CS - 01 - 18

Complete Functional Testing using Object Machines

Anthony J H Simons,
Kirill Bogdanov and
Michael Holcombe

Complete Functional Testing using Object Machines

Anthony J H Simons, Kirill Bogdanov and Michael Holcombe, University of Sheffield

Abstract

Most software testing methods, even though they may detect some faults, can never finally assert whether the tested system is correct. This is due partly to the incompleteness of methods that merely aim to "exercise" the code; partly to the lack of a formal relationship between testing and a functional specification; and partly to the difficulty, especially in object-oriented systems, of testing systems built from many components that may together have millions of states. In this paper, we propose the *Object Machine* as a formal model of an object, which captures both the protocol of an object and the semantics of its methods in a way that bridges the gap between abstract mathematical specifications and the provision of tractable, concrete testing criteria. The *Object Machine* is an adaptation of earlier successful work on *Stream X-Machines*, that has been specifically developed to address formal issues unique to object-oriented systems, in particular the way in which one object depends partly upon others for its own behavioural properties, and the resulting indeterminacy of the next state decision function. Software systems conforming to *Object Machine* specifications may be fully functionally tested, using a hierarchical approach that guarantees the integration at each level, subject to a number of relatively non-restrictive requirements.

1. Introduction

Verification and testing are the two conventional means whereby the correctness of a software system is judged, up to its specification. *Verification* involves reasoning about the properties of a system, often using formal description languages, such as Z, VDM, OBJ or other models, such as X-Machines or Petri nets. Verification is performed upon an abstract model of a system, prior to its implementation; and can be accomplished through theorem proving or model checking. A verified system is typically consistent and complete, up to the assumptions made by the model abstractions. *Testing* involves subjecting the implemented system to sample sets of inputs in order to detect faults with respect to its specification. In practice, much industrial testing only exercises as much of the software as is economically feasible; for this reason, testing is incomplete and, where no clear testing strategy exists, haphazard - amounting to no more than "poking around" [1].

1.1 Different Testing Methods

A definition of different testing methods is given in [2]. Relatively informal kinds of testing include *inspections*, based on a peer review of code, and *random* and *statistical testing* (or *operational profile testing*), in which a random selection of inputs is weighted according to the expected pattern of system usage. There are two main kinds of testing that can claim to be more strongly motivated by an understanding of the system under test. *Functional* (or *black-box*) testing builds a test-set from the system's specification and attempts to prove that the abstract behaviour of the implementation is identical to the specification. *Structural* (or *white-box*, *clear-box*) testing bases its strategy directly on the implementation code and attempts to show that all parts of the software have been exercised without failure.

Structural testing is seldom exhaustive, due to the size of the systems under test and the millions of combinations of input values and decision paths. For even moderately large systems, full *branch* and *statement* coverage (full path exploration) is often abandoned in favour of the strictly weaker *decision* coverage (exercising every decision), or *branch condition* coverage (exercising every boolean combination in decisions). Functional testing may be accomplished by the *category partition* method, which identifies equivalence classes of inputs (and boundary values within these categories) and selects representative values from each category, to determine whether the corresponding output matches that expected in the specification. Testing is most effective in revealing defects where partitions are narrowly based on expected failures [3]. Alternatively, *automaton-based testing* may accomplish a similar aim by comparing an implementation with a finite state machine specification.

To summarise, the best current testing practices execute an incomplete, statistically weighted selection of test cases based on relatively unsophisticated input and path coverage strategies. Testing can only reveal defects; the absence of detected defects typically cannot guarantee correctness.

1.2 Combining Verification and Testing

Testing considered alone is therefore too weak a quality assurance mechanism. By contrast, verification is a much stronger mechanism - equivalent to exhaustive testing - up to the assumptions made by the formal models; this puts the burden of proof on the *refinement strategy*, which must correctly translate the model into the implementation [4]. It is clear that verification and testing represent different activities on a continuum of techniques aimed at exercising the system either symbolically or dynamically to establish its compliance with a specification.

The previous work of Holcombe concerns the little-exploited relationship between verification and testing and has led to an important proof of *model refinement* [5]. From this, Holcombe and Ipate developed a *complete* functional testing method [5, 6]. Based on the theory of Stream X-Machines [7, 8] (see section 2 below), this method allows definite statements to be made about the correctness of combinations of tested components in integrated systems. The success of this approach relies on the ability to deal with systems at different levels of abstraction and the notion of model refinement, the fact that specifications at different levels of detail are provably equivalent. Two further results have emerged from this work. The first is the identification of practical design-for-test conditions that allow systems to be specified such that they may be tested effectively. The second is the identification of a point at which it is safe to stop testing.

This contrasts with current approaches in which, once testing is finished, it is impossible to estimate the likely number, location or importance of the faults that may remain in the code. The complete functional testing method allows you to state categorically that an integrated system is correct, provided that: (i) all the tests have been passed; (ii) the components are individually correct; and (iii) the design-for-test conditions have been satisfied. This result is significant in integration testing, which serendipitously has important implications for object-oriented systems, which are *entirely* based on the integration of components.

1.3 A Complete Functional Testing Method

In section 2 of this paper, Chow's complete functional testing method for systems based on finite state automata is reviewed [9]. While this method is provably complete, it is clear that the majority of realistic software systems exhibit greater formal complexity than the restricted class of problems computable with a simple automaton. This was the motivation behind Holcombe's adoption of the *Stream X-Machine* [7, 8], a class of machine with memory, inputs and outputs, which exhibits Turing computability. The useful properties of the Stream X-Machine are explained in overview, in particular how it admits realistic computations, which can be abstracted at one level of refinement and later exposed as the operation of nested, independent machines at a finer grained level.

