and substitution of the FIFO semantics of a *Queue* for the LIFO semantics of a *Stack*. The counter update fault is detectable for observations made after the first *push* operation. In state-based testing mode, failure to proceed beyond the *Empty* state is also reported. The memory reallocation fault is first detected in state-based mode, after a *push* operation in the *Full* state causes an unexpected out of bounds exception. The substitution of FIFO for LIFO semantics is detected by observations made after two *push* operations.

JWalk found all seeded faults, using test sequences of path length 1-3. Sometimes the visible consequences of a fault were multiple, reported many times over. It is clear that the strong fault-finding ability of *JWalk* comes from the *complete* information stored in the oracle, resulting from the systematic oracle construction method, using bounded exhaustive testing.

4.4.2   Testing Third-Party Software

It is potentially interesting to see how *JWalk* functions when testing third-party software. Work in progress includes unit testing *Java* code developed by others. One useful case study is the *nanoxml* project (De Scheemaecker, 2007) that has been offered for use in software testing experiments (SIR, 2007). SIR provides tools to support a number of different kinds of testing. The complete *nanoxml* project exists in both "clean" and "fault-seeded" versions. A SIR preprocessor program is used to enable or disable the faults seeded in the *nanoxml* java source.

In its default configuration, *JWalk* was initially able to analyse only some of the components in this collection, such as *XMLElement.java*, but failed to analyse other components, which referenced certain *Stream* classes that *JWalk* could not yet synthesize. This was due to the *Streams* in question having no default constructor, as *ObjectGenerator* expects, by default. An obvious workaround was to create a custom *StreamGenerator*, specifically to synthesize controlled *Stream* instances (see section 3.1.2). The *StreamGenerator* was derived from *ObjectGenerator*, and intercepted specific requests for *Stream* objects, delegating other requests to the superclass. Created *Streams* were wrapped around standard text files of the kind anticipated by the application. In *JWalk*, this strategy is commonly adopted when it is desired to create specific kinds of mock, or stub objects. After this action, other classes from the *nanoxml* library were analysed successfully.

5.   Improvements over Previous Approaches

The aim of the current work is to provide the agile development community with a rigorous, rapid-turnaround testing method, equivalent in strength to *complete* functional testing methods. These use a formal specification to drive the selection of test cases in a deterministic way, and validate the tested implementation exhaustively, under some weakening assumptions about regularity (see section 5.3 below). Agile methods are mostly opposed to writing formal specifications, partly because of the extra overhead involved, partly because of the risk of the software evolving independently and partly because of the perceived mathematical difficulty.

While some might find it daunting to construct the kind of recurrence relations found in an algebraic specification (Goguen, et al., 1993; Goguen and Malcolm, 1997), most programmers are capable of detecting when an instance of a property is violated. This leads to the idea of confirming oracle values incrementally in the lazy systematic testing approach, whereby the specification is acquired gradually, and always evolves in step with the code. The systematic sequence generation strategy mimics the exhaustive path generation found in specification-based methods, proposing difficult interleaved test cases that programmers frequently fail to notice. Some recurrence relations are constructed automatically using simple predictive rules, which greatly reduce the number of cases for manual confirmation.

5.1   More than Fault-Finding Approaches

JWalk bears only superficial similarity with previous fault-finding approaches. Like *JCrasher* (Csallner and Smaragdakis, 2004) and *JTest* (Parasoft, 2007), *JWalk* exploits *Java*'s *Reflection API* to extract the signatures of the CUT's public constructors and methods, prior to constructing test sequences. Like tools derived from *Daikon* (Ernst, et al., 2001; 2007), such as *DSD-Crasher* (Csallner and Smaragdakis, 2006a) and *Agitator* (Boshernitsan, et al., 2006), *JWalk* seeks to infer recurrent properties of the tested software and re-apply these in further testing. But apart from these similarities, the tools behave in completely different ways.

