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, United Kingdom
A.Simons@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/~ajhs/

Abstract. This paper presents a *behavioural* theory of object compatibility, based on the refinement of object states. 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 standard practices in regression testing are inadequate. Functionally complete test-sets that are applied as regression tests to subtype objects are usually expected to cover the state-space of the original type, even if they do not cover transitions and states introduced in the subtype. However, such regression testing is proven to cover strictly less than this in the new context and so provides much weaker guarantees than was previously expected. Instead, a retesting model based on automatic test regeneration is required to guarantee equivalent levels of correctness.

Keywords: Object-oriented, behavioural subtyping, state refinement, state-based testing, regression testing, test generation, testing adequacy.

1 Introduction

Practical object-oriented unit testing is influenced considerably by the *non-intrusive* testing philosophy of McGregor *et al* [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 presumed advantage is that test-sets can be inherited from the parent THO and applied, as a suite, to the child OUT, in a kind of regression test. The purpose of such retesting is to ensure that the child class still delivers all the functionality of the parent. The child THO will supply additional test-sets to exercise methods introduced in the child OUT [1, 2].

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 refined 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 makes a strong claim 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" [5]. There are two sides to this claim. Firstly, 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. Secondly, there is the implicit assumption that modified code which *passes* all the tests is still as secure as the original code. Tests are implicitly being used as guarantees of a certain level of correctness.

In this paper, we prove that the second assumption is unsound and unsafe. In section 2, a state-based theory of object refinement is presented, which encompasses object extension with subtyping, the concrete satisfaction of abstract interfaces and refactoring of implementations 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 sections 3 and 4, the theory also predicts that standard practices in regression testing are inadequate. Functionally complete test-sets that are applied as regression tests to subtype objects are usually expected to cover the state-space of the original object, even if they do not cover transitions and states introduced in the refinement. However, such regression testing is proven to cover strictly less of the original object's state space in the new context and so provides much weaker guarantees than expected. After passing the recycled tests, objects may yet contain introduced faults, which are undetected.

In place of the unsafe kinds of regression testing, we propose a new approach, which is based on automatically generating the tests from the refined object's state machine. The theory predicts that simply adding new test suites to the existing encapsulated suites does not achieve coverage. It is necessary to generate new test-sets from scratch, in which methods are interleaved in different orders than before, to obtain the same level of guarantee.

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. 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 modified machine. Object-oriented design methods [6, 7] include *object statecharts*, which are influenced by Harel's statecharts [8] and SDL [9]. 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

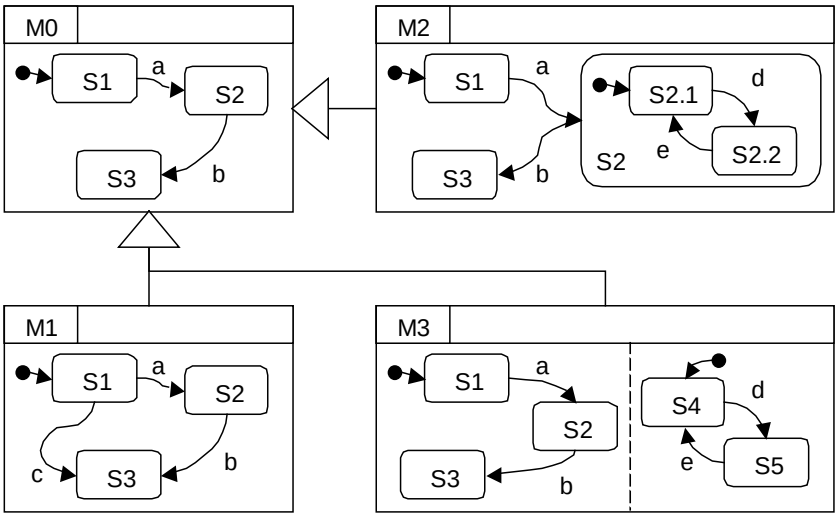

Fig. 1. McGregor's structural statechart refinements. The basic state machine *M0* is refined by the compatible machines *M1, M2, M3*. *M1* refines *M0* by adding an extra transition. *M2* refines *M0* by introducing substates. *M3* refines *M0* by introducing concurrent states.

