Feedback-Based Specification, Coding and Testing with *JWalk*

Anthony J H Simons, Neil Griffiths and Christopher Thomson
Department of Computer Science, University of Sheffield
{a.simons, c.thomson}@dcs.shef.ac.uk, aca05ng@shef.ac.uk

Abstract

JWalk is a lazy systematic unit-testing tool for Java, which supports dynamic inference of specifications from code and systematic testing from the acquired specification. This paper describes the feedback-based development methodology that is possible using the JWalkEditor, an original Java-sensitive editor and compiler coupled to JWalk, which helps programmers to prototype Java class designs, generating novel test cases as they code. Systematic exploratory testing alerts the programmer to unusual consequences in the design; and confirmed test results become part of the evolving specification, which adapts continuously to modified classes and extends to subclasses. The cycle of coding, inferring and testing systematically exposes test cases that are often missed in other test-driven development approaches, which rely on programmer intuition to create test cases.

1. Lazy systematic unit testing

Lazy systematic unit testing is a software testing method based on the two notions of *lazy specification*, the ability to infer the evolving specification of a unit on-the-fly by dynamic analysis, and *systematic testing*, the ability to explore and test the unit's state space exhaustively to bounded depths [1]. *Lazy specification* is a term coined by analogy with lazy evaluation in functional programming and refers to a flexible approach to software specification, in which the specification evolves rapidly in parallel with frequently modified code. The specification is inferred by a semi-automatic analysis of a prototype software unit. This can include static analysis (of the unit's interface) and dynamic analysis (of its behaviour), supplemented by limited interaction with the programmer. *Systematic testing* refers to a complete, conformance testing approach, in which the tested unit is shown to conform exhaustively to a specification, up to the testing assumptions [2]. This contrasts with exploratory, random or other incomplete forms of testing. The aim of systematic testing is to provide guarantees of correctness, once testing is over.

JWalk is a unit-testing tool supporting the lazy systematic unit testing of compiled classes in *Java* [3]. It is provided both as a command line utility, and as an API toolkit for integration with other third-party software development tools. It has been integrated experimentally [1] as a plug-in for the IBM Eclipse 3.0 SDK platform [4] and is currently being trialed by *Java* programming groups at IBM (Hursley) and Accenture (Washington DC), among others [3].

2. The *JWalkEditor* tool

The current work describes a bespoke integration of *JWalk* with a *Java*-sensitive editor, also developed in the *Java* programming language [5]. The *JWalkEditor* was designed with novice programmers in mind, to support the interactive exploration and testing of class APIs as these were being developed. Similar to other editors like *jEdit* [6] and *Eclipse* [4, 7] the *JWalkEditor* offers *Java*-sensitive text highlighting and syntax checking. The *Java* compiler may be invoked from the tool, tracing any compile-time faults back to errors located in the source file. Multiple classes may be developed (in separate tabbed panes) and executed as a system within the same *Java* runtime environment.

In addition, the *JWalkEditor* can exercise the public methods of *any* component class, as a means of validating or testing this unit, at any stage of coding, whether or not the class is finished. Test sequences, consisting of constructors, followed by progressively longer chains of methods, are generated and executed, in a way that systematically explores the test-class's API. The *JWalkEditor* provides a sidebar panel for setting the test parameters, such as the *test mode* (see below) and the maximum *test depth* (sequence length), and a button on the main toolbar initiates unit testing. Depending on the test mode selected, the tool may either help the programmer to validate the test-class's