This compositional property allows systems developed from Stream X-Machine specifications to be tested hierarchically, in a divide-and-conquer fashion [5, 6], avoiding the intractable state explosion common in object-oriented testing approaches [10, 11, 12, 13]. For example, Binder's FREE (Flattened REgular Expressions) method synthesizes subsystem state machines (called *mode* machines) from component object state machines and computes the transition cover for the integrated subsystem [12]; likewise Kim et al.'s approach calculates state products from UML statecharts, testing the flattened system [13]. In contrast, Holcombe and Ipate's hierarchical approach starts with a formally verified abstract architecture which is then

progressively refined. Systems are tested bottom-up, computing a test set based on the transition cover *only* for the state machine of the integration, at each step. This reduced testing obligation is possible because of the formal proof of model refinement [5]; it depends only on the assmption that the components being integrated are correct. This can be assured by testing down to the lowest level component that one is prepared to trust, such as a primitive assignment instruction.

The testing method is *functional*, in that it is based on specifications alone and does not require an analysis of code. It is *complete*, in that it guarantees, for all system states, that all of the required behaviours are present and no undesired behaviour is exhibited. It is *tractable*, in that it greatly reduces the explosion in test set sizes with respect to flattening approaches.

1.4 Meeting Design-for-Test Criteria

The new work reported here concerns the adaptation of the Stream X-Machine (SXM) model to suit specific characteristics of object-oriented systems. Section 3 builds an example SXM specification for a bounded Stack object; and goes on to discuss reservations about the conventions of the SXM model, showing how they do not apply in general to objects.

The properties of systems designed to meet the SXM's design-for-test criteria include that: they complete single-step transitions, which are immediately observable; every transition is distinguishable by a unique output; the next state is computable from the current state, input and global memory; all states are reachable and then every transition may at least be attempted; and transition functions may be tested independently as separate components.

There are a number of differences between monolithic input-driven software systems and systems conceived as a society of communicating objects that impact on these design-for-test criteria. Specific relevant behaviours of object-oriented systems include that: they complete multi-step transitions, with unobserved run-to-completion; computation is driven by messages and responses, not inputs and outputs; memory is distributed and may only be observed through communication; and so the next state depends on remote communication.

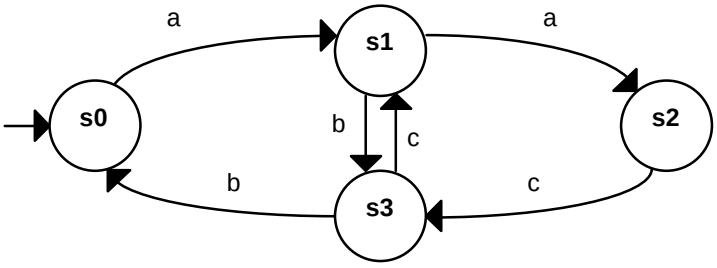
An alternative specification model, known as an *Object Machine*, was devised to behave in a manner that is better aligned with object-oriented implementations. Section 4 describes the architecture of the Object Machine (OM) and explains how it can be used with the adapted design-for-test criteria and an associated testing method to satisfy the same guarantee of correct integration.

2. Automaton-Based Specification and Testing

The background to *complete functional testing* using generalisations of finite state automata is reviewed here. It is important to understand why the approach is *complete*, that is, it provides guarantees of system correctness once testing is complete. Then, it is important to see how the approach is *compositional*, affording a reductionist approach to testing. Finally, it is worth noting that the approach sets a *finite limit* on the amount of testing which is strictly needed to show that a system is in fact correct.

2.1 Chow's Testing Method

Many automaton-based testing methods [12, 14, 15], especially Schumann and Pitt's object-oriented subsystem testing [11] and Holcombe and Ipate's *Stream X-Machine* method [5, 6], are based on Chow's method for *completely* testing systems that conform to finite state machine specifications [9]. The approach puts relatively few constraints on the specification and system under test. The specification must be a *minimal* state machine, that is, there should be no redundancy. While Chow described a *transducer* with inputs and outputs, Ipate showed that even an *accepter* could be tested [16]; the system need only accept inputs taken from a closed alphabet, other than that, it must be determinable whether a transition succeeds or fails in response to an input. This is indicated here using one token from {*ok, fail*}; failure may be inferred from the system crashing or hanging. The states of the system are not directly observable, but may be deduced from the further reactive behaviour of the machine.



S = {s0, s1, s2, s3} Φ = {a, b, c}
W = {<aa>, <bb>, <cc>} C = {<>, <a>, <ab>, <aa>}
K = {<>, <a>, <b>, <c>, <aa>, <ab>, <ac>, <aba>, <abb>, <abc>, <aaa>, <aab>, <aac>}

Figure 1: Simple finite state accepter

The specification is a state machine, like that shown in figure 1, having a finite number of states $s \in S$ and a closed input alphabet $\phi \in \Phi$, corresponding to all the (expected and unexpected) events that may possibly be handled in any state. For each state $s_i \in S$, a subset of inputs $\Phi_{iok} \subseteq \Phi$ labels transitions which exit to states $s \in S$ (including s_i itself); and the set complement Φ

$_{i}\Phi_{fail} = (\Phi - \Phi_{iok})$ are inputs which are illegal in that state. The system under test is supposed to conform *exactly* to this specification, which is to be proven by complete functional testing.

First, it must be possible to determine when the system is in any particular state. A sequence of events $<\phi_1, \phi_2, \phi_3>$ is considered to be taken from the product set $\Phi\times\Phi\times\Phi$, hereafter styled Φ^3, the set of all length-3 sequences. If $\Phi^* = \Phi^0 \cup \Phi^1 \cup \Phi^2 \cup ...$ is the set of all possible arbitrary-length sequences of events chosen from Φ, a *characterisation set* $W \subset \Phi^*$ can be chosen from the specification to identify each state uniquely, based on the reactive behaviour of the system starting in that state. Applying a single sequence $w \in W$ to the system in a state $s_i \in S$ produces an observable result from {*ok, fail*} after $p \leq \text{len}(w)$ transitions are attempted. W is the smallest set of shortest sequences, such that applying every $w \in W$ to s_i produces a unique k-tuple of results {*ok, ok, fail, ...*}, where k = card(W), that is distinct from any other tuple obtained by applying W to every other state in the system. In this way, the system's states can be uniquely identified with states in the specification, by observing the subsequent reactive behaviour of the system. Every time the system fails for some test sequence $w_i \in W$, it must be reinitialised for the next sequence $w_{i+1} \in W$.