McGregor et al. proposed one of the early theories of object statechart refinement [10, 11]. 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. 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 [12], that 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 [10], 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 *{<>, <a>, <a, b>, <a, d>, <a, d, b>, <a, d, e>, <a, d, e, b>, …}* and so include all the traces of *M0*.

M3 introduces concurrent states *S4*, *S5* and extra methods *d* and *e* which 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 we denote as: *{S1/4, S2/4, S3/4, S1/5, S2/5, S3/5}*. The traces of *M3* are infinite, but include *{<>, <a>, <d>, <a, d>, <d, a>, <a, b>, <d, e>, <a, d, e>, <d, e, a>, <a, d, b>,…}* and so include all the traces of *M0*.

2.2 Cook and Daniels' Statechart Refinements

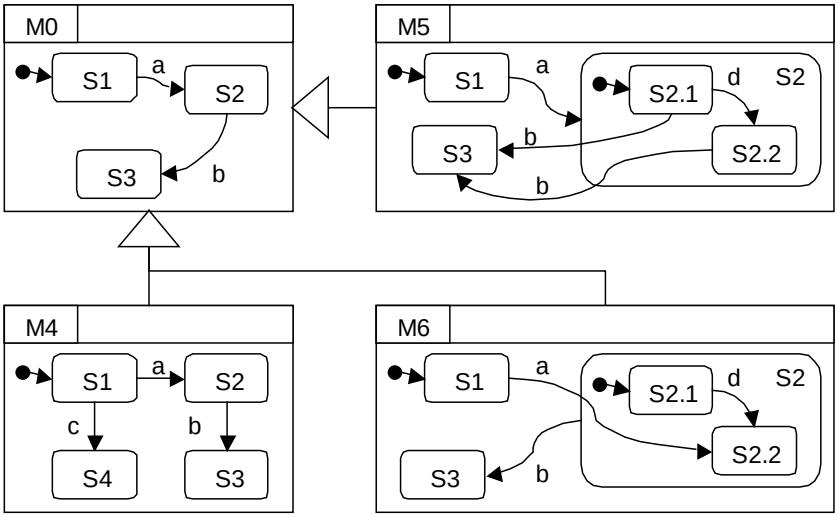

Fig. 2. Cook and Daniels' additional statechart refinements. As before, *M0* is the reference machine. *M4* refines *M0* by adding a transition to a new state. *M5* refines *M0* by transition splitting. *M6* refines *M0* by retargeting a transition.

In their Syntropy method [13], Cook and Daniels permit further extensions to statecharts. Their full set of refinements includes (p207-8): 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*. We sometimes refer to *S4* 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 [13] 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*. We choose to target *S2.2* 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 *M0*'s events. However, we argue 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 problem 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, and so their concatenation in the subtype gives rise to brand-new external states (like *M4* above)? Do these variables already exist *in virtuo* at the abstract level, in which case their concrete exposure in the subtype creates new substates (like *M2* above)? Are these variables orthogonal and so give rise to concurrent states in the subtype, equivalent to state products (like *M3* above)? These different views may be in conflict.

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 we must expect subtype objects to exhibit orthogonal behaviour at least some of the time, so the *M3* refinement is chosen 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 π-calculus process algebra [14], *M3 strongly simulates M0*, whereas *M4* only *weakly simulates M0*. This is discussed in section 5.4 below.

