A theory of regression testing for behaviourally compatible object types

Anthony J. H. Simons*

Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, Sheffield S1 4DP, U.K.

SUMMARY

A behavioural theory of object compatibility is presented, which has implications for object-oriented regression testing. The theory predicts that only certain models of state refinement yield compatible types, dictating the legitimate design styles to be adopted in object statecharts. The theory also predicts that conformity-testing using regression tests is inadequate. Functionally complete test-sets that are applied as regression tests to subtype objects are usually expected to cover the functionality of the original type, even though they are clearly not expected to cover extra functionality introduced in the subtype. However, such regression testing is proven to cover strictly less than the original state-space in the new context and so provides much weaker confidence than expected. A different retesting model is proposed, based on full automatic test regeneration from the subtype's specification. This method can guarantee equivalent levels of confidence after retesting. The behavioural conformity desired by regression testing can then be proven by verification in the theory. Copyright © 2006 John Wiley & Sons, Ltd.

Received 8 December 2005; Accepted 9 March 2006

KEY WORDS: object-oriented; behavioural subtyping; state refinement; state-based testing; regression testing; testing adequacy

1. INTRODUCTION

Practical object-oriented unit testing is influenced considerably by the non-intrusive testing philosophy of McGregor and co-workers [1,2]. In this approach, every object under test (OUT) has a corresponding...
test-harness object (THO), which encapsulates all the test-sets separately. This separation of concerns is the main motivation for McGregor’s *parallel design and test architecture*, in which an isomorphic inheritance graph of test harness classes shadows the graph of production classes [2]. This embodies the beguiling intuition that, since a child class is an extension of its parent, so the test-sets for the child are extensions of the test-sets for the parent. The intended advantage of *parallel design and test* is that the effort invested in creating standard test-sets is not wasted. Test-sets can be inherited from the parent THO and applied, *as a suite*, to the child OUT in a kind of regression test, to ensure that the child class still delivers all the functionality of the parent. While it is clearly understood that the child THO will supply additional test-sets to exercise methods introduced in the child OUT [1,2], the act of encapsulating the inherited tests *as a suite* is shown below to be significant. Contrary to intuition, retesting with this suite *does not confirm* the behaviour of the parent in the child, since it can be proven (see Section 4.2) that the saved test suite exercises *strictly less* of the parent’s state-space in the child than it originally exercised in the parent.

More recently, the JUnit tool has fostered a similar strategy for re-testing classes that are subject to continuous modification and extension [3,4]. JUnit allows programmers to develop test scripts, which are converted into suites of methods behind the scenes. These are executed on demand, to test objects and to re-test modified or extended versions of those objects. One of the key benefits of JUnit is that it makes the re-testing of modified objects semi-automatic, so it is widely used in the eXtreme Programming (XP) community, in which the recycling of test-sets has become a major part of the quality assurance strategy. While XP is to be commended for putting rigorous testing back on the popular agenda, it is difficult to identify precisely what XP expects to gain from its retesting strategy, which seems to address both sociological and technical concerns. Up to three possible reasons for retesting can be inferred from the standard informal account [5].

Initially, test creation takes the place of writing a specification:

‘Development is driven by tests. You test first, then code. Until all the tests run, you aren’t done. When all the tests run, and you can’t think of any more tests that would break, you are done adding functionality’ [5, p. 9].

Test creation is manual, with test cases selected by human intuition. This can provide good diagnostics for anticipated fault classes, but no absolute guarantee of test completeness or correctness after testing. The general position seems to be that writing more tests progressively reduces the defect rate to a statistically acceptable level [5].

The value of automated retesting is partly psychological, creating a sense of well-being and confidence in the programmer and in the customer. In the JUnit community, programmers talk of ‘clicking the test-button and waiting for the test-result bar to turn green’:

‘I just wanted a little jolt of confidence. Seeing that the tests still ran gave me that’ [5, p. 46].

‘You feel good when you see all the tests running.’ . . . ‘The customer feels good about the system when they see all of their tests running’ [5, p. 66].

A second motivation for retesting is to engage in fault detection. It is logically sound to infer that, if a test fails, then a fault has been introduced. However, it is not logically sound to infer that, if all the tests succeed, then no additional faults have been introduced:
'When the machine is free, a pair with code to integrate sits down, loads the current release, loads their changes (checking for and resolving any collisions), and runs the tests until they pass (100% correct)' [5, p. 60].

'You can run the tests quickly so you know you haven’t broken anything.' [5, p. 68].

This is starting to move into dangerous territory, since it is tending towards claiming that the program has not broken after successful retesting (faults may have been introduced in the original functionality that the tests did not detect—see Section 4.2). Such false confidence is encouraged by the ready availability of automatic retesting tools:

'Tests must be automatic, returning an unqualified thumbs-up/thumbs-down indication of whether the system is behaving’ [5, p. 116].

'When I have written all the tests I can imagine could possibly break and they all pass, I’m certain my code is correct’ [6, pp. 101–102].

This reveals the third motivation for retesting, which is to confirm the correct behaviour of modified systems. In XP there is at least a strong expectation (if not an outright claim) that automated unit and integration testing provides protection against software entropy, allowing all of the programmers to modify each other’s code freely, so long as each modification passes all the original tests:

'After they have added a feature, the programmers ask if they can now see how to make the program simpler, while still running all of the tests. This is called refactoring’ [5, p. 58].

'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’ [7].

There are two sides to this kind of claim. First, if the modified code fails any tests, it is clear that faults have been introduced, so there is some benefit in reusing old tests as diagnostics. Second, there is the implicit assumption that modified code which passes all the tests is still as secure as (i.e. no more defective than) the original. Recycled tests are implicitly being used for a stronger purpose, as confirmation of a certain positive level of correctness.

This unsound expectation motivated the current investigation. If a near-perfect unit testing method were found, that could offer measurable guarantees after testing, how would this be affected by regression? How much of a modified or refined object could be tested effectively by saved test suites? Can saved test-sets be extended in any simple way to test modified objects? The findings show that the coverage of regression tests can be quantified with respect to a formal model, but that the confidence offered by regression testing is strictly less than what intuition expects. In place of regression testing, a new retesting paradigm is proposed, which is based on regenerating test-sets automatically from refined specifications, which are then proven compatible with original specifications.