Next, it must be possible to show that the system can reach all of its operating states. A *state cover* $C \subset \Phi^*$ is chosen from the specification, being the set of shortest input sequences that will cause the machine to reach all of its states from its initial state $s_0 \in S$. The concatenative product C*W is computed and the system is tested with this set to ensure that all of its states are reachable and are the expected states. The product C*W concatenates every sequence in C with every sequence in W, resulting in a larger set of test sequences whose size is given by: card(C*W) = card(C) × card(W). Testing with this set ensures that the system has *at least* as many states as the specification; the possibility of extra system states is handled below.

Finally, it must be possible to show that, in every state $s_i \in S$, the system responds correctly to all the legal inputs $\phi \in \Phi_{iok}$, and fails for every illegal input $\phi \in \Phi_{ifail}$. Both positive and negative aspects are required to prove the correct operation of the system. A *correct response* includes *both* accepting a legal input *and* completing the associated transition to the next specified state. Intuitively, every input $\phi \in \Phi$ is applied to the state $s_i \in S$, and for each legal input $\phi \in \Phi_{iok}$ the characterisation set W is then applied, to determine if the next state reached is the expected one. Ultimately, this is treated as applying Φ^1*W to every state $s \in S$, that is, every singleton sequence $<\phi> \in \Phi^1$ concatenated with W. Transition failures, unexpected extra transitions from Φ_{ifail} and incorrect reached states are therefore all detected.

In practice, these test sets contain prefix sequences which can be merged before testing. The testing regime need only generate one large test set T and specify, for each sequence $t \in T$, whether it should succeed or fail. To test all of the above, it is sufficient to generate the *transition cover* $K = C \cup C^*\Phi^1$, which includes the state cover (see also figure 1) and

calculate its product with W to verify the reached states. The merged test set is therefore T = K*W, which simplifies to $T = C*W \cup C*\Phi^1*W$. A system that passes this test series has *at least* as many states as the specification and behaves correctly in all these states.

It is sometimes necessary to assume that the system might contain extra, redundant states. These can be detected by applying pairs, triples, ... n-tuples of inputs from Φ^2, Φ^3,...Φ^n in every state $s \in S$ and verifying the reached states with W. Eventually, a duplicate state will reveal itself by leading either to an unrecognised state, or to one with faulty behaviour. The necessary amount of extra testing is finite and computable from assumptions made about the number of redundant states expected in the system. The *complete* test set for a system with n redundant states per desired state is given by: $T_n = C*(\Phi^0 \cup \Phi^1 \cup ... \cup \Phi^{n+1})*W$. Low values of n are usually sufficient, especially if the coding of the system is generated explicitly from the state specification. For n=2 (expecting two redundant states per desired state), the test set is: $T_2 = C*W \cup C*\Phi^1*W \cup C*\Phi^2*W \cup C*\Phi^3*W$. This set is of a tractable size, and can be generated automatically from specifications [17].

2.2 Holcombe and Ipate's Stream X-Machines

Realistic software systems exhibit more complex behaviour than finite state automata, which can only process regular languages. Full Turing computability is afforded by augmented transition networks with memory; and Eilenberg's *X-Machine* [18] is one such model, an automaton whose transitions are *functions* that process an arbitrary data type X. Holcombe first realised that an X-Machine could be used as the formal basis for specifying realistic systems [7], which could then be tested [6] using Chow's method. Specifications are built by identifying the abstract control states of a system, constructing an automaton for this, and including all the remaining detail in the functions acting upon the memory [19, 20]. These arbitrarily complex functions, treated as atomic in the abstract machine, are later decomposed into further X-Machines, corresponding to a one-step refinement of the original specification. The refined machine can be proven behaviourally equivalent to the more abstract machine [5, 16] using Ipate's proof of model refinement.

Of particular interest is the class of *Stream X-Machines* [8], that are driven by inputs and produce outputs, because these permit Chow's method to be applied directly, driving a system through its single-step transitions, and compositionally, using a hierarchical divide-and-conquer strategy. A *Stream X-Machine* (SXM) is a generalised finite state machine which isolates control flow from data processing. It consists of a set of control states $s \in S$ and a set of transition functions $\phi \in \Phi$, which process the data type $X = (I^* \times M \times O^*)$. I* is an input stream of elements $i \in I$, O* is an output stream of elements $o \in O$, and m : M is a global memory. There exists a distinguished initial state $s_0 \in S$ and an initial memory value $m_0 : M$. The transition functions $\phi \in \Phi$ are the labels on the arcs connecting the states $s \in S$. These

functions are triggered by, and read inputs $i_k \in I$ arriving on the input stream, and, when fired, generate corresponding outputs $o_k \in O$ on the output stream. Each function may inspect the current memory value $m_k : M$ and, when fired, may modify memory, giving $m_{k+1} : M$. The functions $\phi \in \Phi$ are therefore considered to have the type $\phi : (I \times M) \rightarrow (M \times O)$. A SXM is similar to a Mealy-style transducer, with inputs and outputs, except that the labels on the arcs are ***functions*** which inspect and modify memory (see also figure 2 below).

A SXM is ***deterministic*** if, for each state $s \in S$, there exists a unique transition function $\phi \in \Phi$ which can fire in response to any input $i \in I$. Occasions may arise when the abstraction over a system's control states is so coarse-grained that multiple $\phi \in \Phi$ are enabled by a given input, resulting in a non-deterministic machine. To enforce determinism, the selection of any ϕ can be made contingent on ***both*** a particular input value ***and*** a guard on the current state of memory. This predicates some of the behaviour of the model on the contents of memory and allows realistic systems, such as VCR equipment, to be modelled convincingly [20]. The guards must be ***mutually exclusive***, to ensure deterministic, and therefore testable, behaviour. Simons has observed [21] how expressing control logic either as guards or states is essentially arbitrary and one is convertible into the other. Transition functions $\phi \in \Phi$ are later modelled as individual ***Stream X-Machines*** in which the memory-dependent guard behaviour is properly exposed as a state machine.

