Testing with Guarantees and the Failure of Regression Testing in eXtreme Programming

Anthony J.H. Simons

Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield S1 4DP, UK
A.Simons@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/~ajhs/

Abstract. The eXtreme Programming (XP) method eschews all formal design, but compensates for this by rigorous unit testing. Test-sets, which constitute the only enduring specification, are intuitively developed and so may not be complete. This paper presents a method for generating complete unit test-sets for objects, based on simple finite state machines. Using this method, it is possible to prove that saved regression test-sets do not provide the expected guarantees of correctness when applied to modified or extended objects. Such objects, which pass the saved tests, may yet contain introduced faults. This puts the whole practice of regression testing in XP into question. To obtain the same level of guarantee, tests must be regenerated from scratch for the extended object. A notion of guaranteed, repeatable quality after testing is defined.

1 Introduction

The popular eXtreme Programming (XP) method throws away the formal analysis and design stages of conventional software engineering [1, 2, 3] in reaction to the high overhead of standard development processes and the extra documentation that these require [4, 5]. However, the lack of formal design in XP has received some constructive criticism [6]: its "extremeness" may be characterized by this and the fact that the software base is subject to constant modification during the lifetime of the project, with all developers granted the freedom to change any part of the software.

To guard against inadvertently damaging the code base, a great emphasis is placed upon *testing*. Unit tests are developed, sometimes directly from the requirements (*test-first design*), but mostly in parallel with the code (*test-driven development*). Tools supporting the unit testing approach have been developed, such as JUnit, a popular tool for unit testing single classes [7, 8]. Every time the object under test (OUT) is changed, the software must be exercised again with all the existing test-sets, to ensure that no faults have been introduced (known as *regression testing*). This, with the emphasis on regular builds in small increments, is intended to guard against the effects of entropy. (XP also advocates *acceptance testing*, whereby the system is evaluated against user requirements prior to delivery; this aspect is not under investigation in the current paper).

1.1 Parallel Design and Test

Practical object-oriented unit testing has been influenced considerably by the *non-intrusive* testing philosophy of McGregor *et al* [9, 10]. In this approach, every OUT

H. Baumeister et al. (Eds.): XP 2005, LNCS 3556, pp. 118–126, 2005.

has a corresponding test-harness object (THO), which encapsulates all the test-sets separately, but may have privileged access to observe the internal states of the OUT (e.g. via *friend* declarations in C++). The OUT is therefore uncluttered by test code and may be delivered as-is, after testing. This separation of concerns lies behind the *parallel design and test architecture* (see figure 1), in which an isomorphic inheritance graph of test harness classes shadows the graph of production classes [10].

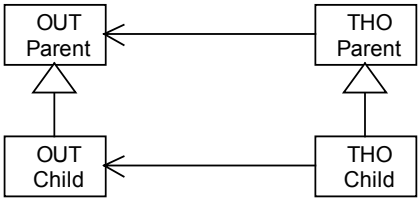

Fig. 1. McGregor's software architecture for parallel design and test abstractions. Motivated by the separation of concerns between production and test code, this leads to the notion of inheritable test-sets, which are reused when testing refined classes

From this arises a beguiling notion: since a child class is an extension of its parent, so the test-sets for the child must be simple extensions of the test-sets for the parent. The presumed advantage is that most test-sets can be *inherited* from the parent THO and reapplied, *as a suite*, to the child OUT, in a kind of regression test. The child THO is understood to supply additional test-sets to exercise the new methods introduced in the child OUT [9, 10]. Below, we show just how unreliable this intuition is.

1.2 Recycling Unit Test Sets in XP

The JUnit tool fosters a similar strategy for re-testing classes that are subject to continuous *refactoring* (internal reorganization yielding the same external behaviour) and *subclassing* (extension without invalidating old behaviour) [7, 8]. JUnit allows programmers to develop and save *test suites*, which may be executed repeatedly. One of the key benefits of JUnit is that it makes the re-testing of refactored and subclassed objects semi-automatic, so it is widely used in the XP community, in which the recycling of old test-sets has become a major part of the quality assurance strategy.