In Section 2, a state-based theory of object refinement is presented, which encompasses object extension with subtyping, the concrete realization of abstract interfaces and incremental redesign with unchanged behaviour. The theory predicts that only certain models of state refinement yield compatible types, dictating the legitimate design styles to be adopted in object statecharts. In Section 3, the object X-machine theory of complete functional testing is presented. A formula is used to generate increasingly more powerful test-sets from a state-based specification that can guarantee the correctness of the tested object’s state-related behaviour, under progressively weakening assumptions.
about the quality of its implementation. In Section 4, the theory of refinement is used to predict that
conformity-testing using regression tests is inadequate. Functionally complete test-sets that are applied
as regression tests to subtype objects are usually expected to cover the state-space of the supertype in
the subtype object. However, such testing is proven to cover strictly less of the original object’s state-
space in the new context and so provides much weaker confidence than expected. Objects that pass the
recycled tests may yet contain introduced faults, which are undetected. In Section 5, it is found that
there is no simple way to extend encapsulated test suites to achieve coverage in modified objects.
Instead, the complete test generation approach is used to regenerate tests for arbitrary state-based
specifications, obtaining predictable and repeatable levels of confidence after testing. In combination
with verification in the formal model of behavioural compatibility, this satisfies the requirement for
a testing method that can certify the conformant behaviour of a modified object, with respect to an
original specification.

2. A THEORY OF COMPATIBLE OBJECT REFINEMENT

In classical automata theory, the notion of machine compatibility is judged by comparing sets of traces,
sequences of labels taken from transition paths computed through the machines in question [8,9].
Two machines are deemed equivalent if their trace-sets are equivalent. A machine is behaviourally
compatible with another if its trace-set includes all the traces of the other, that is, for every trace
in the protocol of the reference machine, such a trace also exists in the protocol of the compared
machine. Object-oriented design methods [10,11] include object statecharts, which are influenced by
Harel’s statecharts [12] and SDL [13]. These notations are more complex than simple finite state
automata. Equivalence and compatibility between statecharts are judged by considering syntactic
relations between the transformed state-spaces, from which the trace behaviour follows.

2.1. McGregor’s statechart refinements

McGregor and Dyer [14] and McGregor [15] proposed one of the early theories of object statechart
refinement. In McGregor’s model, object states derive from the object’s stored variable values, as seen
through observer methods. The machines have Mealy-semantics, with quiescent states and actions on
the transitions representing the invoking of methods. Figure 1 illustrates a contemporary reworking of
McGregor’s three main structural refinements, to allow comparison with trace models.

These kinds of refinement were deemed compatible because they observed the rules:

- all states in the base object are preserved in the refined object;
- all introduced states are wholly contained in existing states;
- all transitions in the base object are preserved in the refined object.

These structural refinements may be compared with trace models. In Figure 1, the traces of M0 are
the set \{⟨⟩, ⟨a⟩, ⟨a, b⟩\}, where ⟨a, b⟩ is a sequence of method invocations in the protocol of M0.
M1 adds an extra method c to the interface of M0. This is a derived method, analogous to function
composition [16], which computes a more direct route to the destination state S3. The traces of M1 are
\{⟨⟩, ⟨a⟩, ⟨a, b⟩, ⟨c⟩\} so it is clear that this includes the traces of M0.
M2 adds two extra methods $d$ and $e$, which examine state $S2$ at a finer granularity. $S2$ is completely partitioned into substates $S2.1$ and $S2.2$. Since states are abstractions over variable products [14], this is equivalent to dependence on disjoint subsets of variable values. The usual statechart semantics of M2 is that entry to $S2$ implies entry to the default initial substate $S2.1$, and the exit transition $b$ from $S2$ preempts other substate events. The statechart may therefore be flattened to a simple state machine, with transition $a$ leading directly from state $S1$ to $S2.1$ and an exit transition $b$ from both substates $S1.1$ and $S1.2$ to state $S3$. The traces of M2 are infinite (due to the infinite alternation of $d$, $e$), but include \{\langle \rangle, \langle a \rangle, \langle a, b \rangle, \langle a, d \rangle, \langle a, d, b \rangle, \langle a, d, e \rangle, \langle a, d, e, b \rangle, \ldots \} \) and so include all the traces of M0.

M3 introduces concurrent states $S4$, $S5$ and extra methods $d$ and $e$ that depend on the new states. This represents the definition of new variables in the object subtype, together with new methods whose behaviour is orthogonal to existing behaviour. The usual statechart semantics is that both machines execute concurrently. Formally, this is equivalent to a flat state machine containing the product of the states of the two concurrent machines, which in this paper is denoted by \{S1/4, S2/4, S3/4, S1/5, S2/5, S3/5\}. The traces of M3 are infinite, but include \{\langle \rangle, \langle a \rangle, \langle d \rangle, \langle a, d \rangle, \langle d, a \rangle, \langle a, b \rangle, \langle d, e \rangle, \langle a, d, e \rangle, \langle d, e, a \rangle, \langle a, d, b \rangle, \ldots \} \) and so include all the traces of M0.

2.2. Cook and Daniels’ statechart refinements

In their Syntropy method, Cook and Daniels [17] permit further extensions to statecharts. Their full set of refinements includes (pp. 207–8): adding new transitions, adding new states, partitioning a state into substates, splitting transitions either at source or destination substates, retargeting transitions onto...
destination substates and composition with concurrent machines. Figure 2 illustrates the three main kinds of transformational refinement not already covered above.

These refinements may also be compared with trace models. M4 refines M0 by adding a new method c leading to a new state S4. This new state represents the addition of object variables, but unlike the case M3, the associated behaviour is not orthogonal, but tightly coupled to state S1. The state S4 is sometimes described as a new external state, to distinguish this from a new substate, of the kind in M2. The traces of M4 are the set \{⟨⟩, ⟨a⟩, ⟨a, b⟩, ⟨c⟩\} and so include the traces of M0. However, this refinement breaks the second of McGregor’s rules about new states being introduced as wholly contained substates.

M5 refines M0 by splitting the exit transition b, which no longer proceeds from the S2 state boundary, but from the individual substates S2.1 and S2.2. This represents the redefinition of the method b in the refinement, to depend disjointly on the introduced substates. The overall response is equivalent to the original b. The traces of M5 are \{⟨⟩, ⟨a⟩, ⟨a, b⟩, ⟨a, d⟩, ⟨a, d, b⟩\} and so include the traces of M0. By the usual semantics of object statecharts, an exit transition from a superstate boundary is equivalent to exit transitions from every substate. It is therefore inevitable that state partitioning will split exit transitions.