5.1.1 Comparison with *JCrasher*

JCrasher uses the method signatures to determine the size of their input parameter space, and thereafter constructs unit tests with random input data chosen from within and around the boundaries of this space, seeking to trigger exceptions. Test sequences are randomly generated, such that no guarantee can be made about protocol or algebraic coverage. The main benefit of the tool is to draw attention to raised exceptions. Heuristics must be used to determine whether the exceptions are merely due to violated preconditions, or whether they signify genuine software faults (Csallner and Smaragdakis, 2004).

By contrast, *JWalk* achieves deterministic coverage of the CUT's method protocols, algebraic constructions and abstract state-transition graph. (The current version does not aim to perform complete category-partition testing on input parameters; this is a work in progress). *JWalk* learns the difference between violated preconditions and genuine faults; likewise it learns the difference between semantically correct and incorrect results (of the non-exceptional kind) from hints supplied by the human tester, so eventually is much better at discriminating between a correct and incorrect result than *JCrasher*.

5.1.2 Comparison with *Daikon*

Daikon monitors all kinds of input, local and loop parameters to infer simple program invariants. These may include equality, inequality and linear relationships, ranges, ordering, containment and sortedness, among other properties (Ernst, et al., 2007). Regular observations are converted into hypotheses, which the human tester may promote to assertions in *Agitator* (Boshernitsan, et al., 2006), for use in conformance testing. Similar combinations of specification inference and testing have been applied in *DSD-Crasher* (Csallner and Smaragdakis, 2006a) to improve the quality of static analysis; and in *Jov* (Xie and Notkin, 2003) and *Eclat* (Pacheco and Ernst, 2005), to filter test-sets to find more fault-revealing inputs (although *Jov* tended to discover inputs that violated preconditions).

The *Daikon* learning strategy is sophisticated, yet the results vary in usefulness. The learned invariants may correspond to key semantic properties of the software, or alternatively, may correspond to derived properties, or even irrelevant properties that are not especially intended (Agitar Software, 2007). By contrast, the oracle values learned by *JWalk* are always key properties of the software, irreducible observations that are used in further result predictions. In "algebra-testing" (see section 4.2.1), the strategy is to populate the oracle systematically, using the algebraic structure of the CUT to drive the elicitation process. As a result, the coverage of learned atomic properties is exhaustive, up to the limit of exploration, and is later found capable of detecting all illegal software mutations (see section 4.4.1).

5.2 More than State-Filtering Approaches

JWalk bears comparison with other attempts to abstract over the state-space of the CUT, for example (Xie, et al., 2004; 2005), but explores the sate-space more efficiently, with dynamic feedback to prune unwanted paths, rather than using state filters on randomly-generated paths. *JWalk* is also quite different in the way that it detects its most abstract design states from other approaches that use predicates (Grieskamp, et al., 2002; Boyapati, et al., 2002). Eventually, *JWalk* requires far less intrusive design-for-test requirements.

5.2.1 Comparison with Rostra and Symstra

Rostra (Xie, et al., 2004) proposes up to five different abstractions over object state, from the most detailed "whole variable state" (the attribute product space), via the "whole method sequence" (c.f. *JWalk*'s *all method protocols*) and "modifying method sequence" (c.f. *JWalk*'s *all algebraic constructions*), to the "pairwise equals" and "monitor equals" functions (external oracles over two object states, the second of which observes a subset of attributes). These increasingly abstract filters over state are applied to randomly generated test sequences, in order to group these into equivalence-classes, with the aim of reducing the size of the retained test-set, by selecting exemplars from each equivalence-class.

Rostra's "whole variable state" offers no abstraction, so is not useful as a filter. *JWalk*'s method sequence abstractions over state are similar to *Rostra*'s, with the proviso that *JWalk* detects *observer/mutator* distinctions empirically, and not from signatures alone. *JWalk*'s detection of design states is entirely different from *Rostra*'s oracle functions (see section 5.2.2). Apart from this, *JWalk* is not random in its exploration, does not adopt a generate-and-test filtering approach to select its test sequences (which is expensive, because of the initial over-generation of sequences) and even uses dynamic feedback to prune the size of the next test-cycle, removing test sequences with failed prefixes, and non-discriminating sequences with observer-prefixes. Parallel work on dynamic pruning of sequences with failed prefixes has also recently been reported for the *Randoop* tool (Pacheco, et al., 2007).