XP claims that a programmer may incrementally modify code in iterative cycles, so long as each modification passes all the original unit tests: "Unit tests enable refactoring as well. After each small change, the unit tests can verify that a change in structure did not introduce a change in functionality" [11]. There are two sides to this claim. Firstly, if the modified code *fails* any tests, it is clear that faults have been introduced (tests as diagnostics). Secondly, there is the assumption that modified code which *passes* all the tests is still as secure as the original code (tests as guarantees). This assumption is unsound and unsafe. After passing the recycled tests, objects may yet contain faults introduced by the refinement. The guarantee provided by passing the recycled tests is strictly weaker than the original guarantee that the same test-set provided in its original context. Using simple state-based models of objects, it is possible to show why recycled test-sets provide incommensurate, and therefore inadequate, test coverage in the new context.

2 Generating Complete Unit Test-Sets

State machines have frequently been used as models of objects, both to articulate their design [12, 5], and more formally as part of a testing method [13, 14, 15, 16]. In the more rigorous approaches, it is possible to develop a notion of *complete test coverage*, based on the exhaustive exploration of the object's states and transitions. The following is an adaptation of the X-Machine testing method [16, 17], which offers stronger guarantees than other methods, in that its testing assumptions are clear and it tests *negatively* for the absence of all undesired behaviour as well as *positively* for the presence of all desired behaviour. It is well known that programmers fail to consider the former, since it is hard to think of all the unintended ways in which a system might possibly be abused, when the focus is on positive outcomes. No matter how simple and direct the hand-crafted tests might appear, if they don't cover the object's state space, they are *incomplete*, and only offer a false sense of security.

Earlier companion work showed how easy and practical it is to generate simple state machines directly from XP story cards [18] and UML use cases [19]. This supported the automatic generation of complete *user-acceptance tests*. In [20] the automatic generation of complete *unit tests* from state machines was shown to outperform ad-hoc test scripting. The focus of the current paper is *regression testing* and complete test *regeneration* in the context of object evolution.

2.1 State-Based Design

An object exists in a series of states, which are chosen by the designer to reflect modes in which its methods react differently to the same messages. The number of states an object can have depends on the independently-varying values of all of its attributes. The theoretical maximum is the Cartesian product of its attribute domains. Typically, this fine-grained attribute space is partitioned into coarser states, in which behaviour is qualitatively "the same" in each partition [15, 21]. We define states more abstractly, as partitions of the product of the ranges of an object's access methods [22]. This permits the design of state machines for fully abstract classes and interfaces, which by definition have no concrete attributes.

Figure 2 illustrates a simple state machine, describing the modal behaviour of a *Stack*. This could represent the design of a concrete class, or an abstract interface. Transitions for the state-modifying methods {*push, pop*} are shown explicitly. For completeness, a finite state machine must define a transition for *every* method in *every* state. We assume by convention that the access methods {*top, empty, size*}, which are not shown, all have implicit transitions looping from each state back to itself.

The state machine is developed by considering the modes in which certain methods behave differently. Here, *pop* and *top* are legal in the *Normal* state, but undefined in the *Empty* state. There is a single transition to the initial state (representing object construction). Transitions are then added for *every* method in *every* state. If an object is no longer useable, it may enter a final state (such as the error state after the illegal *pop* from *Empty*). If multiple transitions could fire for the same message request, the designer should resolve this nondeterminism by placing guards, mutually exclusive and exhaustive conditions, on the conflicting transitions (such as on the two *pop* tran-