Cook and Daniels [17] also allow the symmetrical case, splitting entry transitions to target different destination substates. Mutually exclusive and exhaustive guards are introduced to distinguish which of the substates should be reached by each partial transition. However, fairness in partitioning incoming transitions to all substates is later shown to be irrelevant in the retargeting rule. M6 refines M0 by retargeting the transition a onto an arbitrary substate of S2. The state S2.2 was chosen as the target in this example simply to illustrate how this is different from the default initial substate S2.1, even though the model now cannot enter S2.1. The traces of M6 are \{⟨⟩, ⟨a⟩, ⟨a, b⟩\} and so are exactly the traces of M0.
According to the classical theory of trace inclusion, all of the refinements M1–M6 may be substituted in place of M0 and will exhibit identical trace behaviour in response to the M0 events. However, it is argued below that this is an insufficient guarantee of behavioural compatibility in object-oriented programming, where objects are aliased by handles of multiple types. For this, a stronger theory is required.

2.3. Behaviourally compatible statechart refinement

The fundamental philosophical issue to decide in the theory is how to treat the introduction of new variables in subtype objects. Do these variables correspond to missing pieces of the object’s earlier state, such that their concatenation in the subtype gives rise to brand-new external states (as with M4 above)? Do these variables already exist in a virtuo at the abstract level, in which case their concrete exposure in the subtype creates new substates (as with M2 above)? Are these variables orthogonal and so give rise to concurrent states in the subtype, equivalent to state products (as with M3 above)?

These different views of state refinement are in conflict. To help resolve the issue, the contrasting refinements M2, M3 and M4 are examined in more detail. The M3 refinement can be shown to be more general than M2. By flattening M2, a statechart is obtained in which all $a$-transitions target the default initial substate, S2.1. The product machine obtained by flattening the M3 refinement is more sophisticated, since the $a$-transitions go from S1/4 and S1/5 to S2/4 and S2/5 respectively. M3 is more sensitive to orthogonal behaviour than M2. It is reasonable to assume that subtype objects must exhibit orthogonal behaviour at least some of the time, so the M3 refinement is preferred over M2.

Both M3 and M2 assume that introduced state variables are exposed as substates of existing states. This contrasts with M4, which assumes that entirely new states may be introduced. In M4, the $c$-transition takes an object entirely out of the S1 state, whereas in M3, the $d$-transition still leaves the object in its S1 state (going from S1/4 to S1/5). This means that in all contexts and under all firings of $d$- and $e$-transitions, the M3 object can be abstracted to a M0 object whereas this cannot be done for a M4 object. Abstracting away from M4 in state S4 leaves an object in no recognizable M0 state, and furthermore the object will deadlock in this state for any attempt to fire $a$-transitions. In terms of the $\pi$-calculus process algebra [18], M3 strongly simulates M0, whereas M4 only weakly simulates M0. This is discussed in Section 5.4 below.

Since refinements such as M3 must eventually be expected, in which full state products are computed, the notion of hierarchical superstates encapsulating substates, in the style of M2, becomes moot. It is more sensible to think of the old states as being completely partitioned into new states. Figure 3 illustrates this in a more compelling way. Here, L2 is the refinement resulting from the concurrent composition of L0 and L1. However, it is irrelevant whether L0 is the basis and L1 is the supplement, or vice versa. Whereas in Figure 1 it was tempting to view composition as ordered, such a view cannot be taken here. Accordingly, it cannot be said that any particular superstate hierarchy is more valid. So, superstates are discarded in favour of regions, intersecting areas enclosing states that share some common transition behaviour. In Figure 3, regions are shown as dashed outlines. Four intersecting regions can be identified in L2 that correspond to the pairs of simple states in L0 and L1.

The process of refining a state machine then becomes a matter of turning states into regions, whose enclosed states completely partition the original unrefined state. After this, the main obligation is to ensure that all the transition behaviour of the base object is preserved in the refined object.
Partitioning a state will always split outgoing transitions, for example, the \(a\)-transition from \(S_1\) is turned into a pair of partial \(a\)-transitions from \(S_{1/3}\) and \(S_{1/4}\). Since orthogonal behaviour is assumed, these also target separate partitions of \(S_2\), the states \(S_{2/3}\) and \(S_{2/4}\). However, what if the behaviours of \(c, d\) are not entirely independent of \(a, b\)? In this case, incoming transitions might be retargeted onto different states.

Let a region correspond to a state that is being refined. Retargeting has no adverse effect on the validity of the refinement, so long as the transition retargets a state within the same region. Suppose the \(a\)-transitions were retargeted onto different states within \(S_2\). No matter which target states within region \(S_2\) are chosen, it is still possible to abstract away to \(S_2\). In all cases, the partial \(a\)-transitions would be merged in a single transition from \(S_1\) to \(S_2\). Retargeting may select an arbitrary state, or combination of states within the destination region. Supposing now that the \(c\)-transition from \(S_{1/3}\) were retargeted outside the \(S_1\) region, to \(S_{2/4}\), within the different region \(S_2\). The \(c\) message now interacts unfavourably with the alternating behaviour of \(a, b\). This means that a sequence \(\langle c, a \rangle\) will deadlock from \(S_{1/3}\).

While this modification is not compatible with \(L_0\), it is compatible with \(L_1\). Retargeting must therefore be considered with respect to the compatibility relation desired between specific machines. From these considerations, a simplified set of rules for state and transition refinement may be obtained. Note that this approach comes entirely out of a theory of state partitioning, so it belongs alongside models of data refinement (see other comparisons in Section 2.4). The rules for statechart refinement are expressed operationally, to provide practical guidelines for engineers using a statechart design tool (following the style of McGregor, Cook and Daniels). With respect to the statechart for a given object type,
the statechart for a compatible object may introduce additional states, corresponding to the exposure of extra variable products, and additional transitions, corresponding to the introduction of new methods, so long as the following conditions are observed:

- Rule 1: new states are always introduced as complete partitions of existing states, which become enclosing regions;
- Rule 2: new transitions for additional methods do not cross region boundaries, but only connect states within regions;
- Rule 3: refined transitions crossing a region boundary completely partition the old entry/exit transitions of the original unrefined state;
- Rule 4: refined transitions within a region completely partition the old self-transitions of the original unrefined state.

Rule 1 is the fundamental rule, which preserves the hierarchy of state abstractions. It confirms McGregor’s second rule of statechart refinement [15]. It disallows the introduction of new external states, so rules out Cook and Daniels’ refinement by extension (such as M4) [17]. Rule 2 defines limits on state retargeting for new methods, with respect to the chosen compatibility relationship. Section 5.4 discusses how these two rules relate to strong simulation. Rule 3 captures all of Cook and Daniels’ rules about transition splitting and retargeting within a superstate (a region, in the approach advocated here). The important generalization is the complete partitioning of transitions, which ensures that the set of new transitions behaves exactly like the old single transition. Rule 4 is a similar rule to ensure that self-transitions are preserved explicitly in the refinement. These two rules essentially describe the faithful replication of transitions for states that have been partitioned. They ensure that the refined machine is a non-minimal simulation of the original machine.