JWalk shares the symbolic execution goals of *Symstra* (Xie, et al., 2005), which seeks to abstract over classes of inputs to a binary tree sorting algorithm, by using a smaller set of *symbolic values* that exercise each branching function in a deterministic way. *JWalk* may use custom *Generators* to create symbolic objects, under the control of the tester, in a similar way to mock-objects, or stubs.

5.2.2 Comparison with *Korat* and *Java Pathfinder*

JWalk's more subtle use of natural internal predicates to infer abstract design states improves on the use of external functions, widely used as state observers elsewhere (Grieskamp, et al., 2002; Xie, et al., 2004). Other approaches insist on an exhaustive set of internal state predicates, as a design-for-test requirement, e.g. *Catalysis* (D'Souza and Wills, 1998). These approaches are considered overly intrusive: the former requires an external specification of states, which might not be supported internally by the CUT, whereas the latter imposes more restrictive design-for-test requirements on the CUT. *JWalk*'s *Boolean* product algorithm is far less intrusive, and always makes the best use of whatever state information is supplied.

Although *JWalk* uses predicates, it is not related to some other predicate-based approaches, such as *Korat* (Boyapati, et al., 2002), which generates executable *Java* predicates from formal specifications. Preconditions are converted into test input filters, and postconditions into test oracles. The specifications supplied in *Korat* may be compared with other JML-based annotation and testing methods (Cheon and Leavens, 2002), where a partial algebraic specification is constructed in the JML annotation language, prior to generating tests from the specification. By contrast, *JWalk* requires no external formal specification, relies only on predicates that naturally form part of the CUT's interface, and uses these in a different way to detect significant design states.

JWalk's ability to find recurrent states, by partial order reduction on test sequences, may be compared superficially with a similar approach used by *Java Pathfinder* (Vissner, et al., 2003; 2006). The latter works directly on bytecode traces in the *Java Virtual Machine*, merging similar

traces to match fine-grained states (see section 2.2), whereas *JWalk* works at a higher level on *observer/mutator* method sequences, to discover regularities in algebraic constructions.

5.3   Convergence with *Complete* State-based Testing

The most important property of lazy systematic unit testing is its convergence with *complete* state-based testing (Chow, 1978; Holcombe and Ipate, 1998; Simons, 2006). The notion of convergence refers to J*Walk*'s gradual acquisition of the specification required to perform testing to this high standard. Complete state-based testing offers stronger guarantees of correctness after testing than other approaches, due to the progressively relaxed assumptions about the regularity of the implementation.

5.3.1   State-based Testing for Objects

Early state-based testing methods include (Chow, 1978; Jard and von Bochmann, 1983), which proposed generating test sequences from simple finite state automata, to test communications protocols and finite-state devices. McGregor (1994) exploited state-based testing for object-oriented software, using incrementally-derived state machines to model the partitioning of states in subclasses. A complete formal treatment of subtyping and state partitioning, and their relationship to test coverage and process algebra, may be found in (Simons, 2006). Bounded exhaustive unit testing from state-based specifications is tractable (McGregor, 1994), but synthesizing the state-space of entire systems from object state machines produces a state explosion (Binder, 1996) unless a suitable formal strategy is found for partitioning the tests (Holcombe and Ipate, 1998; Ipate and Holcombe, 1997).

5.3.2   Complete State-based Testing

The complete state-based testing approach is based on (Chow, 1978), which relates a minimal state specification to a possibly non-minimal implementation, with redundant states and transitions. This was taken up by Holcombe and Ipate (1998) in their *X-Machine* testing method, which replaced simple input/output transitions by arbitrary processing functions, which are subsequently decomposed into sub-machines. The method uses a divide-and-conquer approach that tests the components of a system exhaustively and then proves the correct integration of these by formal verification (Ipate and Holcombe, 1997).