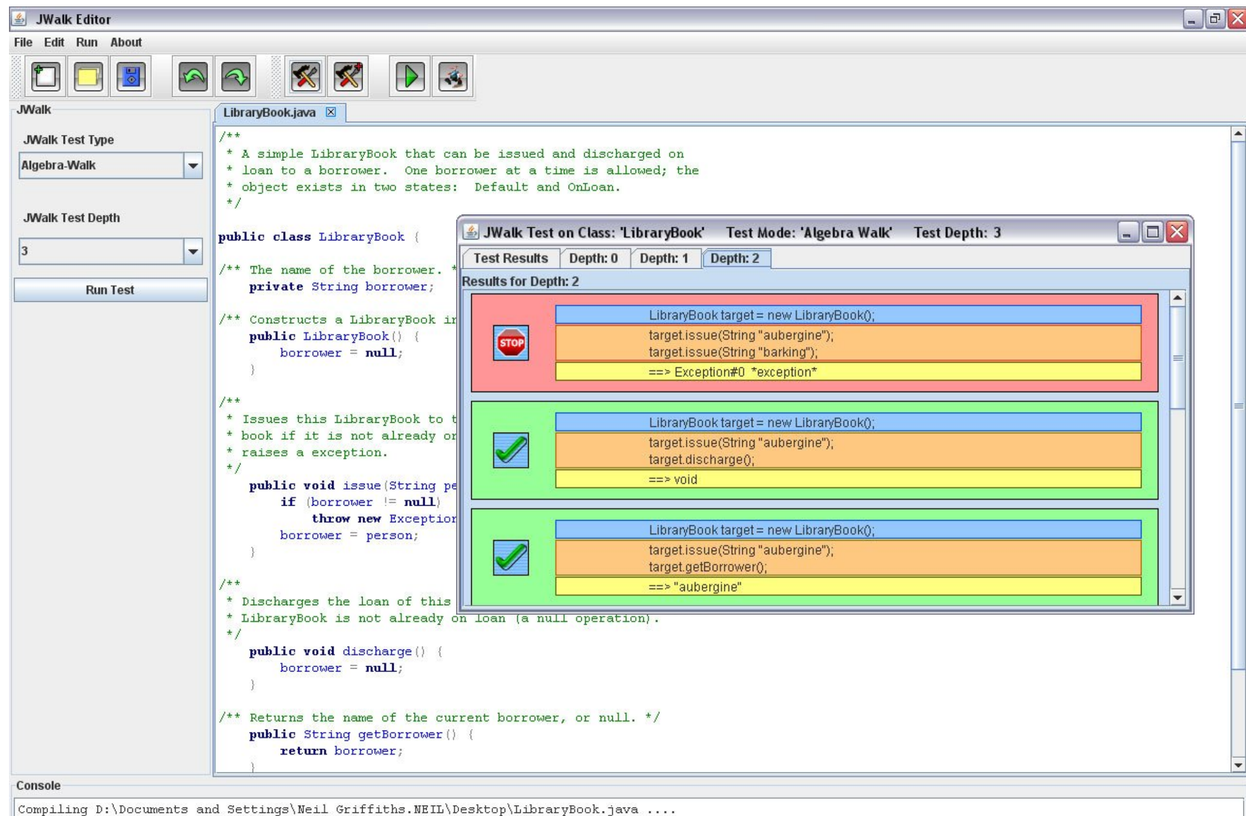

Figure 1. JWalkEditor, exploring the API of a LibraryBook during the validation phase

observable behaviour, by presenting the results of exploratory sequences for inspection, or formally test the class's behaviour with respect to a test oracle, which is created according to the lazy specification method described above.

During validation, the tool may explore *all method protocols* (all interleaved orderings of methods), *all algebraic constructions* (all interleaved state-modifying methods, followed by every observer-method) or *all design states and transitions* (the switch-1 … switch-*n* cover). This can be viewed as exploring the test-class according to different models of state abstraction. The first mode can be compared roughly with *JCrasher* [8] and *Rostra*'s [9] method-states and the second mode with *Rostra's* modifier-states (except that *JWalk* is not random, but deterministic in its selection of arguments, and detects state-modification empirically, rather than by signature analysis). The design state mode is original to *JWalk* and utilizes the Cartesian product of state predicate observations as indicators of qualitative states [1]. The results of exploring the test-class are presented to the programmer for validation, as sets of observations, organized by sequence length, in a window containing a tabbed pane for each set (fig. 1).

During formal testing (as opposed to exploratory validation), the tool verifies the outcomes of the same tests semi-automatically against predictions made by a test oracle. The oracle is gradually populated with known correct and incorrect results, as the programmer accepts or rejects key test sequences, using a dialog that presents one sequence at a time. By making a mixture of opportunistic and conservative assumptions, *JWalk* predicts further test outcomes, given the initial results. For example, *void* methods typically yield no result (but may raise exceptions); and sequences with observer-methods in their prefix are predicted to yield the same result as shorter sequences without the observers. When used incrementally, *JWalk* predicts over 90% of test outcomes (amortized over test depth; see section 5), allowing significantly large numbers of paths to be tested [1] for minimal human intervention.

3. Feedback-based development method

The *JWalkEditor* supports a novel paradigm for specification, coding and testing, which contrasts both with formal development, and with more recent agile approaches. Formal software development methods

have an initial specification stage, in which the design is specified in a formal language, such as *Z* or *VDM*, or using a state-based tool, such as *SDL* or *Statemate*. This approach supports fully automated and systematic test generation, using the specification to inform the selection of test cases and determine coverage; but has the extra overhead of developing a specification in the first place; and runs the risk that the software may later evolve independently.