A SXM is ***output distinguishable*** if the firing of each distinct transition function $\phi \in \Phi$ generates a unique output $o \in O$. Outputs are significant in the testing of systems based on SXM specifications. Firstly, there is only an indirect association between an input value and firing a transition (firing may also be contingent upon memory guards). Secondly, it is necessary to determine which transition function ϕ fired for a given input $i \in I$. This is so that each function $\phi \in \Phi$ may be decoupled from the state machine at the current level of abstraction and tested separately. Assuming that the component functions are correct, the test of the integration must prove that the correct components were selected for each transition firing. This is achieved by associating a unique output $o \in O$ with each function $\phi \in \Phi$, to identify both correct and incorrect transition firing. The unique association of outputs to functions is assured in turn through component function testing.

A ***Stream X-Machine*** is ***test-complete*** if, no matter what its current memory values, it can be driven through all its states and transitions when under test. Real systems have rarely-entered and sometimes unreachable states. In a ***Stream X-Machine*** specification, this may be reflected by a set of guards that do not ***exhaustively*** cover the memory value-space. If $\Phi_s \subseteq \Phi$ is the subset of transitions that may legally fire for a given state $s \in S$, then ***normal completeness*** is the no-hang property which ensures, for each state $s \in S$, that the function set Φ_s has guards which exhaustively cover all possible memory conditions, even unexpected combinations. This

can be achieved artificially by extending the set Φ_s with a default function ϕ_{sd}, whose guard is the memory-complement of the guards of the remaining functions in $\Phi_s - \{\phi_{sd}\}$. An easier design-for-test criterion to obtain is *test completeness* under Φ, which is achieved by extending the input domain I with special test values which will drive the machine through all its transitions *whatever* the current state of memory [5, 6].

Another important property of SXMs is their *compositionality*, assured through the independence of each memory-processing function $\phi \in \Phi$ from the current state $s \in S$ in which the machine finds itself. Recall that $\phi : (I \times M) \rightarrow (M \times O)$ implies no dependence on S. Each ϕ may therefore be taken out of context and tested independently. Compositionality is achieved at the cost of a more complex next-state function *F*, which must be capable of selecting the appropriate ϕ to fire, based on $s \in S$, $i \in I$ and $m : M$.

3. Mapping Objects onto Stream X-Machines

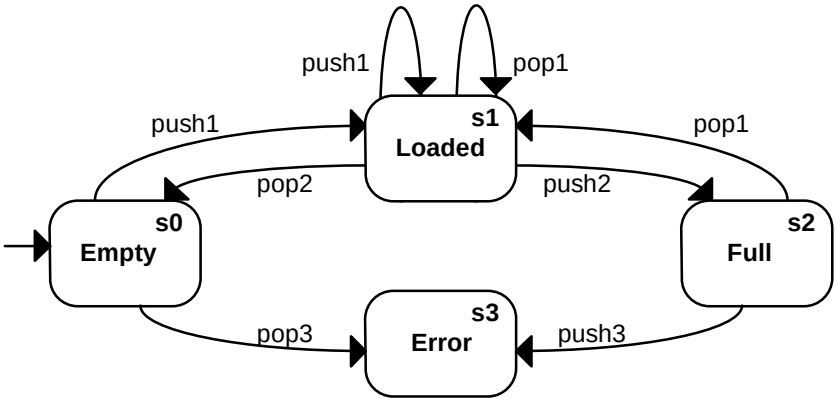
Stream X-Machine specifications are typically developed for a system's *abstract architecture*, and are refined by top-down decomposition of the functions $\phi \in \Phi$ [5, 6, 7]. The top-level state machine is derived by modelling the modes of the user interface, in which user interactions are readily mapped onto inputs and outputs in the model. Here, we seek to apply the technique differently to specify the behaviour of *object-oriented components*, with a view to exploiting the proof of correct integration.

3.1 The Bounded Stack Stream X-Machine

An example SXM specification for a simple bounded Stack object is given in figure 2 below, both as a means of illustrating the elements of a SXM using a familiar data type and also as a starting point for discussing why object-oriented software generally has difficulty in satisfying the model's conventions. The bounded Stack is implemented using a counter and an array. The Stack's bounded size is assumed to be some $n > 1$, to prevent the coalescing of interesting states. The Stack offers the methods *push* and *pop*, and is subject both to overflow and underflow failures.

The Stack machine has four states in the set S = {*Empty, Loaded, Full, Error*} and the initial state is *Empty*. The choice of states is motivated by considering modes in which some of Stack's methods are illegal. Partial functions always provide a good *prima face* case for the selection of control states; each state above corresponds to a mode in which a distinct subset of Stack's methods is illegal. Both the legal and illegal transitions from all these states are modelled in the state machine. For an object with k methods, there are no more than 2^k states.