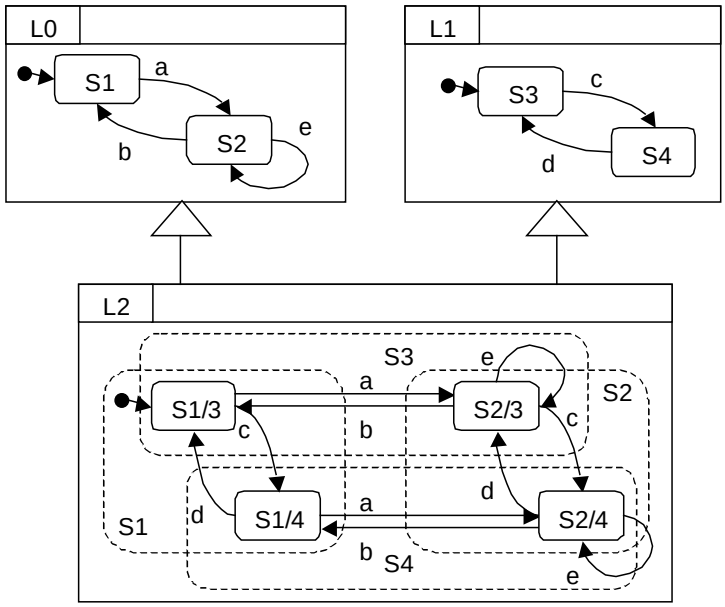

Fig. 3. The model of behaviourally-compatible refinement. *L2* is the refined statechart resulting from the concurrent composition of *L0* and *L1*, without respect to order. The states of *L0* and *L1* become intersecting regions in the refinement, which contains the product of states.

Since in general we must expect to support refinements like *M3*, 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 we were tempted to view composition as ordered, here we cannot. Accordingly, we cannot say that any particular superstate hierarchy is

more valid. So, we dispense with superstates and think instead 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 *S1* is turned into a pair of partial *a*-transitions from *S1/3* and *S1/4*. Because we are assuming orthogonal behaviour, these also target separate partitions of *S2*, the states *S2/3* and *S2/4*. 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 *S2*. No matter which destination states within region S2 we retarget, we should still be able to abstract away to *S2*. In all cases, the partial *a*-transitions would be merged in a single transition from *S1* to *S2*. Retargeting may select an arbitrary state, or combination of states within the destination region. Supposing now that the *c*-transition from *S1/3* were retargeted outside the *S1* region, to *S2/4*, within the different region *S2*. The *c* message now interacts unfavourably with the alternating behaviour of *a, b*. This means that a sequence <*c, a*> will deadlock from *S1/3*. While this modification is not compatible with *L0*, it is compatible with *L1*. Retargeting must therefore be considered with respect to the compatibility relation desired between specific machines.

From these considerations, we obtain the statechart refinement rules for behavioural compatibility. 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:

- 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 [11]. It disallows the introduction of new external states, so rules out Cook and Daniels' refinement by extension (such as *M4*) [13]. Rule 2 defines limits on state retargeting for new methods, with respect to the chosen compabitility relationship. In section 5.4 we show 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 our approach). The important generalisation 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 equivalent to the original machine.

Together, the four rules enforce a strict *behavioural consistency* between the refined and original state machines, analagous to *strong simulation* (see 5.4). 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 generalisation. The *invocational consistency* of Ebert and Engels [15] requires the subtype to contain all the traces of the supertype. This is equivalent to Cook, Daniels and McGregor's position, described above [11, 13]. 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 [15].

3 The Generation of Complete Unit Test-Sets

In state-based testing approaches [16, 17, 11, 18], it is possible to develop a notion of complete test coverage, based on the exhaustive exploration of the object's states and transitions. However, the nature of the guarantee obtained after testing varies from approach to approach. The following is an adaptation of the X-Machine testing method [18, 19], 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.

3.1 State-Based Specification

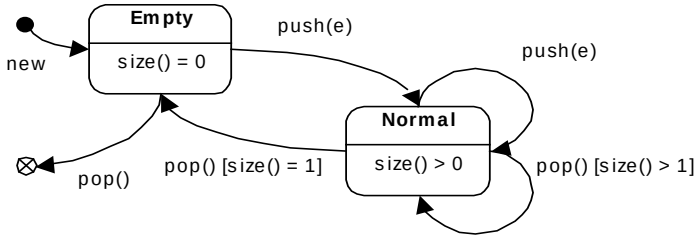

Fig. 4. Abstract state machine for a *Stack* interface. The two states (*Empty*, *Normal*) are defined on a partition of the range of the *size* access method. No self-transitions for access methods are notated, by convention, but all other transitions must be shown

We assume that the object under test (OUT) exists in a series of states, which are chosen by the designer to reflect modes in which its methods react differently to the same message stimuli (formally, the notion of state derives from state-contingent re-

sponse and has nothing to do with whether the object has quiesecent periods). The OUT is assumed to have a unique transition to its initial state and may or may not have a final state, a mode in which it is no longer useable, for example, an error state (representing a corrupted representation – see figure 4), or a terminated state (representing the end of the object's life history).