When testing an individual unit, *completeness* refers to the ability to achieve full positive and negative testing; and to relax assumptions about the regularity of the implementation, while still proving conformance to the specification. The testing method drives the unit into all of its design states (using the *state cover*) and then constructs progressively longer transition sequences to exercise the unit, starting in each state. Full positive and negative testing is achieved, confirming the presence of all desired paths and the absence of all undesired paths, by algorithmically attempting all possible interleaved transitions. Whereas other testing methods may stop after sequences of length 1 or 2 (the *transition cover*, or *switch cover*), longer sequences are explored. This is to account for passing through possibly longer chains of redundant, duplicated states in a poorly-coded unit, which might at the limit exhibit faulty behaviour.

After testing is over, the unit is guaranteed to be correct, up to the assumption that it contains duplicated paths of length less than k, the upper bound on test generation. In practice, relatively small values of $k \leq 5$ account for even unreasonably tangled implementations. Systems have been developed to very high levels of confidence; for example, the *Stamp Dealer Trading System* reported in (Holcombe and Ipate, 1998) has functioned without faults since its delivery. Further

examples of the effectiveness of test coverage are given in (Holcombe, 2003; Holcombe, et al., 2001).

5.3.3   *JWalk* Compared to *X-Machine* Testing

It is possible to contrast *JWalk*'s "algebra-testing" mode with the formal testing methodology adopted in algebraic testing (Bernot et al., 1991; Doong and Frankl, 1991; 1994; Chen, et al., 1998; 2001). As oracle values are confirmed, more and more of the algebraic structure of the CUT is validated. However, *JWalk* does not yet synthesize canonical test exemplars in the same way as *TACCLE* (Chen, et al., 1998; 2001), since this explores all method arguments recursively as algebraic types in the same way. On the other hand, *JWalk* can test for properties, such as the absence of unwanted side-effects, that a functional algebra cannot even predict. *JWalk* is not tied to the *regularity assumption* that recurrence in the algebra's axioms necessarily corresponds to recurrence in the implementation.

JWalk is capable, in its "state-based" testing mode, of identifying redundant states that exhibit equivalent algebraic properties, but encode implementation differences (e.g. the *Full* and *Default* states of a *BoundedStack*). *JWalk*'s state discovery procedure (see section 3.3.2) identifies the *state cover* necessary to begin state-based testing. The states are discoverable, because the CUT's designer expects the state predicates to be satisfiable within a reasonable space, similar to the *test-completeness* requirement in the X-Machine method (Holcombe and Ipate, 1998). The reached states are identified by direct observation (Simons, 2006), rather than by applying further characterisation sequences (Chow, 1978).

It is fairly easy to see how testing all transition paths of increasing depth, starting from each state, corresponds exactly to the exhaustive *X-Machine* testing method. *JWalk* performs full positive and negative testing, attempting all paths, including paths that should be ignored (nullops), or refused (illegal actions). The generation of longer *JWalk* transition sequences corresponds to the higher values of *k* in the X-Machine method, which test redundant implementations. *JWalk* makes precise observations on all outputs, similar to methods that supplement state-space validation with precise variable observations (Petrenko, et al., 2004). In this way, *JWalk* obtains the best of both worlds, but completely satisfies the unit testing assumptions of *X-Machine* testing.

6.   Conclusions

JWalk has been exercised both with simple classes and more complex library classes in *Java* (Simons, 2007). Further recent work has compared tests generated with *JWalk* against manual test-sets created by an experienced user of *JUnit* (Simons and Thomson, 2007). Work in progress involves using *JWalk* to test third-party software, in particular a fault-seeded version of the *nanoxml* project (De Scheemaecker, 2007; SIR, 2007). Another current project aims to extend test oracle value generation to follow a category-partition style, c.f. (Irvine and Offutt, 1995).