The set of transition functions Φ = {*push1, push2, push3, pop1, pop2, pop3*} contains multiple transitions for each method, each triggered by a different guard. The splitting of a single function into multiple transitions in the specification is characteristic of this approach; it allows state-dependent algorithms to be factored into different contextual fragments. There must be *at least* as many fragments ϕ_i as there are distinct, mutually exclusive guards on the concrete functions ϕ to avoid nondeterminism; but the same ϕ_i may label more than one transition, provided that the required guard (and intended algorithm) is identical. Here, *pop* transitions have three guarded variants, with [i > 1], [i = 1] and implicitly [i < 1].



push1(push, e, i, a) [i < n-1] == (i+1, a⊕{i→e}, ⊥, pushed)
push2(push, e, i, a) [i = n-1] == (i+1, a⊕{i→e}, ⊥, filled)
push3(push, e, i, a) == (i+1, a, ⊥, overflow)
pop1(pop, ⊥, i, a) [i > 1] == (i-1, a, a(i-1), popped)
pop2(pop, ⊥, i, a) [i = 1] == (i-1, a, a(i-1), emptied)
pop3(pop, ⊥, i, a) == (i-1, a, ⊥, underflow)

Figure 2: Stream X-Machine specification for a bounded stack

Since a SXM is driven by inputs and must produce distinguishable outputs, these are supplied as tuples I = (C × E) and O = (E × T), where C = {*push, pop*} is the set of Stack commands, E is the Stack's element type and T = {*pushed, filled, overflow, popped, emptied, underflow*} is the set of status indicators revealing which $\phi \in \Phi$ was fired. Not every operation requires or produces a valid element e ∈ E, so ⊥ denotes the undefined element. An invocation of the *push* method is modelled as supplying the input pair: (*push*, e) and, if *push1* is triggered, will yield the output pair: (⊥, *pushed*).

The memory type: M = (N × E[n]) is a product type, chosen to meet the specific data storage needs of a Stack. In this, N is the set of natural numbers and E[n] is the array type of length n of the element type E. Memory values are therefore pairs: (i, a) whose first projection i : N is a counter and whose second projection a : E[n] is an array, indexed from 0 to n-1. The initial memory value is (0, {j → 0}) for j = 0..n-1. Executing a transition $\phi \in \Phi$ reads and modifies

the memory; and the specification models this in a functional style, according to the signature $\phi : (I \times M) \rightarrow (M \times O)$, treating the array as a map from indices to values. Expanding out the tuple types for I, O and M gives the full definition of *push1* as:

$$\textit{push1} : (C \times E \times N \times E[n]) \rightarrow (N \times E[n] \times E \times T)$$
$$\textit{push1}\ (\textit{push}, e, i, a)\ [i < n{-}1] == (i{+}1, a \oplus \{i \rightarrow e\}, \perp, \textit{pushed})$$

where the memory guard $[i < n{-}1]$ is used to discriminate between the firing of *push1* and *push2* when the machine is in the *Loaded* state. Further definitions of transition functions are given in figure 2.

There is a certain freedom in the choice of functions when specifying this machine. It can be advantageous, especially for testing purposes, to have fewer functions $\phi \in \Phi$ at this level of abstraction, since a smaller Φ reduces the size of the test set. Accordingly, some of the complexity may be pushed down to a lower level of abstraction. Here, we could merge *push3* with *push2*, since their guards may be combined as $[i \geq n\text{-}1]$. A definition of the merged *push2-3* is given as:

$$\textit{push2-3} : (C \times E \times N \times E[n]) \rightarrow (N \times E[n] \times E \times T)$$
$$\textit{push2-3}\ (\textit{push}, e, i, a)\ [i \geq n{-}1] ==$$
$$\quad \textbf{if}\ (i = n{-}1)\ \textbf{then}\ (i{+}1, a \oplus \{i \rightarrow e\}, \perp, \textit{filled})\ \textbf{else}\ (i, a, \perp, \textit{overflow})$$

This is a more complex function, branching internally on the state of memory before returning one of two possible outputs. It may be exposed in turn at a finer level of granularity as a SXM. Similarly, we could merge *pop3* with *pop2* by combining their guards as $[i \leq 1]$. Even finer grained machines may be constructed for array update and counter arithmetic, down to the lowest level of component that can be trusted.

The Stack machine is *deterministic*. Note that we could not merge the guards of *push1* and *push2* without losing the determinism. The *Loaded* state requires distinct transition functions for the re-entrant and exiting cases, otherwise the behaviour of the machine in response to a *push* command in this state cannot be determined. The same is true for *pop1* and *pop2*.

This machine is *output distinguishable*, since every $\phi \in \Phi$ is associated with a unique status indicator $t \in T$. Although in the specification, the current state $s \in S$ may highly constrain those functions $\Phi_s \subseteq \Phi$ which should legally fire, when driving the corresponding system through its test sequences, it is necessary to expect that any ϕ may fire erroneously and this must be detectable. Sometimes output distinguishability can be assured without recourse to an artificial status indicator set. Naturally, if there are fewer merged functions, fewer status indicators are required: *push2-3* could simply return *filled* for both branches.

To make this machine *test complete*, it must be possible to drive the Stack through all its transitions, starting from the initial *Empty* state. This means that every transition must at least be attempted in every state. Objects immediately satisfy the latter requirement, since any of their methods may be attempted at any time. In practice, we also need to reach the Stack's *Loaded* state within a tractable time interval.

3.2 On the Validity of the Stream X-Machine Model

It is clear that the above Stack specification satisfies the design-for-test requirements of the SXM model, and it is immediately testable using Chow's method. However, there are a number of important mismatches between the modelling conventions of SXMs and the conventions of object-oriented systems.

The first issue is the supposition of the global memory tuple m : M. In theory, any part of m may be accessed by any function $\phi \in \Phi$. In practice, different parts of m are sometimes accessed by different ϕs and it is then possible to partition m into encapsulated chunks that are handled by distinct subsystems [19, 20]. Memory in object-oriented systems is distributed over collections of objects and is encapsulated inside these, as attributes. It is not possible to access *arbitrary* pieces of memory directly, but only the local attributes of an object. This becomes an issue later, especially in the testing of guards. In the revised model, we consider that memory is distributed and also encapsulated, such that parts of memory are not immediately available to the object under test.

The second issue is the way that *Stream X-Machines* are driven in single steps by inputs that are matched with corresponding outputs. It is an important tenet of the testing philosophy that every step of the computation must be observable. Against this, object-oriented systems generally exhibit multi-step behaviour, in which one message invokes a chain of methods that *run to completion* before execution halts. Multi-step transition firing means that observed outputs can no longer be associated uniquely with transitions: an output sequence <a, b> observed for a transition sequence $<\phi_a, \phi_b>$ can be associated three ways with the transitions: wholly with the first, or the second transition, or element-wise with each, the latter being the intended correct association. This means that observing the expected sequence of outputs only has a 1/3 chance of guaranteeing the correct behaviour of the transitions. This scales up badly for longer observation sequences: a length 3 observation sequence may be associated in ten ways with a length 3 transition sequence, for example. In the revised model, we must find a way of assuring single-step progression and a means of observing this.