The states of an object derive ultimately from the product of its attribute variables, but can be characterised more abstractly as the product of the ranges of its access methods. Formally, we assume that states are a complete partition of this product. For completeness, a finite state model must define a transition for each method in every state. However, suitable conventions may be adopted to simplify the drawing of the state transition diagram, in particular, to establish the meaning of missing transitions. Figure 4 shows a simplified state machine for an abstract *Stack* interface, in which the omitted transitions for all the 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 nondeterministic. 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 the OUT can be driven deterministically through all of its states and transitions [18]. 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

The basic idea, when testing from a state-based specification, 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 formalised 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 $p : \mathit{Tuple} \rightarrow \mathit{Boolean}$, where each predicate p corresponds to a unique state $s \in S$. In practice, these predicates are implemented as external functions $p : \mathit{Object} \rightarrow \mathit{Boolean}$ invoked by the test

harness upon the OUT : *Object*, which detect whether the OUT is in the given state using some combination of its public access methods.

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 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. Verification is accomplished by concatenating every sequence in the state cover C with every predicate in the predicate language P, denoted: $C \otimes P$, where $\otimes$ is the concatenated product which appends every sequence in P to every sequence in C.

$$T^0 = C \otimes P \quad (1)$$

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 $<p> \in P$.

$$T^1 = (C \cup C \otimes M^1) \otimes P \quad (2)$$

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. 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 predicate. The product term $C \otimes M^2$ denotes the attempted firing of all pairs of transitions from every state.

$$T^2 = (C \cup C \otimes M^1 \cup C \otimes M^2) \otimes P \quad (3)$$

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 $<p> \in P$, 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 factorised and expressed for test-sets of arbitrary sophistication as:

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

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*. It represents the default initial transition, executed when an object is first constructed, which in the formula is represented by the empty method sequence <>. The smallest state cover C = {<>, <*push*>}, since the "final state" is really an exception raised by *pop* from the

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 *LinkedStack*, which uses a linked list to store its elements.

3.3 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. 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 *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 specification. 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* [20], 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. 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 [18, 19], 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 algorithm adapts Chow's W-method for testing finite state automata [16]. 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 characterisation set $W \subseteq M^*$, each state uniquely identified by a particular combination of diagnostic outcomes. Here, we know that the OUT's state is inspectable, since it must be characterised by some partition 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 *realisation of interfaces* (in the UML sense that a concrete class implements an abstract interface [7]) and also to the *specialisation 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. In this paper, we also consider that the need to *re-implement an object*, in the sense of XP's *refactoring* [5, 21], constitutes a refinement in the same sense. This is because modification typically replaces simple solutions with more complex ones, in response to new requirements. At the unit-testing level, individual OUTs tend to become more complex. (It is also possible, when *refactoring* an entire subsystem [21], for certain objects to become simplified, at the expense of introducing new objects, or shifting the complexity onto other objects, or by deleting unnecessary code – we do not consider this here).

4.1 Test Coverage of a Modified or Refactored Object

Figure 5 illustrates a refined object statechart for a *DynamicStack*, an array-based implementation of a *Stack*. We may either consider this to be a concrete realisation of the *Stack* interface of figure 4, or else a change in implementation policy, a refactoring of an old linked *Stack*. Firstly, we wish to confirm that the *DynamicStack* specification conforms to the abstract *Stack* specification in figure 4.

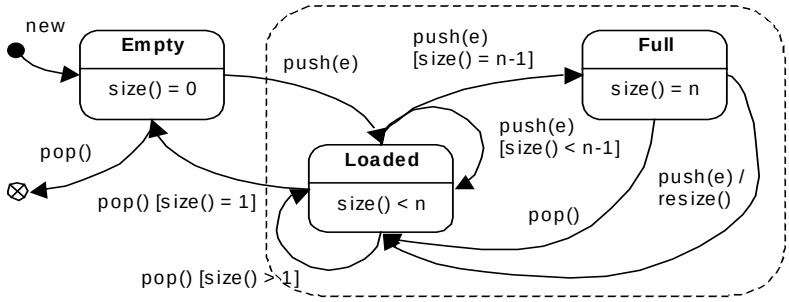

Fig. 5. Concrete machine for a *DynamicStack*, which realizes the *Stack* interface. The two states (*Loaded*, *Full*) partition the old *Normal* state in fig. 4, resulting in the replication of its transitions. The behaviour of *push* in the *Full* state must be tested

The main difference between the *DynamicStack* and the earlier *Stack* machine 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. 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).