More recently, agile development methods, such as *extreme programming* (XP), have advocated a "test-first" approach [10] in which programmers create tests before writing software, or "test-driven development" [11] in which testing and coding are inter-dependent activities. The tests take the place of a formal specification, encoding the properties that the software must eventually satisfy. This has the advantage that the specification (*viz.* the test-set) is executable, but the disadvantage that it is developed in a piecemeal way, according to the fallible insights of the programmer, who must constantly update the test-set if the production code is modified (arguably no easier than maintaining the validity of a formal specification *w.r.t.* evolving code).

The *JWalkEditor* offers a new approach, in which the programmer is entirely free to prototype the code as they wish; and the tool supports this by "growing" an associated specification, which evolves in step with the code. The specification is in the form of the saved oracle, generated during interactive testing. At first, the tool presents key test cases to the programmer for confirmation (state-modifying sequences, followed by single observations); but later it uses saved results to predict further test outcomes by rule [1]. Whenever a novel test outcome is observed (because it breaks a prediction, or contradicts a previously-saved result), the tool requests another confirmation. Otherwise, it assumes that the existing prediction is still valid (which holds in practice most of the time – see the discussion below; and [1]). In this way, *JWalk* incrementally builds a bounded, exhaustive model of an algebraic specification. It then generates a more abstract, high-level state-based specification, exploring the test class's state space, using state-predicate methods in the class's interface to identify any interesting states. *JWalk* acquires the *state cover* test set (reaching all states) and from this may generate the *transition cover*, the *switch-1 cover* (all method pairs), the *switch-2 cover* (all method triples), starting from each state.

But the tool also offers something else that is quite valuable, namely an on-the-fly validation of the latest design choices. When the programmer makes a change to the code in the editor, *JWalk* may immediately explore the consequences of the latest modification, systematically revealing the effects of novel interleavings of methods and exposing corner-cases (such as testing nullops, or all interleaved observer-methods for their unexpected side-effects), which the programmer might not have fully considered. This experience is somewhat similar to that of model-validation from a partial specification, the approach adopted by model-checking tools such as *Alloy* [12]. For this reason, it is relevant to consider the *JWalkEditor* also as a kind of specification tool, which helps the programmer to determine dynamically the desired design for the class under development. We call this a "feedback-based" approach to specification, since the programmer may immediately see the consequences of particular design choices.

The cyclic development methodology is extremely habitable, because it capitalizes on what the different parties do best. Programmers are motivated mostly by the creativity of writing new code, and the gratification of seeing this execute, rather than by the process of creating a watertight specification. On the other hand, automated tools are better at performing systematic tasks, such as exploring all method combinations, states and transitions. The process of building confidence in the design is also on a human scale, since the tool presents information gradually to the programmer, showing first the obvious cases, then only presenting interesting novel cases, which could not be predicted.

4. An example of class development

To illustrate the experience of developing code in the *JWalkEditor*, the following example is given, as an indication of how a typical apprentice programmer might approach the task of providing two classes, related by inheritance. The first class, a *LibraryBook*, has the following structure:

```
public class LibraryBook {
  private String borrower;
  public LibraryBook();
  public void issue(String);
  public void discharge();
  public String getBorrower();
  public Boolean isOnLoan();
}
```

Initially, the programmer codes *issue* and *discharge* to set and clear the *borrower* attribute, and ensures that *isOnLoan* returns *true* when *borrower != null*. Next, a *protocol-walk* is performed, which reveals interesting observations: sequences that repeat *issue*, or *discharge* are apparently acceptable! The programmer considers this, deciding that it is legitimate for *discharge* to be a

nullop when the *LibraryBook* is not on loan, but that sequences repeating *issue* violate the business rules of the library. So, he returns to the editor, and inserts a precondition into the *issue* method, which raises an exception if an attempt is made to *issue* the *LibraryBook* to more than one borrower. Focusing now on state-modifying sequences, the programmer initiates an *algebra-walk* (see fig. 1) to confirm that the precondition correctly raises the exception. After this, he may explore algebraic constructions to greater depth, to be assured that it is possible to *discharge* and then *issue* the *LibaryBook* to a different borrower.