The issue of observations is also tricky in the object-oriented model, which is not so much driven by data as by functional requests and responses. We do not especially favour the practice of instrumenting code with additional input and output statements that are

conditionally compiled, since the delivered system is then a different artefact from the tested system [22]. Other testing approaches use *oracle objects* that have privileged access to the object under test (eg using the *friend* mechanism in C++, which adds no overhead to the object under test) [12, 14, 15, 22]. Below, we address the same issue differently using the notion of *stub objects* under the control of the tester.

The most difficult problem in applying the described testing method relates to computing the next-state function. In Chow's testing method, the next state must be deterministically decidable from the next input, so that reached states may be verified with respect to expected states. In the SXM model, certain control states, such as *Loaded*, exhibit non-determinism under inputs alone, which can only be resolved by the checking of guards. The first difficulty here is that the required memory values are not always locally available. They may have to be requested from a collaborator object, such as the Stack requesting the value of n from an underlying Array object, in order to evaluate the guard [i < n–1]. Such a communication constitutes a separate step in the machine. Inter-object communications occur during the firing of transitions. This means that transitions are commenced before the eventual result of the communication is known. In general, the behaviour of an object's method may depend in an arbitrary way on messages sent to other objects, such that the destination of the transition may not be known in advance.

A second difficulty, related to this one, concerns the functional nature of tests. In the *Loaded* state, the guards chosen to disambiguate the destination states for *push* and *pop* commands require information related to *boundary pre-conditions*, or the value of the Stack counter i *just prior* to crossing a category boundary (ie from *Loaded* to *Full*; or from *Loaded* to *Empty*, respectively). This kind of information is usually related to *structural* knowledge about implementations, rather than *functional* specifications. A functional specification for a Stack should be expressed in terms of its own category partitions, which are refined in terms of functional specifications of its component objects, the Integer counter and the Array:

empty() $\Leftrightarrow$ i == 0; loaded() $\Leftrightarrow$ 0 < i < a.size();
full() $\Leftrightarrow$ i == a.size(); error() $\Leftrightarrow$ i < 0 $\vee$ i > a.size();

These are *boundary post-conditions*, expressed in terms of values of the Stack counter i *just after* crossing a category boundary. Without privileged access to a Stack's implementation, we may only reason backwards to the *boundary pre-conditions* in this example, because the increment operation on the counter happens to have an inverse, decrement operation. In general, there are kinds of operation for which we cannot expect to be able to compute the inverse, either because the unique inverse does not exist, or because it is computationally intensive to derive - an example is the SHA-1 cryptographic signature algorithm [23]. As a result, we cannot always know the values to be checked in guards, until after the transition has

been completed (or at least the communication part). For this reason also, the destination of a transition may not be known in advance.

4. The Object Machine Model and Testing Method

To handle these concerns, an alternative specification model, known as an *Object Machine*, has been developed, which behaves in a manner that is better aligned with object-oriented implementations. An *Object Machine* (OM) describes the state changes and responses of an object triggered by the reception of message requests. An Object Machine is a concrete, testable specification, derived by transforming a more abstract model of the specification. The initial abstract specification, known as a *Protocol Machine*, describes the object's protocol states, and integrates many *Method Machines*, describing the operations of its methods. The OM testing method is an adaptation of the SXM approach, using stub client- and server-objects under the control of the tester to drive the object under test through single-step computations.

4.1 Protocol and Method Machines

The underlying semantics assumes a universe of primitive *values* and object *references*, from which an object's local memory is constituted. Simple values are immediately available, in the manner of the SXM's global memory. Any computation involving references requires communication via message passing with the object concerned. In the example developed below, it is assumed that a bounded Stack encapsulates a simple-valued counter i, and a reference to a fixed-size Array having n > 1 elements of the type E. Whereas the counter is immediately accessible to Stack methods, the Array size can only be discovered by sending a message to the Array, which is a *communication*.

A *Protocol Machine* is a kind of transducer whose inputs are message requests and whose outputs are the invoked methods. The Protocol Machine for the bounded Stack is shown in figure 3. In addition to the familiar protocol states S_P = {*Empty, Loaded, Full, Error*}, there are two diamond-states representing decision points *after* the main firing of the transition but *before* the following protocol state has been determined. For convenience, we label these decision points S_d = {*Pushed, Popped*}. The total set of states $S = S_P \cup S_d$. Guards placed after the diamonds indicate *postconditions* to be evaluated after the main transition has completed (a departure from the *preconditions* of SXMs). In general, guards may require further inter-object communication; in figure 3, the guard condition *i == a.size()* is checked, which requires a message interaction with the collaborating Array. Guards may therefore be considered *bona fide* transitions in their own right.