Next, we consider the issue of test coverage. 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 = {*Empty, Normal*}, even in the presence of "ghost" states. T^2 will include 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. 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 guarantee the same level of correctness. If the *Stack's* T^2 test-set were reused to test a *DynamicStack* constructed with $n \geq 3$, so having all the states {*Empty, Loaded, 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. 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.

4.2 Test Coverage of a Subclassed or Extended Object

In more complex examples of subclassing, the refinement introduces new behaviour, which partitions all existing states. Figure 6 illustrates the development of an abstract class hierarchy leading to concepts like the loan items in a 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 with four states, resulting from the concurrent composition of the *Loanable* machine with a supplementary *Reservable* machine (not illustrated), which, we may infer, oscillates between *Unreserved* and *Reserved* states. The resulting four states are named {*OnShelf, PutAside, NormalLoan, Recalled*}. The behaviours of loaning and reserving are dependent on each other in interesting ways.

First, we check the refinement for compatibility. The four 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. Looking now at the splitting of transitions required by rule 3, while

return has been split by the partitioning of *OnLoan* into two states {*NormalLoan, Recalled*}, the *borrow* transition is more interesting. One partial transition from *On-Shelf* 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 the practical effect of breaking refinement rules. However, compabitility 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 nondeterministic, since the choice of the successful or failing *borrow* transition cannot yet be decided.

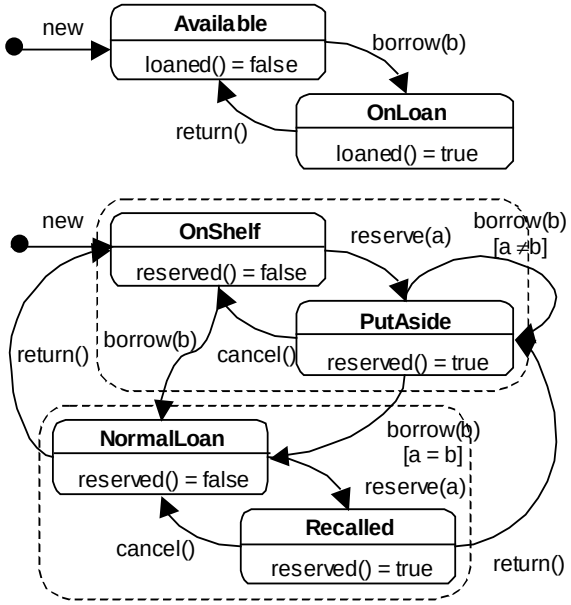

Fig. 6. The upper state machine captures the behaviour of a *Loanable* entity, with the methods *{borrow, return}*. The lower *LoanItem* machine extends this with reservations, combining the behaviour of *{borrow, return, reserve, cancel}*. The refined machine is not yet wholly compatible with the base machine, but this can be addressed

Next, we consider the issue of test coverage. Assuming that a T^2 test-set is generated from the *Loanable* specification in figure 6, this will robustly confirm that *borrow* and *return* succeed and fail correctly (for a *Loanable* instance), even in the presence of "ghost" versions of the *OnLoan* and *Available* states. 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 no

sequence will contain *reserve* or *cancel*, which are first introduced in the subclass's protocol. The test-set will therefore oscillate between the states {*OnShelf, Normal-Loan*} 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. Partitioning states always results in splitting transitions. Consider now that every pair of methods like {*borrow, return*} and {*reserve, cancel*} introduces further partitions in every existing state. The proportion of the original transitions still covered falls off as a geometrically decreasing fraction in each successive refinement. Contrary to popular expectations that recycled regression tests confirm the base object's behaviour in the refined object, regression tests actually cover significantly less of the base object's state space in each successive refinement.