6.1   Integrating *JWalk* with Other Tools

The current stable version is *JWalk 0.8* (Simons, 2007), which includes the modular toolkit. Interested readers are invited to explore the possibility of integrating *JWalk* with their preferred third-party software environment. As an example of this, *JWalk* has recently been integrated successfully as a plugin component for the IBM open source *Eclipse IDE* software environment (Eclipse Foundation, 2007). Apart from the pleasant graphical user interface, this brings certain benefits of synergy, in that testing with *JWalk* can be linked to editing sessions in the *Eclipse IDE*. Editing and testing sessions are interwoven (see figure 1, above). Where the tested class raises a

Java exception, it is possible to trace back to the point in the source code where this was raised, placing the relevant file in the editor.

The main integration issues are at the top level, where the third-party system interfaces to the *JWalk* unit testing framework via the APIs in *JWalker*, a component whose purpose is to accept the test parameters from third-party input sources, and *JTalker*, a component whose purpose is to establish call-backs to display user prompts and test results via third-party display facilities (see figure 1). Internally, all interactive queries and all publishing of test results are carried out by Java's event handling mechanism. The third-party system must register suitable components that satisfy the *QuestionListener* and *ReportListener* interfaces. These respectively accept *QuestionEvent* and *ReportEvent* events, which encapsulate the different kinds of communication to and from the environment (Simons, 2007).

6.2   Future Extensions to *JWalk*

Current work on *JWalk* is directed toward extending the power of the *Generator* hierarchy. At the moment, the deterministic output of the generators is sufficient to achieve full coverage of method protocols, algebraic constructions and design state spaces (of the CUT in question). Further extensions are required to generate key data points in the input space of method arguments automatically, such as would be required in category-partition testing. The first step involves a heuristic determination of partitions in the value-space of each basic type. The second step involves a prior automated exploration of the structure of other objects, when these are used as arguments to methods.

For the moment, custom generators are used wherever particular input argument properties are desired. These are also used to create instances of classes that have particular construction requirements that cannot be anticipated by the tool. Future work will attempt to generalise the argument generation approach to synthesise inputs from classes with arbitrary construction requirements.

The measures taken so far to reduce the size of the search space when generating all algebraic constructions permits the testing of CUTs with 5-15 public operations, to a depth of 3-5. Larger CUTs can realistically be tested only to shallower depths. A far greater pruning of this state space, with a concomitant reduction in the number of requested manual confirmations, may in future be made possible by tracking all revisited concrete attribute states in a different way. This would permit a strong prediction rule to eliminate algebraic *transformer* prefixes, in much the same way as the current rule to eliminate *observer* prefixes.

6.3   *JWalk* Challenges *JUnit*

The testing tool most widely used in the agile software development community is *JUnit* (Beck, 2004; JUnit, 2007), which captures manual test suites and re-executes them on demand. *JWalk* is able to challenge *JUnit* on several fronts, including the completeness of testing, the productive use of automation in testing, the optimal deployment of human expertise in testing and the selective adaptation of test-sets to modified and extended objects.

6.3.1   Human Effort and the Use of Automation in Testing

The main benefit of a tool such as *JUnit* is that it automates the repeated execution of handwritten tests. For this reason, it has become a mainstay in object-oriented regression testing. Yet, consider how *JUnit* does not make the best use of automation in testing. The human effort involved in

thinking up suitable test-cases is considerable and is not guaranteed to be effective. By contrast, *JWalk* proposes suitable test-cases automatically, each of which are guaranteed to exercise a unique property, or make a unique observation. Furthermore, *JWalk* will systematically detect all the difficult and commonly forgotten cases. In related work (Simons and Thomson, 2007), *JWalk* was used to test up to *two orders of magnitude* more test cases than the best suites developed by an experienced *JUnit* tester, in the same limited time period.

JWalk gains by having fewer intellectual overheads than *JUnit*. The human effort goes into reviewing the key test results, rather than having to identify these important cases and then create the tests. No time is wasted writing non-revenue earning code to inject suitable oracle values, because these are generated by the production code itself, and are accepted (or rejected) by a keystroke. With *JWalk*, automation is used to discover the important test cases and the human tester merely has to be alert and patient to build powerful oracles. For each large increase in exhaustive, automatic validation, the human input is still proportionately small (see section 4.3).