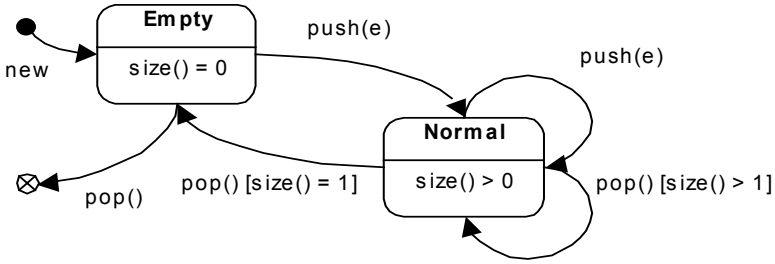

Fig. 2. State machine for a *Stack*. The two states (*Empty, Normal*) are defined on a partition of the range of the *size* access method. Only state-modifying transitions are shown explicitly

sitions from *Normal*). Certain design-for-test conditions may apply, to ensure that the machine can be driven deterministically through all of its states and transitions [16]. For example, in order to know when the final *pop* transition from *Normal* to *Empty* is reached, the accessor *size* is required as one of *Stack*'s methods, so that suitable guards may be constructed. The above design is logically *complete*, in the sense that the modal behaviour of every method from every state is known.

2.2 Elements of the State Model

In formal approaches to state-based test generation, test-sets are constructed from *languages,* sequences of symbols drawn from the total *alphabet* of transition labels. The *alphabet* is the set of methods $m \in M$ that can be called on the interface of the object (including any inherited methods). For the *Stack* shown in figure 2, the alphabet M = {*push*, *pop*, *top*, *empty*, *size*}. Note that *new* is not technically in the method-interface of *Stack*, but rather constructs the *Stack* instance. The object responds to all $m \in M$, and to no other methods (which are ruled out by compile-time syntax checking). This puts a useful upper bound on the scope of negative testing.

The state-transition diagram defines a number states $s \in S$. The states of the *Stack* in figure 2 are S = {*Empty, Normal*}. The "final state" is not treated as a first-class state, but as an exception raised by *pop* from the *Empty* state. It is assumed that the test harness can construct a predicate p : *Stack* → *Boolean* for each state $s \in S$, to detect whether the OUT is in that state. Predicates may freely use public access methods internally: for example, *isEmpty(s)* ≡ *s.size() = 0*. Our earlier definition of state (see 2.1) ensures that such predicates may always be defined.

Sequences of methods, denoted $<m_1, m_2, \ldots>$, $m \in M$, may be constructed. *Languages* M^0, M^1, M^2, … are sets of sequences of specific lengths; that is, M^0 is the set of zero-length sequences: {<>} and M^1 is the set of all unit-length sequences: $\{<m_1>, <m_2>, \ldots\}$, etc. The infinite language M* is the union $M^0 \cup M^1 \cup M^2 \cup \ldots$ containing all arbitrary-length sequences. A predicate language $P = \{<p_1>, <p_2>, \ldots\}$ is a set of unit-length predicate sequences, testing exhaustively for each state $s \in S$.

2.3 Complete Unit-Test Generation

When testing from a state-based design, the tester drives the OUT into all of its states and then attempts every possible transition (both expected and unwanted) from each

state, checking afterwards which destination states were reached. The OUT should exhibit indistinguishable behaviour from the design, to pass the tests. It is assumed that the design is a *minimal* state machine (with no duplicate, or redundant states), but the tested implementation may be non-minimal, with more than the expected states. Testing must therefore take this into account, exercising more than the minimal state machine. These notions are formalised below.

The *state cover* is a set $C \subseteq M^*$ consisting of the shortest sequences that will drive the OUT into all of its states. C is chosen by inspection, or by automatic exploration of the model. For the *Stack* shown in figure 2 above, C = {<>, <*push*>} is the smallest state cover, which will enter the *Empty* and *Normal* states. An initial test-set T^0 aims to reach and then verify every state. Verification is accomplished by invoking each predicate in the predicate language P after exploring each path in the state cover C, a test set denoted by: $C \otimes P$, the *concatenated product* that appends every sequence in P to every sequence in C. Exactly one predicate in each sequence should return true, and all the others false, as determined from the model.

$$T^0 = C \otimes P \tag{1}$$

A more sophisticated test-set T^1 aims to reach every state and also exercise every single method in every state. This is constructed from the *transition cover,* a set of sequences $K^1 = C \cup C \otimes M^1$, which includes the state cover C and the concatenated product term $C \otimes M^1$, denoting the attempted firing of every single transition from every state. The states reached by the transition cover are validated again using all singleton predicate sequences $<p> \in P$.