2.4. Comparisons with other refinement approaches

Under the assumptions of data refinement, the four rules enforce a strict behavioural consistency between the refined and original state machines, analogous to strong simulation (see Section 5.4) in which the existing behaviour is guaranteed in all future contexts of usage (no deadlocks). This is stronger than some other trace-based models of consistency, which only look at model executions in the absence of a theory of state and state generalization. The invocational consistency of Ebert and Engels [19] requires the subtype to contain all the traces of the supertype. This is equivalent to Cook and Daniels’ position [17], described above, which only guarantees existing behaviour in the original context of usage. Ebert and Engels’ observational consistency is weaker still, since it merely requires all the supertype’s traces to be derivable by censoring the subtype’s traces to remove methods that were introduced in the subtype [19].

Other work in this area has independently arrived at similar conditions for data refinement, using formal models of statecharts. Rules 1, 3 and 4 would seem to correspond closely to the adequacy conditions of Lano et al. [20]. Rule 2 would seem to correspond to Fiadeiro and Maibaum’s locality condition [21]. Complete partitioning is a general condition for adequacy and can be derived as a necessary condition for the theory of a subclass statechart to validate all the axioms of a superclass statechart. It is related to the method subtyping rule [22], which requires the precondition of a refined operation to be no stronger than the precondition of the operation that it replaces [23].

The data refinement model requires certain reasonable assumptions to be made. Methods are terminating, rather than divergent. If this were not so, the introduction of a non-terminating method in a subclass could break a system that used the subclass in the parent’s context. (Arguably, this would also break Rule 2, since the transition would have no destination.) Objects execute within a single process. If this were not so, a subclass object might not be able to respond to a parent method if it were already livelocked in a parallel process. Other process algebra models with refusals and divergences would be needed to handle these more difficult cases. Other kinds of refinement are possible, including changing the granularity of transitions, splitting one transition into a chain that achieves the same behaviour or distributing the behaviour of one object over several objects. This requires further rules to relate the composition of behaviours back to a façade object (in the sense of the Façade design pattern [24]) that could be substituted according to these rules. In general, the transitions in object-oriented software are complex, involving messages to further objects, such that the behaviour of a single object depends on many others. Nonetheless, provided that the states represented in a statechart are directly derivable from the data that an object controls (this accounts for simple and façade objects) then the above rules will apply.

3. THE GENERATION OF COMPLETE UNIT TEST-SETS

In state-based specification and testing approaches [15,25–27], the notion of complete test coverage is based on the exhaustive exploration of states and transitions. However, the nature of the guarantee obtained after testing varies from approach to approach. The following is an object-oriented adaptation of the X-machine testing method [27,28], 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. State-based specifications are developed for individual object types and abstract interface types. Testing may then be carried out with respect to these specifications. Of particular interest is the question whether a complete test-set generated for a given type may be used to test a behaviourally compatible subtype.

3.1. State-based specification

The state-based specification of an object type presumes that the object exists in a series of states. These are chosen by the designer to reflect modes in which the object’s methods react differently to the same message stimuli (the property of having state is due to state-contingent response and has nothing to do with the object having quiescent periods [11]). The possible states of an object type ultimately derive from different ways of partitioning the Cartesian product of its attribute domains [14,15]. More abstractly, the states of an interface type may be characterized as a complete partition of the product of its (abstract) access methods. Since access methods yield a projection on the concrete state of any compatible object, the partitions formed by abstract states are bound to include the object’s concrete states. This motivates the establishment of formal compatibility relations between abstract interface machines and the concrete object machines, which refine them.

Figure 4 illustrates the state-based specification for a Stack. This could be considered as specifying a concrete object type, a linked Stack existing in the Empty and Normal states. The preferred interpretation is that it specifies an abstract interface type, existing in at least these two states, which
are subject to refinement in subtypes. A syntactically correct state transition diagram has a unique transition to the initial state (to Empty, in Figure 4) and may have one or more transitions to a final state, a mode in which all behaviour is terminated (in Figure 4, the pop transition from Empty raises an exception, so leads to a final error state denoting a corrupted object representation). For completeness, the state transition diagram must define a transition for each method in every state. However, suitable conventions may be adopted to simplify the drawing of the diagram, in particular, to establish the meaning of missing transitions. Figure 4 is a simplified diagram, in which the omitted transitions for access methods size, empty and top may be inferred implicitly as self-transitions in every state.

It must be possible to determine the desired behaviour of the object, in every state, and for each method. If more than one transition with the same method label exits from a given state, the machine is non-deterministic. Qualifying the indistinguishable transitions with mutually exclusive, exhaustive guards will restore determinism (in Figure 4, ambiguous pop transitions exiting the Normal state are guarded). Certain design-for-test conditions may apply, to ensure that an exemplar test object can be driven deterministically through all of its states and transitions [27]. 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.

### 3.2. State-based test generation

When testing from a state-based specification, the objective is to drive the OUT into all of its states and then attempt 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 specification, to pass the tests. It is assumed that the specification 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. These notions are formalized below.

The alphabet is the set of methods \( m \in M \) that can be called on the interface of the OUT (including all inherited methods). The OUT responds to all \( m \in M \), and to no other methods (which are ruled out by the syntactic checking phase of the compiler). This puts a useful upper bound on the scope of negative testing.
The OUT has a number of control states \( s \in S \), which partition its observable memory states. A control state is defined as an equivalence class on the product of the ranges of the OUT’s access methods. If a subset \( A \subseteq M \) of access methods exists, then each observable state of the OUT is a tuple of length \(|A|\). Formally, tuples fall into equivalence classes under exhaustive, disjoint predicates \( p : Tuple \rightarrow \text{Boolean} \), where each predicate \( p \) corresponds to a unique state \( s \in S \). In practice, these predicates are implemented as external functions \( p : \text{Object} \rightarrow \text{Boolean} \) invoked by the test harness upon the OUT: \( \text{Object} \), which detect whether the OUT is in the given state using some combination of its public access methods.