At this point, the observed behaviour seems acceptable, so the programmer switches to the *algebra-test* mode, in order to build the oracle, and confirms each presented observation as correct. In order to verify the class more thoroughly, he then selects the *state-test* mode, in which *JWalk* identifies two abstract design states, the *Default* state and the *OnLoan* state, determined from the *false* and *true* outcomes of the state predicate *isOnLoan*. *JWalk* computes the state cover, and tests all interleaved method sequences, starting in each of these states, to the desired depth. If the depth parameter is 3, this is equivalent to testing the *switch-2 cover* [13], which according to Chow's testing theory is sufficient to guarantee the correct behaviour of even a poorly implemented test class, containing redundant states and duplicated paths of length 2.

Subsequently, the programmer wishes to extend the behaviour of the *LibraryBook*, in a subclass called *ReservableBook*. This has the structure:

```
public class ReservableBook
             extends LibraryBook {
  private String requester;
  public ReservableBook();
  public void reserve(String);
  public void cancel();
  public String getRequester();
  public Boolean isReserved();
}
```

Let us assume that the programmer expects to validate combinations of the state-modifying methods *reserve*, *cancel* and observers *getRequester, isReserved* in a similar style to the above. What he may not have anticipated is that methods inherited from *LibraryBook* interact with *ReservableBook*'s methods in unexpected ways (he has "tunnel vision", a common fault).

In *algebra-test* mode, *JWalk* imports the existing oracle for the *LibraryBook* superclass, using this as the basis for the new oracle. The tool exercises *reserve* and *cancel* as expected, but does not re-present any of the old mutation sequences that involved only *issue* and *discharge*, which it can predict from the old oracle. However, previously unseen sequences that interleave *reserve* and *cancel* with the inherited state-modifying method sequences containing *issue* and *discharge* are presented. This is a considerable improvement over regression testing with saved test-sets in *JUnit*, since it interleaves local and inherited methods in all possible combinations, rather than simply applying the superclass's test-set as a whole to the subclass, which has been proven to hide introduced faults [2].

At depth 2, the *algebra-test* interleaves *reserve* and *issue.* At depth 3, *getBorrower* and *getRequester* observe that a book can be reserved by *borrower-A* and then issued to *borrower-B*, which violates the library's business rules again. The programmer is able to *reject* this outcome, which is logged in the oracle as a known fault. Furthermore, *issuing* the book should cancel the prior reservation (and does not). At the end of the test cycle, all known faults are listed in a summary. Returning to the editor, the programmer decides to override *issue* in *ReservableBook* to ensure that a book is only loaned (a) if it was not reserved, or (b) if the new borrower is the requester who reserved it; and then the prior reservation should be cancelled. Re-compiling the test-class, he re-runs the *algebra-test*, and this time is only presented with the cases involving the modified code, which he *confirms* as correct.

Finally, to demonstrate *JWalk's* ability to detect interesting high-level states, the *state-test* mode may be selected. *JWalk* will detect four abstract design states: *Default, OnLoan, Reserved* and *OnLoan&Reserved*, named automatically after the *boolean* product of the predicates *isOnLoan* and *isReserved* (which yield *false* and *true* in four combinations). *JWalk* will determine how to reach each of these states and may be directed to verify the *switch-n* cover. In this test mode, *JWalk* will predict most test results, either from previously seen cases (during the *algebra-test*) or by rule-based prediction, identifying equivalence-classes of test sequences, which all map onto canonical sequences with no observers in the prefix. Some longer unseen sequences that start in the more distant states (e.g. *OnLoan&Reserved*) will request new confirmations.

5. Experimental Evaluation

The effectiveness of this cyclic feedback-based coding, specification and testing method can be measured in several ways. Firstly, the number of new test-confirmations in each cycle is small, compared to the overall number of automated tests. Table 1 shows the amortized cost of confirmations over test cycles of increasing depth, for the *algebra-test* mode (*a1, a2, a3*), followed by the *state-test* mode (*s1, s2, s3*). The

rows marked "*con*" denote new manual confirmations per depth cycle, while the rows marked "*pre*" denote automated retests and predictions, which increasingly dominate the *state-test* results. The level of automation rises from 40% to well over 90%. But even if confirmations are *not* amortized over test cycles, they still form a small fraction of overall tests executed: 20/138 or 14% for the *LibraryBook*, and 167/1816 or 9% for the *ReservableBook*.