$$T^1 = (C \cup C \otimes M^1) \otimes P \tag{2}$$

An even more sophisticated test-set T^2 aims to reach every state, fire every single transition and also fire every possible pair of transitions from each state (sometimes known as the *switch cover*). This is constructed from the augmented set of sequences $K^2 = C \cup C \otimes M^1 \cup C \otimes M^2$ and the reached states are again verified using the predicates. The product term $C \otimes M^2$ denotes the attempted firing of all pairs of transitions from every state.

$$T^2 = (C \cup C \otimes M^1 \cup C \otimes M^2) \otimes P \tag{3}$$

In a similar fashion, further test-sets are constructed from the state cover C and low-order languages $M^k \subseteq M^*$. Each test-set subsumes the smaller test-sets of lesser sophistication in the series. In general, the series can be factorised and expressed for test-sets of arbitrary sophistication as:

$$T^k = C \otimes (M^0 \cup M^1 \cup M^2 \ldots M^k) \otimes P \tag{4}$$

The general formula describes all test-sequences starting with a state cover, augmented by firing all single, pairs, triples etc. of transitions and then verifying the reached states using state predicates.

3 Test Completeness and Guarantees

The test-sets produced by this algorithm have important completeness properties. For each value of k, specific guarantees are obtained about the implementation, once testing is over. Below, we show how regression testing in XP does not provide the

same guarantees after retesting. However, for simple extensions to a state-based design, tests may be re-generated by this algorithm and all the same guarantees upheld.

3.1 Guarantees After Testing

The set T^0 guarantees that the implementation has *at least* all the states in the specification. The set T^1 guarantees this, and that a *minimal* implementation provides exactly the desired state-transition behaviour. The remaining test-sets T^k provide the same guarantees for *non-minimal* implementations, under weakening assumptions about the level of duplication in the states and transitions.

A non-minimal, or *redundant* implementation is one where a programmer has inadvertently introduced extra "ghost" states, which may or may not be faithful copies of states desired in the design. Test sequences may lead into these "ghost" states, if they exist, and the OUT may then behave in subtle unexpected ways, exhibiting extra, or missing transitions, or reaching unexpected destination states. Each test-set T^k provides complete confidence for systems in which chains of duplicated states do not exceed length k-1. For small values of k, such as k=3, it is possible to have a very high level of confidence in the correct state-transition behaviour of even quite perversely-structured implementations.

Both *positive* and *negative* testing are achieved; for example, it is confirmed that access methods do not inadvertently modify object states. Testing avoids any *uniformity assumption* [23], since no conformity to type need be assumed in order for the OUT to be tested. Likewise, testing avoids any *regularity assumption* that cycles in the specification necessarily correspond to implementation cycles. When the OUT "behaves correctly" with respect to the specification, this means that it has all the same states and transitions, or, if it has extra, redundant states and transitions, then these are semantically identical duplicates of the intended states in the specification.

3.2 Refactoring, Subclassing and Test Coverage

Figure 3 illustrates the state machine for a *DynamicStack*, an array-based implementation of a *Stack*. We may think of this either as a subclass design for a concrete class that implements an abstract *Stack* interface [22], or as a refactored design for a *Stack*, after the decision is taken to switch to an array-based implementation. The main difference between this and the earlier machine in figure 2 is that the old *Normal* state, now only shown as a dashed region, has been partitioned into the states {*Loaded, Full*}, in order to capture the distinct behaviour of *push* in the *Full* state, where this triggers a memory reallocation. Though this design appears more complex, it is easily generated by following the simple rule: show the behaviour of *every* method in *every* state (self-transitions for the access methods are inferred, as above).

Increasing the state-space has important implications for test guarantees. Consider the sufficiency of the T^2 test-set, generated from the abstract *Stack* specification in figure 2. This robustly guarantees the correct behaviour of a simple *LinkedStack* implementation with S = {*Empty, Normal*}, even in the presence of "ghost" states. T^2 will generate one sequence <*push, push, push, isNormal*>, which robustly exercises <*push, push*> from the *Normal* state and will even detect a "ghost" copy of the *Normal* state.