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

In common with other state-based testing approaches, the state cover is determined as the 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. An initial test-set \( T^0 \) aims to reach and then verify every state. States are verified by executing every state predicate for each reached state, expecting only one predicate to yield true and the rest false. This is accomplished by computing test sequences: \( C \otimes P \), where \( \otimes \) is the concatenated product, which appends every sequence in \( P \) to every sequence in \( C \),

\[
T^0 = C \otimes P
\]  
(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 extra 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 again validated by concatenating all singleton sequences \( \langle p \rangle \in P \),

\[
T^1 = (C \cup C \otimes M^1) \otimes P
\]  
(2)

An even more sophisticated test-set \( T^2 \) aims to reach every state, fire every single transition and also every possible pair of transitions from each state. This is constructed from the switch cover, an augmented set of sequences \( K^2 = C \cup C \otimes M^1 \cup C \otimes M^2 \), which explores every single transition and every pair of transitions from each state, and, as above, the product with \( P \) to verify all reached states. In this, the extra product term \( C \otimes M^2 \) denotes the attempted firing of all pairs of transitions from every state,

\[
T^2 = (C \cup C \otimes M^1 \cup C \otimes M^2) \otimes P
\]  
(3)

In a similar fashion, further test-sets are constructed from the state cover \( C \) and low-order languages \( M^k \subseteq M^* \). The reached states are always verified using \( \langle p \rangle \in P \), for which exactly one should return true, and all the others false. The desired Boolean outcome is determined from the model. Each test-set subsumes the smaller test-sets of lesser sophistication in the series. In general, the series can be factorized and expressed for test-sets of arbitrary sophistication as

\[
T^k = C \otimes (M^0 \cup M^1 \cup M^2 \cup \cdots \cup M^k) \otimes P
\]  
(4)
For the Stack shown in Figure 4, the alphabet \( M = \{ \text{push}, \text{pop}, \text{top}, \text{empty}, \text{size} \} \). Note that \text{new} is not technically in the method-interface of Stack. It represents the default initial transition, executed when an object is first constructed, which in the formula is represented by the empty method sequence \( \langle \rangle \). The smallest state cover \( C = \{ \langle \rangle, \langle \text{push} \rangle \} \), since the ‘final state’ can be treated as a required exception raised by \text{pop} from the \text{Empty} state. Other sequences are calculated as above. Test-sets generated from this model may be used to test any Stack implementation that has identical states and transitions, for example, a \text{LinkedStack}, which uses a linked list to store its elements.

### 3.3. Test completeness and guarantees

The test-sets produced by this formula have important completeness properties. For each value of \( k \), specific guarantees are obtained about the implementation, once testing is over. The set \( T^0 \) guarantees that the implementation has \textit{at least} all the states in the specification. The set \( T^1 \) guarantees this, and that a \textit{minimal} implementation provides exactly the desired state-transition behaviour. The remaining test-sets \( T^k \) provide the same guarantees for \textit{non-minimal} implementations, under weakening assumptions about the level of duplication in the states and transitions.

A \textit{redundant} implementation is an implementation where a programmer has inadvertently introduced extra ‘ghost’ states, which may or may not be faithful copies of states desired in the specification. Test sequences may lead into these ‘ghost’ states, if they exist, and the OUT may then behave in subtle unexpected ways, exhibiting extra transitions, 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 \textit{positive} and \textit{negative} testing are achieved. As an example of the latter, it is confirmed that sequences of access methods do not inadvertently modify object states. Testing avoids any \textit{uniformity assumption} [29], since no conformity to type need be assumed in order for the OUT to be tested. Likewise, testing avoids any \textit{regularity assumption} that cycles in the specification necessarily correspond to implementation cycles. When the OUT ‘behaves correctly’ with respect to the specification it 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. Testing demonstrates full conformity up to the level of abstraction described by the control states.

The state-based testing approach described here is an adaptation of the X-machine approach for complete functional testing [27,28], replacing input/output pairs with method invocations. The need for ‘witness values’ in the output is eliminated by the guaranteed binding of messages to the intended methods in the compiler. The test generation method adapts Chow’s \( W \)-method for testing finite state automata [25]. In Chow’s method, states are not directly inspectable. Instead, reached states are verified by attempting to drive the implementation through further diagnostic sequences chosen from a characterization set \( W \subseteq M^* \), each state uniquely identified by a particular combination of diagnostic outcomes. Here, it is guaranteed that the OUT’s state is inspectable, since it must be characterized by some partition of the product of the ranges of its access methods.
4. OBJECT REFINEMENT AND TEST COVERAGE

The notion of behaviourally-compatible refinement introduced in Section 2 applies equally to the realization of interfaces in the UML sense that a concrete class implements an abstract interface [11] and also to the specialization of object subtypes. In both cases, the notion of refinement is explained in terms of deriving a more elaborate state transition diagram by subdividing states and adding transitions to a basic diagram. This paper also considers that an incremental modification of an object that preserves all of its existing behaviour, in the sense of XP’s incremental design with refactoring [5,6], constitutes a refinement in the same sense. Incremental modification in response to new requirements typically replaces simple solutions with more complex alternatives, having more states and transitions.

At the unit-testing level, individual OUTs tend to become more complex. It is also possible, when refactoring an entire subsystem [5,30] for certain objects to become simplified, either at the expense of introducing new objects, or by shifting the complexity onto other objects or by deleting unnecessary code. This is not considered in any detail here, because it usually involves altering interfaces at the unit level. However, if a collection of objects is refactored behind a façade, this interface can be verified according to the theory of compatibility.

4.1. Refinement of more concrete specifications

Figure 5 illustrates the object statechart for a DynamicStack, an array-based implementation of a Stack. This is taken to model a concrete refinement of the abstract Stack illustrated in Figure 4, similar to the UML notion of realizing an interface with a type that provides all the required methods. To demonstrate that the DynamicStack can be plugged safely into a system that requires at least an abstract Stack, the statechart in Figure 5 must be proven compatible with the statechart in Figure 4 in accordance with the behavioural theory. Alternatively, if Figure 4 is taken to represent the states and transitions of a concrete Stack with a linked list-based implementation, then Figure 5 can be understood as a change in implementation policy, similar to incremental redesign in XP. In this case, the modified object must also be shown to be substitutable for the original.

The main difference between the DynamicStack and the earlier Stack statechart is that the old Normal state, now only shown as a dashed region, has been partitioned into the states {Loaded, Full}, in order to model the dynamic resizing of the DynamicStack (push will behave differently in the Full state, triggering a memory reallocation). This is a complete partition (no other substate of Normal exists), so Rule 1 is satisfied. No new methods are introduced, so Rule 2 is not applicable. This is a characteristic of refinements that exactly satisfy an abstract specification. Even though no new methods are added, the partitioning of states will result in the splitting of transitions for the existing methods.