5. Conclusions: Regression versus Regeneration

The weakness in conventional regression testing comes from recycling saved test-sets as a whole, rather than reconstructing test-sequences from scratch. 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 as a whole. 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. Likewise, in JUnit [3, 4], test scripts are saved and recycled as a whole, in the expectation that this provides a guarantee against the effects of entropy in the modified code.

5.1 Overestimation of Regression Test Coverage

Programmers do not expect regression tests to exercise the new features introduced in the refinement. For this, they develop additional tests, sometimes exercising the new features in combination with old features. However, they do expect the regression tests to exercise all of the original features completely. This corresponds to an impoverished view of refinement, as illustrated by the model *M4* (see section 2.2 above). The state space of a valid refinement is actually much greater, more like the model *L2* (see section 2.3 above).

Unfortunately, recycled test-sets always exercise *significantly less* of the refined object than the original. 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. To achieve the same level of coverage, 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.

5.2 Completeness of Regenerated Test Sets

In the proposed approach, the test-sets for refined object types, such as the *Dynamic-Stack* or the *LoanItem* introduced in section 4, should be regenerated entirely from scratch, using the algorithm from section 3. With even very simple object state machine specifications, this process can be automated, generating test-sets to the desired T^1, T^2, T^3… confidence levels.

The regenerated tests are not regression tests in the normal sense, but all-new tests in which the state-space of the refined OUT is fully explored. Regenerating the 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, not the original one. For this reason, the proposed re-testing approach is robust under all kinds of software evolution, whether this is by subclassing, by refactoring or by simple textual editing of the OUT, and works independently of behavioural compatibility. 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 from object state machines, 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 algorithm. 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 guarantees are progressively weakened in each new context.

5.3 Testing to a Repeatable Level of Quality

This paper turns a number of regression-testing concepts on their head. Conventional regression testing assumes that a refined object is compatible with its unrefined precursor, if it passes the same tests [2, 3, 5, 21]. This was shown to be false, in section 4 above. Compatibility cannot be assured directly through re-testing, but it can be proved indirectly by verification in a formal model. Figure 7 shows the different philosophies.

Compatibility is redefined as a verifiable *refinement* relationship between two object *specifications*. Each OUT may only be proven to conform to its own specification, by a specific test-set generated from that specification (the B-test and R-test sets in figure 7). The refined OUT is then only provably compatible with the basic specification by virtue of the transitive composition of the *R-test conforms* and *refines* relationships.

The strength of the guarantee obtained in conventional regression testing is badly overestimated. Recycled test-sets 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. By comparison, in the test regeneration approach, it is possible to provide specific guarantees for levels of confidence in the OUT. 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.

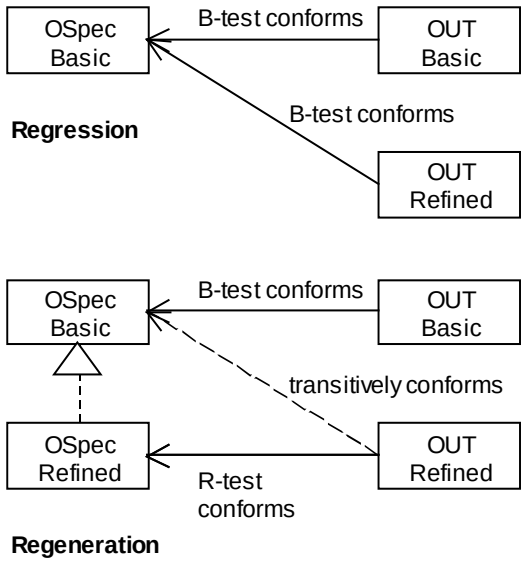

Fig. 7. The new philosophy for testing. The *Refined OUT* does not conform to the *Basic OSpec* because it *B-test conforms* to that specification, but rather because it *R-test conforms* to the *Refined OSpec*, which is a provably correct refinement of the *Basic OSpec*.