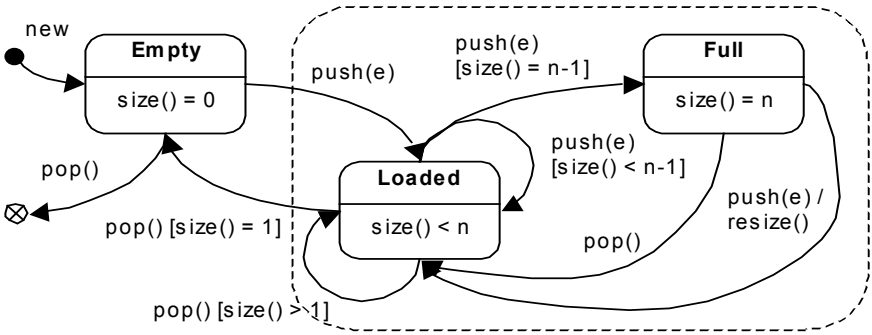

Fig. 3. State machine for a *DynamicStack*, which conforms to the behaviour of the *Stack* in fig. 2. The two states (*Loaded, Full*) partition the old *Normal* state in fig. 2, resulting in the replication of its transitions. The behaviour of *push* in the *Full* state must be tested

In XP regression testing, saved test-sets are reapplied to modified or extended objects in the expectation that passing all the saved tests will guarantee the same level of correctness. If the *Stack's* T^2 test-set were reused to test the *DynamicStack* in figure 3, with S = {*Empty, Loaded, Full*}, the resizing *push* transition would never be reached, since this requires a sequence of four *push* methods. To the tester, it would appear that the *DynamicStack* had passed all the saved T^2 tests, even if a fault existed in the resizing *push* transition. This fault would be undetected by the saved test-set.

In general, subclasses have a larger state-space than superclasses, due to their introduction of more methods affecting object states. In Cook, Daniels [24] and McGregor's [15, 21] state models, subclasses could introduce new states. In Simons' more careful model of state refinement [22, 25], subclassing was shown always to result in the *subdivision of existing states* (as in figure 3). The current paper shows that saved tests not only miss the new states formed in the child, but also fail to test all of the transitions that were tested in the parent, due to the splitting [22] of these transitions. This is important for regression testing, since it means that saved tests exercise *significantly less* of the *same behaviour* in the child as they did in the parent.

To achieve the same level of coverage, it is not sufficient to supply extra tests for the new methods, but rather it is vital to test *all the interleavings* of new methods with the inherited methods, so exploring the state-transition diagram completely. This simply cannot be done reliably by human intuition and manual test-script creation. However, it could be done reliably if the test-sets were *regenerated* for the modified state machine designs, using the algorithm from section 2.3 above.

4 Conclusions

The strength of the guarantee obtained after regression testing is overestimated in XP. Recycled test-sets always exercise significantly less of the refined object than the original, such that re-tested objects may be considerably less secure, for the same testing effort. As the state-space of the modified or extended object increases, the guarantee offered by retesting is progressively weakened. This undermines the validity of popular regression testing approaches, such as parallel design-and-test, test set inheritance and reuse of saved test scripts in JUnit.

By comparison, if complete test-sets are always generated from simple state machine designs, it is possible to provide specific guarantees, for example up to the k=2 or k=3 confidence level, in the OUT. After retesting a subclassed or refactored OUT, the same guarantees may be upheld by generating from the revised state machine to the same confidence level. Not only this, but *all* the objects in a software project may be unit-tested to a given confidence level. This notion of *guaranteed, repeatable quality* is a new and important concept in object-oriented testing.

This research was undertaken as part of the MOTIVE project, supported by UK EPSRC GR/M56777.