The Normal state’s old entry and exit transitions now cross over the region boundary, reaching the exposed Loaded state. The new pair of push, pop transitions exactly replaces the old pair (without splitting), so Rule 3 is satisfied. The Normal state’s old self-transitions are now replicated inside the region, as a consequence of splitting the state. The former push transition is first split in two (one replication for each new state) and then the transition from Loaded is split again, with exclusive guards on size. Similarly, the former pop transition is replicated for each new state and its former guard: size() > 1 is preserved in both states; however, the guard need not be notated in the Full state, as there is no other conflicting pop transition. So, Rule 4 is also satisfied. The refined DynamicStack implementation (in Figure 5) is therefore compatible with the original Stack interface’s behaviour (in Figure 4).
4.2. Regression test coverage of concrete realizations

Next, the issue of regression test coverage is considered. 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 4. This robustly guarantees the correct behaviour of a simple LinkedStack implementation with $S = \{\text{Empty}, \text{Normal}\}$, even in the presence of ‘ghost’ states. $T^2$ will include one sequence $\langle \text{push}, \text{push}, \text{push}, \text{isNormal} \rangle$, which robustly exercises $\langle \text{push}, \text{push} \rangle$ from the Normal state and will even detect a ‘ghost’ copy of the Normal state. A strong guarantee of correctness after testing may therefore be given for a LinkedStack implementation.

In classical regression testing, saved test-sets are reapplied to modified or extended objects in the expectation that passing all the saved tests will provide the same level of confidence in the modified object after testing. If the Stack’s $T^2$ test-set (related to the switch cover criterion) were reused to test a DynamicStack constructed with $n \geq 3$, so having all the states $\{\text{Empty}, \text{Loaded}, \text{Full}\}$ and all the transitions shown in Figure 5, the resizing push transition would never be reached, since this requires a sequence of four push methods. It would appear to the tester 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. This provides initial evidence that regression testing is less useful than generally supposed. Note that the testing guarantee that applied in the original case is broken, despite the intention not to modify any of the Stack’s abstract behaviour in the refinement. This is because the state-space has increased, resulting in more transitions to cover in the refinement.

4.3. Refinement of extended and subtype specifications

A common refinement in object-oriented programming is subclassing, in which the subclass extends its parent class and adds extra attributes and methods. The statechart for the subclass is a refinement of the parent statechart, in which the extra attributes and methods result in a partitioning of states and the introduction of new transitions for the extra methods. If the subclass’s statechart can be proven
compatible with the parent’s statechart, then it describes a behavioural subtype whose instances may be safely substituted where the parent type was specified. However, subclass extensions may break the conditions for subtyping, if they change the semantics of the parent’s methods. This is easily detected in the theory of behavioural compatibility.

Figure 6 illustrates the development of a class hierarchy leading to concepts such as the loaned items in a lending library. The upper state machine describes the abstract behaviour of a Loanable entity, which oscillates between its Available and OnLoan states. The lower state machine describes a LoanItem entity that extends the Loanable entity. This is a product machine (see Section 2.3) with four states, resulting from the concurrent composition of the Loanable machine with a supplementary Reservable machine (not illustrated) that, it may be inferred, oscillates between Unreserved and Reserved states. The resulting four states are named \{OnShelf, PutAside, NormalLoan, Recalled\}. However, in LoanItem, the behaviours of loaning and reserving have been made dependent on each other.

The refined LoanItem machine must be checked for behavioural compatibility with the abstract Loanable machine. The four LoanItem states completely partition the two states of Loanable, so Rule 1 is satisfied. The new methods \{reserve, cancel\} introduced in LoanItem stay within the prescribed region boundaries, so Rule 2 is satisfied. This is a characteristic of subclass refinements that orthogonally compose the behaviour of the parent (Loanable) with the behaviour of the incremental extension (the putative Reservable, a kind of mixin-class [31]). However, elsewhere in this example,
behaviours are not completely independent. In the subclass, the borrow method has been redefined to take reservations into account.

Considering now the splitting of transitions required by Rule 3, whereas return has correctly been split as a consequence of partitioning OnLoan into two states {NormalLoan, Recalled}, the refinement of the old borrow transition is more problematic. One partial transition from OnShelf allows the loan to go ahead. The other partial transition from the PutAside state is guarded, and only succeeds if the LoanItem is borrowed by the same person who reserved it previously. While such behaviour is reasonable, it makes LoanItem incompatible with Loanable. The refinement of the borrow transition breaks Rule 3, since the partials are not a complete partition of the original. From Loanable’s perspective, borrow always succeeds from the Available state whereas it sometimes fails for a LoanItem.

This illustrates how easy it is to make an adaptation in a subclass that breaks subtyping. In formal terms, the redefined borrow has strengthened the precondition of the method it replaces, which violates operation refinement. Strengthened preconditions can be related to covariant argument redefinition [23], which is known to break subtyping [32]. In this situation, a designer can either decide to accept that the subclass is not compatible with the parent class and so refuse to allow programs to alias a LoanItem through a Loanable variable (this can be enforced using private inheritance in C++, for example). Alternatively, the specification of Loanable can be revisited and generalized to permit the desired compatibility, which may be restored by adding a borrow transition from the Available state to itself, in the Loanable abstract class, indicating the anticipated null operation. The abstract state machine is then non-deterministic, since the choice of the successful or failing borrow transition cannot yet be determined (it depends on information revealed in the subclass). This is formally correct, and corresponds to axiomatic treatments in which the supertype is left deliberately underspecified [23]. The programmer may still provide the obvious default implementation of borrow in Loanable which always succeeds, since this satisfies the specification.

4.4. Regression test coverage of extended objects

Next, the extent of the coverage of extended objects provided by saved regression tests is examined. The subtype-incompatible refinement of Figure 6 will be considered, without the suggested retrospective modification. Assuming that a $T^2$ test-set is generated from the Loanable specification in Figure 6, this will robustly confirm whether borrow and return succeed and fail correctly for a Loanable instance, even in the presence of ‘ghost’ versions of the OnLoan and Available states. Strong guarantees of correctness are offered by a successful test outcome.

However, when the same tests are reapplied to the extended LoanItem, they will only cover half of the partitioned states. The saved $T^2$ test-set includes the sequences: {⟨isAvailable⟩, ⟨borrow, isOnLoan⟩, ⟨return, exception⟩, ⟨borrow, return, isAvailable⟩, . . . } and naturally no sequence will contain reserve or cancel, which are first introduced in the subclass’s protocol. The saved test-set will therefore oscillate between the states {OnShelf, NormalLoan} and will not reach the states {PutAside, Recalled}. Because of this, only half of the borrow and return transitions will be exercised in the refinement, compared to all of them in the original. This runs counter to the intuitions of programmers, who tend to believe that regression tests exercise all of the parent’s functionality in the subclass.