6.3.2 Test Completeness in the Presence of Subclassing

In previous work, formal weaknesses were found in the practice of reusing *JUnit* tests as regression test suites (Simons, 2005). Not only are manual tests incomplete and possibly redundant, but they also fail to exercise the state-space of the CUT as testers intuitively expect. This is due formally to the partitioning of states, brought about by modifications and extensions to objects. Rather than covering the same state-space as in the original object, the saved tests cover a geometrically decreasing fraction of the same state-space in the modified or extended object.

Because of this loss of coverage, a different state-based approach to retesting called *test regeneration* was proposed (Simons, 2006). This re-created the test sets from scratch, according to the coverage criteria described above (see section 5.3.2), from the evolving state-based specification. *JWalk* follows this test regeneration approach, since the tests evolve in step with the code. All novel interleavings of old and new methods are exercised, reusing existing test results only where these are still valid. *JWalk* therefore provides *repeatable* guarantees of confidence, after retesting up to the chosen path depth, instead of the progressively weaker guarantees offered by regression testing (Simons, 2006).

Acknowledgement

References

Agitar Software. 2007. Agitator. http://www.agitar.com/products/20040518-agitator.html. Accessed 12 February.

Ball, T. and Larus, J. R. 2000. Using paths to measure, explain and enhance program behaviour. IEEE Computer, 33 (7): 57-65.

Beck, K. 2000. Extreme Programming Explained: Embrace Change, 1st edn. New York: Addison-Wesley.

Beck, K. 2004. The JUnit Pocket Guide, 1st edn. Beijing: O'Reilly.

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

Bernot, B. Gaudel M.-C. and Marre, B. 1991. Software testing based on formal specifications: a theory and a tool. Softw. Eng. J., 6 (6): 387-405.

Bezier, B. 1990. Software Testing Techniques. International Thomson Computer Press.

Binder, R. V. 1996. Testing object-oriented software: a survey. Softw. Test., Verif. Reliab., 6 (3/4), 125-252.

Binder, R. V. 2006. TOOTSIE, a high-end OO development environment. http://www.rbsc.com/pages/tootsie.html. Accessed 28 April.

Boshernitsan, M., Doong, R. and Savoia, A. 2006. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. Proc. 5th ACM Sigsoft Int. Symp. on Softw. Testing and Analysis, Portland Maine, pp. 169-180.

Boyapati, C., Khurshid, S. and Marinov, D. 2002. Korat: Automated testing based on Java predicates. Proc. ACM Sigsoft 3rd Int. Symp. on Softw. Test. and Analysis (ISSTA '02), Rome, Italy, pp. 123-133.

Buy, U., Orso, A. and Pezzè, M. 2000. Automated testing of classes. Proc. 2nd ACM Sigsoft Int. Symp. on Softw. Testing and Analysis, Portland, Oregon, pp. 39-48.

Chen, H. Y., Tse, T. H., Chan F. T. and Chen, T. Y. 1998. In black and white: an integrated approach to class-level testing of object-oriented programs. ACM Trans. Softw. Eng. Meth., 7 (3): 250-295.

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

Cheon, Y. and Leavens, G. T. 2002. A simple and practical approach to unit testing: the JML and JUnit way. Proc. 16th European Conf. Obj.-Oriented Progr., LNCS 2374, Springer Verlag, Malaga, Spain, pp. 231-255.

Chow, T. 1978. Testing software design modeled by finite state machines. IEEE Trans. Softw. Eng., 4(3): 178-187.

Csallner, C. and Smaragdakis, Y. 2004. JCrasher: An automatic robustness tester for Java. Software – Practice and Experience, 34 (11): 1025-1050.

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

Csallner, C. and Smaragdakis, Y. 2006b. Dynamically discovering likely interface specifications, Proc. 28th Int. Conf. on Softw. Eng. (ICSE '06), Shanghai, pp. 861-864.

De Scheemaecker, M. 2007. NanoXML 2.2.1. Sourceforge. http://nanoxml.sourceforge.net/orig/. Accessed 23rd February.