5.4 Links with Simulation in Process Calculi

As demonstrated in section 2.2, Cook and Daniels' [13] examples of statechart refinement are all equivalent to the classical refinement of automata, which judges compatibility by trace inclusion [15]. This works so long as the subtype object aliased through the supertype handle is *only* manipulated through the protocol of that supertype. In more realistic execution contexts, objects may be 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, p325]). In this context, an object may be manipulated by more than one protocol, and messages from the different protocols may be interleaved, which may cause deadlocks [22, 23].

We showed in section 2.2 above how an *M4* object could be manipulated through the protocol of *M0*, until it receives <*c*> through another *M4* protocol, at which point the *M0* protocol deadlocks. *M4* is not strongly compatible with *M0*, although it clearly includes the traces of *M0*. We therefore draw an analogy with Milner's π-calculus [14], which contrasts trace inclusion with the stronger *simulation* relationship. From the viewpoint of the *M0* protocol, unseen events that affect the aliased

object through the simultaneous *M4* protocol are "invisible actions", rather like τ-actions in π-calculus. *Weak simulation* is where one process behaves like another up to null assumptions about invisible τ-actions (ie that they do not affect behaviour). The contrasting *strong simulation* is where one process behaves like another in all contexts, irrespective of the τ-actions' unseen behaviour. Our behavioural compatibility is like strong simulation, because the protocol of the supertype is preserved, no matter what invisible actions may be interleaved by the protocols of subtype handles. This is achieved by making sure, in rules 1 and 2, that invisible actions cannot force a refined object into a state that is unrecognised by its supertype's protocol. The rules are therefore normative, since simulation follows from this.

5.5 Acknowledgement

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

References

1. McGregor, J. D. and Korson, T.: Integrating Object-Oriented Testing and Development Processes. Communications of the ACM, Vol. 37, No. 9 (1994) 59-77
2. 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)
3. Beck, K. Gamma E. et al.: The JUnit Project. Website http://www.junit.org/ (2003)
4. 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
5. Wells, D.: Unit Tests: Lessons Learned, in: The Rules and Practices of Extreme Programming. Hypertext article http://www.extremeprogramming.org/rules/unittests2.html (1999)
6. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W.: Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991
7. Object Management Group, UML Resource Page. Website http://www.omg.org/uml/ (2004)
8. Harel, D. and Naamad, A: The STATEMATE Semantics of Statecharts. ACM Trans. Softw. Eng. and Meth., Vol. 5, No 4 (1996), 293-333
9. Bjorkander, M: Real-Time Systems in UML (and SDL), Embedded Systems Engineering, October/November, 2000, http://www.telelogic.com/download/paper/realtimerev2.pdf
10. McGregor, J. D. and Dyer, D. M.: A Note on Inheritance and State Machines. Software Engineering Notes, Vol. 18, No. 4 (1993) 61-69
11. McGregor, J. D.: Constructing Functional Test Cases Using Incrementally-Derived State Machines. Proc. 11th International Conference on Testing Computer Software. USPDI, Washington (1994)
12. Liskov, B., and Wing, J. M.: A New Definition of the Subtype Relation, Proc. ECOOP '93, LNCS 707, Springer Verlag, 1993, 118-141
13. Cook, S. and Daniels, J.: Designing Object-Oriented Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, London (1994)

14. Milner, R.: Communicating and Mobile Systems: the π-Calculus, Cambridge University Press, 1999.
15. Ebert, J. and Engels, G.: Dynamic Models and Behavioural Views. International Symposium on Object-oriented Methods and Systems. Lecture Notes in Computer Science, Vol. 858. Springer Verlag, Berlin Heidelberg New York (1994)
16. Chow, T.: Testing Software Design Modeled by Finite State Machines. IEEE Transactions on Software Engineering, Vol. 4 No. 3 (1978) 178-187
17. Binder, R. V.: Testing Object-Oriented Systems: a Status Report. 3rd edn. Hypertext article http://www.rbsc.com/pages/oostat.html (2001)
18. Holcombe, W. M. L. and Ipate, F.: Correct Systems: Building a Business Process Solution. Applied Computing Series. Springer Verlag, Berlin Heidelberg New York (1998)
19. 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
20. 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
21. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley, New York (2000)
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. 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)
24. Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley (1995)