References

1. Wells, D.: The Rules and Practices of Extreme Programming. Website and hypertext article http://www.extremeprogramming.org/rules.html (1999)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, New York (2000)
3. Beck, K. and Fowler, M.: Planning Extreme Programming. Addison-Wesley, New York (2000)
4. Kruchten, P.: The Rational Unified Process: an Introduction. 3rd edn. Addison-Wesley, Reading (2003)
5. Object Management Group, UML Resource Page. Website http://www.omg.org/uml/ (2004)
6. Stephens, M. and Rosenberg D.: Extreme Programming Refactored: The Case Against XP. Apress, Berkley (2003)
7. Beck, K. Gamma E. et al.: The JUnit Project. Website http://www.junit.org/ (2003)
8. Stotts, D., Lindsey, M. and Antley, A.: An Informal Method for Systematic JUnit Test Case generation. Lecture Notes in Computer Science, Vol. 2418. Springer Verlag, Berlin Heidelberg New York (2002) 131-143
9. McGregor, J. D. and Korson, T.: Integrating Object-Oriented Testing and Development Processes. Communications of the ACM, Vol. 37, No. 9 (1994) 59-77
10. McGregor, J. D. and Kare, A.: Parallel Architecture for Component Testing of Object-oriented Software. Proc. 9th Annual Software Quality Week, Software Research, Inc. San Francisco, May (1996)
11. Wells, D.: Unit Tests: Lessons Learned, in: The Rules and Practices of Extreme Programming. Hypertext article http://www.extremeprogramming.org/rules/unittests2.html (1999)
12. Schuman, S. A. and Pitt, D. H.: Object-oriented Subsystem Specification. In: Program Specification and Transformation. Elsevier Science, North Holland (1987)
13. Chow, T.: Testing Software Design Modeled by Finite State Machines. IEEE Transactions on Software Engineering, Vol. 4 No. 3 (1978) 178-187
14. Binder, R. V.: Testing Object-Oriented Systems: a Status Report. 3rd edn. Hypertext article http://www.rbsc.com/pages/oostat.html (2001)
15. McGregor, J. D.: Constructing Functional Test Cases Using Incrementally-Derived State Machines. Proc. 11th International Conference on Testing Computer Software. USPDI, Washington (1994)
16. Holcombe, W. M. L. and Ipate, F.: Correct Systems: Building a Business Process Solution. Applied Computing Series. Springer Verlag, Berlin Heidelberg New York (1998)
17. Ipate, F. and Holcombe, W. M. L.: An Integration Testing Method that is Proved to Find All Faults. International Journal of Computational Mathematics, Vol. 63 (1997) 159-178

18. Holcombe, M., Bogdanov, K. and Gheorghe, M.: Functional Test Generation for Extreme Programming. Proc. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering. XP 2001, Sardinia, Italy (2001) 109-113
19. Dranidis, D., Tigka, K. and Kefalas, P.: Formal Modelling of Use Cases with X-machines. Proc. 1st South-East European Workshop on Formal Methods. South-Eeast European Research Centre, Thessaloniki (2004)
20. Holcombe, M.: Where Do Unit Tests Come From? Proc. 4th. International Conference on Extreme Programming and Flexible Processes in Software Engineering. XP 2003, Genova, Italy, Lecture Notes in Computer Science, Vol. 2675. Springer Verlag, Berlin Heidelberg New York (2003) 161-169
21. McGregor, J. D. and Dyer, D. M.: A Note on Inheritance and State Machines. Software Engineering Notes, Vol. 18, No. 4 (1993) 61-69
22. Simons, A. J. H., Stannett, M. P., Bogdanov, K. E. and Holcombe, W. M. L.: Plug and Play Safely: Behavioural Rules for Compatibility. Proc. 6th IASTED International Conference on Software Engineering and Applications. SEA-2002, Cambridge (2002) 263-268
23. Bernot, B., Gaudel, M.-C. and Marre, B.: Software Testing Based on Formal Specifications: a Theory and a Tool. Software Engineering Journal, Vol. 6, No. 6 (1991) 387-405
24. Cook, S. and Daniels, J.: Designing Object-Oriented Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, London (1994)
25. Simons, A. J. H.: Letter to the Editor, Journal of Object Technology. Received December 5, 2003. Hypertext article http://www.jot.fm/general/letters/comment_simons_html (2003)