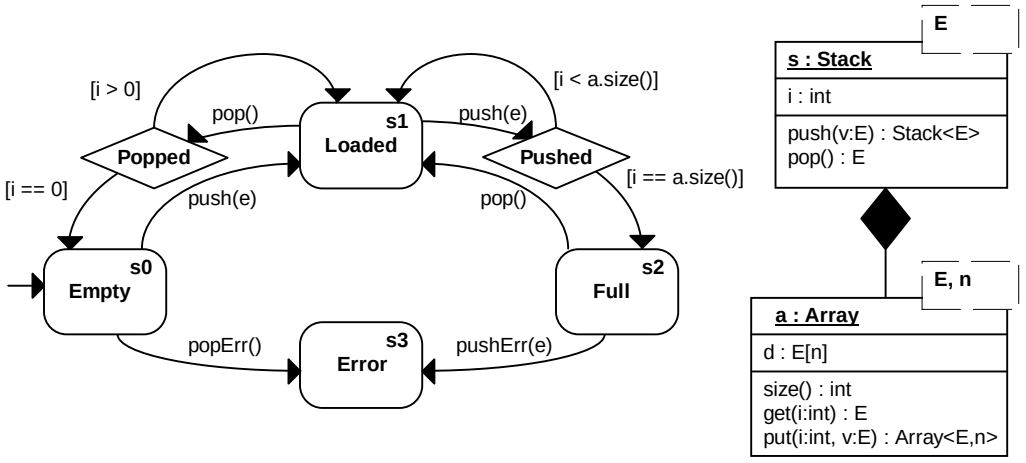


Figure 3: Protocol machine and class diagram for a bounded Stack

One of the motivations behind the Protocol Machine is that it decouples the additional effort required to determine the next protocol state from the fundamental operation of the object's methods. Three insights led to this conceptual view. The first relates to the indeterminacy of the destination state prior to commencing a transition. The issue was forced by engineering a deterministic transition to some state (a diamond-state). Methods now always reach a known state, but this is not always a *protocol* state. It is possible to test this design using Chow's method, by executing guard transitions in the same manner as normal transitions.

The second insight is that the description of the object's *Method Machines* is greatly simplified if guards are abstracted and elevated to the Protocol Machine. There are only two fundamental versions of each method: $\Phi_m = \{push, pushErr, pop, popErr\}$, to which we may add the guards: $\Phi_g = \{gFull, gEmpty\}$. The full set of transitions is $\Phi = \Phi_m \cup \Phi_g$.

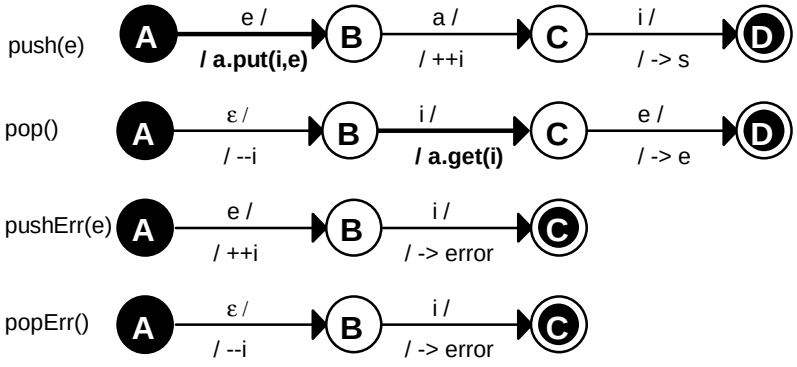


Figure 4: Method machines for a bounded Stack

The *Method Machines* for the bounded Stack are shown in figure 4. A Method Machine is a kind of transducer, whose inputs are values and whose outputs are *communications* (shown highlighted) or *primitive computations* (increments, comparisons, return expressions). The

states of a method machine are the points in between the execution of single instructions. Every transition is triggered by a value or reference returned by the previous computational step, including the initial transition which is triggered by supplying method arguments or ε, the trivial value. Error-reporting is handled by extending every type with the *error* value, which signals an exception. The abstracted guards may themselves be described in exactly the same way as methods in a method machine, since they are *bona fide* transitions. The important boundary postcondition guards for a bounded Stack are shown in figure 5.

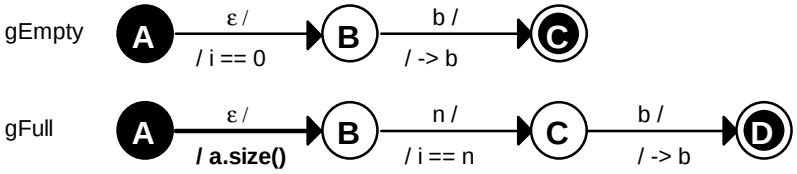


Figure 5: Boundary postcondition guards for a bounded Stack

Each Method Machine is simple, requiring no knowledge of the protocol context in which it is invoked. Method machines may be tested independently of their protocol context, since they only require access to the underlying concrete memory. Naturally, any communication must assume that the collaborator object functions correctly (see section 4.4 below).

4.2 Transformation of the Protocol Machine

The third insight relates to a fundamental mismatch between object-oriented coding styles and automaton-based protocol specification styles. Informally, a method doesn't care which protocol state an object enters when the method terminates. This insight follows from observing the unnaturalness of contrived implementations for *push* that try to compute whether the Stack should enter the *Full* or *Loaded* state after the main operation of the method. On the other hand, an object *does* need to know which state it is in, before it starts executing its *next* method, since partial functions become a salient concern. For instance, *push* should not be legally executable from the *Full* state.

So, whereas a specification will tend to express guards as postconditions on the *previous* transition, implementations tend to delay checking guards until the beginning of the *next* transition, a phenomenon that we refer to as *lazy protocol checking*. This suggests that a decoupling of guard-related computations may be useful, since they are most naturally specified in one way, but checked in another. It should also be possible to transform the Protocol Machine into something closer to the state machine of the implementation, by delaying guard checks until the next transition. This reduces the representational gap between specifications and implementations; and if the rules governing the transformation can be properly codified, the transformation may be performed as part of a model refinement stage.