Doong, R. K. and Frankl, P. 1994. The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Meth., 3 (4): 101-130.

Doong, R. K. and Frankl, P. 1991. Case studies on testing object-oriented programs. Proc. 4th Symp. Softw. Testing, Analysis and Verif., ACM Press, pp. 165-177.

Dranidis, D., Tigka, K. and Kefalas, P. 2004. Formal modelling of use cases with X-machines. Proc. 1st South East European Workshop on Formal Methods, Thessaloniki, Greece, pp. 72-83.

D'Souza D. F. and Wills, A. C. 1999. Objects, Components and Frameworks with UML: the Catalysis Approach, Reading MA: Addison Wesley Longman.

Eclipse Foundation. 2007. Eclipse: an open development platform. http://www.eclipse.org/. Accessed 14 February.

Ernst, M. D. 2000. Dynamically discovering likely program invariants. PhD Thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington.

Ernst, M. D., Cockrell, J., Griswold, W. G. and Notkin, D. 2001. Dynamically discovering likely program invariants to support program evolution, IEEE Trans. Softw. Eng., 27 (2): 99-123.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S. and Xiao, C. 2007. The Daikon system for dynamic detection of likely invariants, Science of Computer Programming. *In press*.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Massachusetts: Addison-Wesley.

Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K. and Jouannaud, J.-P. 1993. Introducing OBJ. Technical Report, Oxford Programming Research Group and SRI International, Menlo Park, CA, 90425.

Goguen, J. and Malcolm, G. 1997. Algebraic Semantics of Imperative Programs. Cambridge MA: MIT Press.

Grieskamp, W., Gurevich, Y., Schulte, W. and Veanes, M. 2002. Generating finite state machines from abstract state machines. Proc 3rd ACM Sigsoft Int. Symp. on Softw. Testing and Analysis (ISSTA '02), Rome, Italy, 112-122.

Henkel, J. and Diwan, A. 2003. Discovering algebraic specifications from Java classes. Proc. 17th. European Conf. Obj.-Oriented Progr., Darmstadt, Germany, LNCS 2743, Springer Verlag, 431-456.

Henkel, J. and Diwan, A. 2004. A tool for writing and debugging algebraic specifications. Proc. 26th Int. Conf. Softw. Eng., IEEE Computer Society, 449-458.

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

Holcombe, M., Bogdanov, K. and Gheorghe, M. 2001. Functional test generation for eXtreme Programming. Proc. 2nd Int. Conf. on Extreme Progr. and Flexible Proc. in Softw. Eng., Sardinia, Italy, pp. 109-113.

Holcombe, W. M. L. and Ipate, F. 1998. Correct Systems: Building a Business Process Solution. Applied Computing Series. Berlin: Springer Verlag.

Ipate, F. and Holcombe, W. M. L. 1997. An integration testing method that is proved to find all faults. Int. J. Comp. Math., 63: 159-178.

IPL (Information Processing, Ltd., UK) 2007. Cantata++ for testing C, C++ and Java. http://www.ipl.com/. Accessed 12 February, 2007.

Irvine, A. and Offutt, A. 1995. The effectiveness of category partition testing of object-oriented software. ISSE Department, George Mason University, Fairfax VA, 22030.

Jard, C. and von Bochmann G. 1983. An approach to testing specifications. Journal of Systems and Softare, 3 (4): 315-323.

JUnit. 2007. The JUnit project website. http://www.junit.org/. Accessed 12 February, 2007.

Kiczales, G., des Rivieres, J. and Bobrow, D. G. 1991. The Art of the Metaobject Protocol, Cambridge MA: MIT Press.

Lerda, F. and Visser, W. 2001. Addressing dynamic issues of program model checking, Proc. 8th Int. SPIN Workshop (SPIN '01), Toronto, pp. 80-102.

Lorenzoli, D., Mariani, L. and Pezzè, M. 2006. Inferring state-based behaviour models, Workshop on Dynamic Analysis (WODA '06), Shanghai, China, pp. 25-32.