Table 1. Amortized user interaction costs

Test class	*a1*	*a2*	*a3*	*s1*	*s2*	*s3*
LibBk con	3	5	7	0	0	5
LibBk pre	2	8	18	18	38	133
ResBk con	3	14	56	0	11	83
ResBk pre	6	27	89	36	241	1649

With practice, a programmer can confirm each key test-result in 2-3 seconds, building the oracle at around 25 test cases per minute. This compares favourably against manual testing methods, in which programmers take much longer to think up suitable test cases. Table 2 shows how long it took two developers to test "the transition cover, plus argument equivalence partitions" [14] both manually "*man*", and using the tool "*jwk*" for the same examples. The time column indicates *min.sec* taken to develop and conduct tests.

Table 2. Speed and adequacy of testing

Test class	T	T_E	T_R	*Adq*	*time*
LibBk man	31	9	22	90%	11.00
ResBk man	104	21	83	53%	20.00
LibBk jwk	10	10	0	100%	0.30
ResBk jwk	36	36	0	90%	0.46

The test coverage adequacy *Adq* is expressed as a fraction of effective test cases T_E over ideal test cases T that were determined by inspection. The redundant tests T_R indicate wasted effort, showing how the manual tester over-compensated, creating duplicated test cases. *JWalk*'s coverage was nearly total (100% effective on state-based criteria, but missing 4 partitions on input-criteria: *JWalk* does not yet perform full equivalence-partition testing).

The power of *JWalk* comes from its predictive rules, especially the predictions about sequence equivalence-classes (the observer-prefix elimination case); this is a strong *conservative* assumption, which always holds (side-effect-free invocations are detected empirically). A weaker *opportunistic* assumption, such as where *stack.pop()* is expected to return *void*, may not always hold, and early testing may fail to spot the missing precondition on empty stacks. Violated assumptions are usually detected by longer test-sequences in the next cycle, such as the unexpected result: *stack.size() == -1*. The same principle applies to missing overrides (the case of *issue*, above), or rare cases of double-faults that happen to map onto the correct result. Testing to depth k+1 usually exposes unwanted states masquerading as expected states in the previous cycle, c.f. Chow's method [13]. Opportunistic assumptions are so useful in cutting down the number of cases presented to the programmer, that it would be impractical to do without them. Further examples of the test-coverage of *JWalk* may be found in [14].

Acknowledgement: Thanks are due to Arne-Michael Toersel (at TaicPart '07) for test cases used to develop the *JWalk* beta-2 revision.

6. References

[1] A. J. H. Simons, "JWalk: a tool for lazy systematic testing of Java classes, by design introspection and user interaction", *J. Auto. Softw. Eng.*, *14 (4)*, 2007, 369-418.
[2] A. J. H. Simons, "A theory of regression testing for behaviourally compatible object types", *Softw. Test., Verif. Reliab., 8(2)*, 2006, 133-156.
[3] A. J. H. Simons, "JWalk: lazy systematic unit testing", http://www.dcs.shef.ac.uk/~ajhs/jwalk/, 2007.
[4] O. Gruber, B. J. Hargrave, J. McAffer, P Rapicault and T. Watson, "The Eclipse 3.0 platform: adopting OSGi technology", *IBM Sys. J., 44(2),* 2005, 289-299.
[5] N. Griffiths, "Test-as-you-code Java editor", BSc dissert., Dept. Comp. Sci., University of Sheffield, 2008.
[6] S. Pestov et al., "jEdit programmer's text editor", http://www.jedit.org/, accessed 8 May 2008.
[7] The Eclipse Foundation, "Eclipse – an open development platform", http://www.eclipse.org/, accessed 8 May 8, 2008.
[8] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness checker for Java", *Software: Practice and Experience, 34(11),* 2004, pp. 1025-1050.
[9] T. Xie, D. Marinov and D. Notkin, "Rostra: a framework for detecting redundant object-oriented unit tests", *Proc. 19th IEEE Conf. Automated Softw. Eng.*, IEEE, Washington DC, 2004, pp. 196-205.
[10] K. Beck, *Extreme Programming Explained: Embrace Change, 1st edn*. New York: Addison-Wesley, 2000.
[11] ibid., *2nd edn*. New York: Addison-Wesley, 2005.
[12] D. Jackson, *Software Abstractions: Logic, Language and Analysis*, MIT Press, 2006.
[13] T. S. Chow, "Testing software design modeled by finite-state machines", *IEEE Trans. Softw. Eng., 4(3),* 1978, 178-187.
[14] A. J. H. Simons and C. D. Thomson, "Benchmarking effectiveness for object-oriented unit testing", *Proc. 1st Software Testing Benchmark Workshop, IEEE 1st Int. Conf. Softw. Testing*, Lillehammer, 2008.