Consider how serious this is. Partitioning states always results in splitting transitions. Every pair of methods such as {borrow, return} and {reserve, cancel} introduces further partitions in every
existing state. The proportion of the original transitions that are still covered by a saved test suite falls off as a geometrically decreasing fraction in each successive refinement. Contrary to popular expectations that recycled regression tests positively confirm the parent’s behaviour in the subclass, regression tests actually cover significantly less of the parent’s state-space in each successive subclass and may eventually cover only a trivial portion. To re-emphasize the point, the failure exposed here has nothing to do with testing new methods that are introduced in the subclass. It is clear that regression tests are not designed to do this. Rather, the issue is that regression tests fail to test the original behaviour.

Finally, this example also demonstrates how regression testing cannot reliably determine whether a refinement is compatible. If the designer expected borrow always to succeed, as it does in Loanable, then no amount of testing with any saved test-set will reveal that borrow sometimes fails in LoanItem. To reveal this requires a particular interleaving of methods from Loanable and LoanItem. In practice, programmers write extra tests to do this, although manual test creation is risky since the appropriate interleaving may not be discovered. The point here is that regression tests alone are insufficient.

5. REGRESSION TESTING VERSUS TEST REGENERATION

The weakness in conventional regression testing comes from recycling saved test-sets as a whole. In object-oriented testing, this culture goes back to the parallel design and test architecture [1,2] (see Section 1), in which test-suites are saved as methods of the THO and are inherited for reuse as a whole. Likewise, supplementary test-sets for the refined object are encapsulated and may not interact fully with the original behaviour. The prospect of reusing whole test-suites is so beguiling that it is hard to refuse, especially after the effort invested in developing the tests in the first place. In test-driven development with JUnit [3,4], test scripts are saved and recycled as a whole, in the expectation that this provides protection against the effects of entropy in modified code. However, the theory of object state refinement presented in this paper (see Section 2 above) predicts that the entropy of modified objects may increase, and not be detected, despite their passing all the saved tests.

5.1. Overestimation of regression test coverage

Programmers tend to overestimate the effectiveness of regression testing. Naturally, they do not expect the regression tests to exercise any new features introduced in the refinement. For this, they develop additional tests, exercising the new features alone and in combination with old features. However, they do expect the regression tests to exercise all of the original features adequately, as demonstrated in some of the citations in Section 1 above. In talking to testers in industry, the author has found this fallacy to be quite persistent.

One explanation for this could be that programmers have a different mental model of how modifications and extensions affect the state-space. This can be ascribed to an impoverished view of refinement that corresponds to the discredited model M4 (see Section 2.2 above), in which new external states and transitions are added in a linear fashion to the original machine. If this were the appropriate model, then clearly the saved tests would appear to cover all of the parent’s state-space in the subclass. In reality, the state-space of the parent is partitioned in the subclass rather than extended, more like the model L2 (see Section 2.3 above). This is what causes the progressive loss of coverage, as parent transitions are split into partials of which at most half are exercised in the subclass (and possibly fewer than this, depending on how many times the state-space is partitioned by new methods).
Unfortunately, recycled test-sets always cover significantly less of the parent’s state-space in the subclass than they did in the parent. As the state-space of the modified or extended object increases, by a factor of two for each state subdivision, the confidence offered by retesting is progressively weakened. The saved tests only cover a half, a quarter, an eighth and so on, of the original object’s state-space in each successive refinement. This seriously undermines the validity of popular regression testing approaches, which are based on the reuse of inherited test-sets [1,2] or saved and recycled test suites [3,4].

Given the geometric decrease in coverage, it is difficult to quantify how useful the saved test-sets are, if regression tests are all that is available. One possible measure is the static coverage of the saved test-set, expressed as the fraction of the original state-space covered in the refinement, by the same tests. However, this ignores the dynamic usage of the system. Instead, a weighted statistical coverage of the saved test-set may be calculated from the frequency that saved tests exercise used parts of the refined system, based on collecting sets of traces from live usage of the refined system. This kind of reassurance is questionable, since testing is incomplete and offers no useful guarantees.

5.2. Completeness of regenerated test sets

To achieve complete coverage of the refined machine, it is vital to test systematically all the interleavings of new methods with the inherited methods, so exploring the refined state-transition diagram completely. This simply cannot be done reliably by human intuition and manual test-script creation. Instead, the complete test-sets for refined object types, such as the DynamicStack or the LoanItem introduced in Section 4, must be regenerated from scratch, using an algorithm based on the formula from Section 3. If programmers are willing to accept even simple specifications that idealize the behaviour of an object as a finite state machine then this process can be fully automated, generating test-sets to the desired $T^1$, $T^2$, $T^3$, . . . confidence levels. This test-generation approach has been used successfully in testing automotive systems with hundreds of thousands of states and millions of transitions [33].

Regenerated tests are not regression tests in the conventional sense, but all-new tests in which the state-space of the refined OUT is fully explored. The tests must be regenerated from scratch, because there is no predictable way in which the regenerated sequences could be derived by extending the original test sequences, without reference to an exploration of the derived object’s state-space. For convenience’s sake, the additional tests required to test the refined object could be isolated a posteriori by computing the set difference of the two test sets $T^k(\text{Refined}) \setminus T^k(\text{Basic})$, for some confidence level $k$, but this would still require computing the refined set $T^k(\text{Refined})$ from scratch. So, the cheapest policy is to regenerate all-new tests for each refined specification.

However, regenerated tests do satisfy the expectations of regression testing, in that they test up to the same confidence-levels as the original tests. In common with all test-sets generated using the object X-machine method, regenerated tests provide specific guarantees for specific amounts of testing. Because the test-sets are generated systematically, the tester may choose whether to test using $T^1$, $T^2$, $T^3$, . . . , etc. up to the desired level of $k$ in the test generation formula (see Section 3.2). The significance of this is that the same levels of guarantee may be provided for both the original and retested objects, something that is not possible with conventional regression testing using recycled test-sets, for which the level of confidence is progressively weakened in each new context.

Regenerating a test-set works equally well, whether or not the OUT is a behaviourally compatible refinement of some original object, since the test-set is derived directly from the refined specification.
5.3. Determining behavioural compatibility

According to the opinions cited in Section 1, a major goal of regression testing is to provide reassurance that a modified object is still behaviourally compatible with its unrefined precursor. Regression testing seeks to confirm this by testing the refined object, until it passes the saved tests [2,3,5–7]. However, this expectation was shown to be fallacious. Section 4.3 demonstrated how a subclass (LoanItem) may violate the behaviour of its parent (Loanable), by the redirection of just one of its transitions away from the semantically-consistent target state (one of the three partials of the borrow transition). The regression tests could not detect that the semantics of the object had changed (see Section 4.4). However, the rules for behavioural compatibility were able to determine this (see Section 4.3).