Marinov, D. and Khurshid, S. 2001. TestEra: A novel framework for testing Java programs. Proc. 16th IEEE Conf. Automated Softw. Eng. (ASE '01), San Diego, California, pp. 22-31.

McGregor, J. D. 1994. Constructing functional test cases using incrementally-derived state machines. Proc. 11th Int. Conf. on Testing Computer Software, USPDI, Washington.

Pacheco, C. and Ernst, M. D. 2005. Eclat: Automatic generation and classification of test inputs. Proc. 19th European Conf. Obj.-Oriented Prog., pp. 504-527.

Pacheco, C., Lahiri, S. K., Ernst, M. D. and Ball, T. 2007. Feedback-directed random test generation. Proc. 29th Int. Conf. Softw. Eng., Minneapolis, MN, USA, IEE Computer Society, 75-84.

Parasoft. 2007. Parasoft Jtest ® product description, http://www.parasoft.com/, Parasoft, Monrovia CA, Accessed 22 August 2007.

Perkins, J. H. and Ernst, M. D. 2004. Efficient incremental algorithms for dynamic detection of likely invariants. Proc. ACM Sigsoft 12th Symp. Found. Softw. Eng. (FSE '04), Newport, California, pp. 23-32.

Petrenko, A., Boroday, S. and Groz, R. 2004. Confirming configurations in EFSM testing. IEEE Trans. Softw. Eng., 30 (1): 29-42.

Robillard, P. N. and Kruchten, P. 2002. Software Engineering Process with the UPEDU. Addison-Wesley.

Simons, A. J. H. 2005. 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 Verlag, Sheffield: 118-126.

Simons, A. J. H. 2006. A theory of regression testing for behaviourally compatible object types. Softw. Testing, Verif. and Reliability, 16 (3): 133-156.

Simons, A. J. H. 2007. JWalk: Lazy systematic class unit testing. http://www.dcs.shef.ac.uk/~ajhs/jwalk/. Accessed 12 February 2007.

Simons, A. J. H. and Thomson, C. D. 2007. Lazy systematic testing: JWalk versus JUnit. Proc 2nd Testing in Academia and Industry Conference – Practice and Research Techniques (TaicPart '07), Windsor Great Park, London, IEEE Computer Society, 138.

SIR. 2007. Software-artifact Infrastructure Repository: the *nanoxml* project. http://sir.unl.edu/content/sir.html. Accessed 22 February.

Tillmann, N. and Schulte, W. 2005a. Parameterized unit tests. Proc. 5th European Softw. Eng. Conf. and ACM Sigsoft Symp. on Found. Softw. Eng. (ESEC/FSE '05), pp. 253-262.

Tillmann, N. and Schulte, W. 2005b. Parameterized unit tests with Unit Meister. Proc. 5th European Softw. Eng. Conf. and ACM Sigsoft Symp. on Found. Softw. Eng. (ESEC/FSE '05), pp. 241-244.

Ural, H., Saleh, K. and Williams, A. W. 2000. Test generation based on control and data dependencies within system specifications in SDL. Computer Communications, 23 (7): 609-627.

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

Visser, W., Pasareaunu, C. S. and Pelánek, R. 2006. Test input generation for Java containers using state matching. Proc. 5th ACM Sigsoft Int. Symp. Softw. Testing and Analysis (ISSTA '06), Portland Maine, pp. 37-48.

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

Xie, T., Marinov, D., Schulte, W. and Notkin, D. 2005. Symstra: A framework for generating object-oriented unit tests using symbolic execution. Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS '05), Edinburgh, pp. 365-381.

Xie, T. and Notkin, D. 2003. Tool-assisted unit test selection based on operational violations. Proc. 18th IEEE Int. Conf. Automated Softw. Eng. (ASE '03), Montreal Canada, pp. 40-48.

Yuan, H. and Xie, T. 2005. Automatic extraction of abstract-object-state machines based on branch coverage. Proc. 1st Int. Workshop on Reverse Engineering to Requirements (RETR '05), Pittsburgh Pennsylvania, pp. 5-11.