This paper seeks to turn a number of regression-testing concepts on their head. Compatibility cannot be assured directly through re-testing a modified object up to the original specification, as is conventionally expected. Instead, compatibility can be assured by a combination of testing up to the refined specification and then by verifying that the refined specification is compatible with the original specification in a formal model. Figure 7 shows the difference between the two philosophies.

Compatibility is defined in this paper as a verifiable refinement relationship between two object specifications. Testing only has the power to confirm that each OUT conforms to its own specification, using a specific test-set generated from that specification (the B-test and R-test sets in Figure 7).
The refined OUT is then proved compatible with the basic specification by virtue of the transitive composition of the testing relationship (R-test conforms, in Figure 7) and the verification relationship (refines, in Figure 7). This is a novel combination of formal verification and practical testing.

The general rules for verifying compatible statechart behaviour were presented in Section 2.3. These rules provide a framework for comparing earlier work. A scale of strictness may be established, from the weakest approach of trace censoring (cf. Ebert and Engels’ observational consistency [19]), via the intermediate trace inclusion of Cook and Daniels [17] and classical automata theory [9] (cf. Ebert and Engels’ invocational consistency [19]), up to the strictest state and transition partitioning approach adopted here, which includes McGregor’s refinement rules [14,15], the adequacy conditions of Lano et al. [20] and Fiadeiro and Maibaum’s locality condition [21]. The difference between weak and intermediate compatibility is the collapsing, versus the preservation of transition paths. The key difference between intermediate and strict compatibility is the strict insistence that new states are partitions of existing states.

Strict compatibility has important practical consequences, where objects are aliased simultaneously by handles of many types. This is, in fact, quite common in object-oriented design, where generic algorithms are factored into parts introduced at different levels in the inheritance hierarchy (see the Template Method design pattern [24, p. 335]). In this context, an object may be manipulated by more than one protocol and the messages from the different protocols may be interleaved, which may sometimes cause deadlocks [34,35].

5.4. Comparison with weak and strong simulation

The difference between strict and intermediate compatibility may be explained in terms of concepts in process calculus. Cook and Daniels’ examples [17] of statechart refinement are equivalent to the classical refinement of automata, which judges behavioural compatibility by trace inclusion [9,19]. Milner’s $\pi$-calculus [18] defines compatibility more subtly, taking into account the effects of non-deterministic choice and invisible $\tau$-actions. Of particular interest is the distinction between weak and strong simulation. An automaton weakly simulates another, if its observed behaviour is identical, under the null assumption about invisible $\tau$-actions (namely that they do not affect behaviour). An automaton strongly simulates another if its observed behaviour is identical in all contexts, irrespective of the $\tau$-actions’ unseen behaviour.

Suppose that a subtype object is aliased through a supertype handle. An analogy with $\pi$-calculus may be drawn by comparing the supertype’s protocol to visible actions, and the subtype’s additional protocol to invisible $\tau$-actions (from the supertype’s point of view). So long as the subtype object is manipulated only through the protocol of the supertype, it will behave in a conformant way (under both intermediate and strict compatibility). However, if the subtype object is also aliased by a subtype handle, it may be manipulated in ways that are unforeseen by the supertype. Section 2.2 above showed how an M4 object (see Figure 2) could be manipulated through the protocol of M0, until it receives $\langle c \rangle$ through the M4 protocol, at which point the M0 protocol deadlocks. M4 is therefore not strictly compatible with M0, although it clearly includes the traces of M0. For this reason, M4 only weakly simulates M0. By contrast, in Section 2.3 the L2 object strongly simulates L0 (see Figure 3). It always behaves in the identical way to L0, even under arbitrary interleaving of messages from the L1 protocol (which are invisible from L0’s point of view).

The strict behavioural compatibility described in this paper is therefore similar to strong simulation, because the protocol of the supertype is preserved, under arbitrary interleaving of subtype protocols.
This is achieved by making sure, in refinement Rules 1 and 2, that invisible actions cannot force a refined object into a state that is unrecognized by its supertype’s protocol. The rules are therefore normative, since strong simulation immediately follows. This is a significant advantage for designers, who may use the statechart refinement rules without reference to a more detailed trace analysis.

5.5. Conclusion: guarantees of repeatable quality

The goal of all verification and testing is to assure the quality of the software end product. The analysis developed in this paper has seriously undermined the validity of conventional regression testing as a quality control mechanism to confirm behavioural properties. Regression testing can neither assure the conformant behaviour of modified or extended objects up to some original specification, nor can it assure the correctness of inherited or preserved functionality in extended objects. This is because the recycled tests exercise significantly less of the original specification in the refinement than they did in the unrefined precursor, such that modified and re-tested objects may be considerably less secure for the same testing effort.

By contrast, the theory of statechart refinement provides a strong guarantee of behavioural compatibility. It is more expressive than Liskov and Wing’s behavioural subtyping [16], which adds new transitions but admits no new states [34]. It eliminates protocol deadlock under polymorphic aliasing, a stronger guarantee than that provided by some earlier statechart refinement approaches [17,19]. For non-divergent processes, the new rules are equivalent to Milner’s strong simulation [18], guaranteeing conformant behaviour notwithstanding any future actions taken in the protocols of subtypes. Having a practical means of guaranteeing behavioural compatibility is a relatively new benefit for object-oriented programming, which conventionally judges compatibility only in terms of interface matching.

To satisfy the requirement for a method that can test whether an object conforms to some supertype specification, a new approach was developed. Complete functional test-sets are regenerated systematically from each refined or extended statechart specification. Conformity to a supertype specification is then assured by a combination of rigorously testing up to the refined specification and then verifying in the theory that this conforms to the supertype specification. In the test regeneration approach, it is possible to provide strong guarantees for specific levels of confidence in the OUT once testing is over. Furthermore, after the OUT has been refined, the same levels of confidence may be retained after re-testing using fully regenerated test-sets. This notion of guaranteed, repeatable quality is a new and important concept in object-oriented testing.

ACKNOWLEDGEMENTS

This research was undertaken as part of the MOTIVE project (Method for Object Testing, Integration and Verification), supported by the U.K. Engineering and Physical Sciences Research Council (EPSRC) under grant number GR/M56777.

REFERENCES

Copyright © 2006 John Wiley & Sons, Ltd.