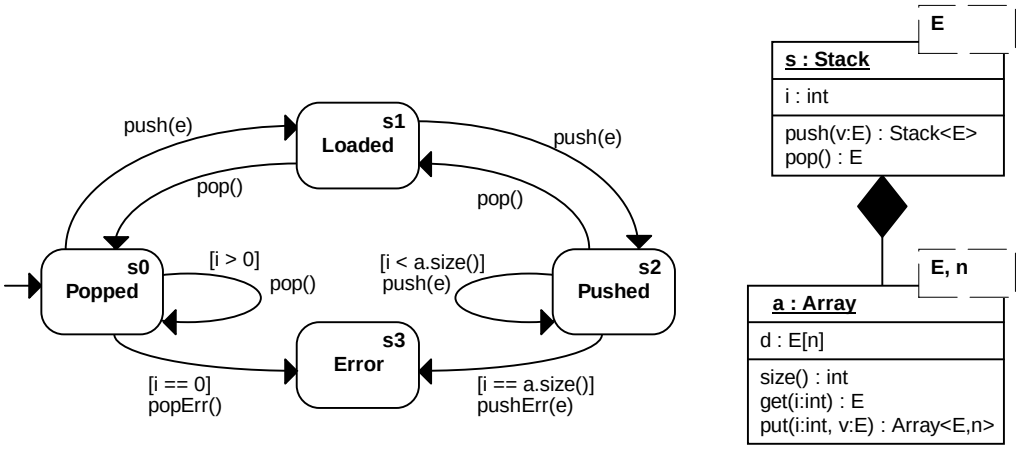


Figure 6: Object machine and class diagram for a bounded Stack

The transformed *Object Machine* corresponding to the Protocol Machine from figure 3 is shown in figure 6. An Object Machine is a state machine whose states correspond exactly to the wait-states of an object prior to receiving a message request. The states are not necessarily protocol states; some are derived by delaying decisions about the next protocol state, such that state-related postconditions on a transition ϕ_i become preconditions on ϕ_{i+1} in transition sequences from Φ^*. The rules for constructing this machine are outlined below.

The reactive behaviour of the Object Machine in figure 6 is interesting. From the initial state, two applications of *push* are possible before a guard must be tested. The third *push* requires a delayed guard check on the counter to determine whether the current *Pushed* state was equivalent to *Full*, and so ensure that the next state is the *Error* state. However, it is possible to apply *pop* in the *Pushed* state without a guard check (*cf* figure 3). From the *Pushed* state, two applications of *pop* are possible before a guard must be tested. Interleaved applications of *push* and *pop* from the *Loaded* state flip between that state and some $s \in \{Pushed, Popped\}$. The state space is also interesting. Compared with the Protocol Machine, the *Pushed* state subsumes *Full* and some of *Loaded*; likewise, the *Popped* state subsumes *Empty* and some of *Loaded*. Whereas the control states of a Protocol Machine were determined uniquely by the content of memory, the control states of an Object Machine are not unique with resepect to memory, but are determined only by the history of method invocations. For example, i == 2 always corresponds to *Loaded* in a n > 2 Protocol Machine, but may correspond to any $s \in \{Popped, Loaded, Pushed\}$ in the equivalent Object Machine; however two applications of *push* from *Popped* always lead to the *Pushed* state, for i = 0, 1, ... n–2.

The rules for constructing the Object Machine from a Protocol Machine specification are based on turning diamond states (decision points) in the Protocol Machine into first-class states in the Object Machine and then delaying postcondition guards by moving them one transition ahead in the graph. For example, the diamond state *Pushed* in figure 3 exits to one of the *Loaded* or

Full protocol states on a postcondition check. In figure 6, *Pushed* becomes a first-class state, subsuming *Full* and some of the old *Loaded* state. The rule for moving guards is based on determining which methods are partial functions in the following protocol states. Recall that protocol states map onto unique sets of partial functions:

$$\{Empty \rightarrow \{pop\}, Loaded \rightarrow \{\}, Full \rightarrow \{push\}, Error \rightarrow \{push, pop\}\}$$

from which we determine that only *push* is disallowed in *Full* and no method is disallowed in *Loaded*. The old postcondition [i == a.size()] related directly to entering the *Full* state, in which the *push* method was illegal. This guard becomes a precondition on the *pushErr* transition exiting the new *Pushed* state. The complementary postcondition [i < a.size()] controlled re-entry to the *Loaded* state, in which all methods were legal. In the transformation, this becomes the complementary precondition on a *push* self-transition to the *Pushed* state, in which the same methods are legal. The *pop* method is unaffected by this particular transformation, because it is not in the set of partial functions under consideration. However, *pop* is affected in the symmetrical consideration of the *Popped* state.

4.3 Transformation of the Method Machines

Similar transformations may be performed upon the methods and guards, whose machines are transferred over from the Protocol Machine and plugged directly into the appropriate place in the Object Machine. Recall that there are two versions of each method: Φ_m = {*push, pushErr, pop, popErr*}, corresponding to pairs of normal and error cases. From most states, the methods may execute unguarded. There is one state, for each pair, in which preconditions must be checked. This can be handled by prepending the guards to the methods, constructed formally by joining the guard and method machines, which can be rendered directly into pseudocode as:

if [i == a.size()] **then** pushErr(e) **elseif** [i < a.size()] **then** push(e) **else end**
if [i == 0] **then** popErr() **elseif** [i > 0] **then** pop() **else end**

However, if the guards can be assumed to cover the entire space of memory values (note that the remaining i > a.size() and i < 0 imply that the Stack is already in the *Error* state) , then we may simplify the above, only checking for the disallowed error cases. This is like saying that methods know themselves when to enter the error state; and looks close to the form of code that programmers typically write for a bounded Stack:

if [i == a.size()] **then** pushErr(e) **else** push(e) **end**
if [i == 0] **then** popErr() **else** pop() **end**

Recall that these specifications relate only to the guarded versions of *push* and *pop*; other occurrences of *push* and *pop* are not guarded, according to the specification. However, programmers normally provide only one implemented version of *push* and *pop*. The specification for these single implemented methods is derived by merging the specifications for all the method variants. The rule for merging unguarded and guarded variants works on the principle that the guards are significant in some cases and redundant in others. The result of a merge will keep the guards, which are redundantly executed in some cases.

A further transformation is possible, based on the idea that error conditions may be detected as late as possible. Assuming that the Array supporting the Stack is bounds-checked, the job of detecting out-of-range insertions and extractions may be delegated to the collaborator. This has the merit that the method machines for *push* and *pushErr* can be constructed to share transition paths; this is even more the case with *pop* and *popErr*. The advantage of merging specifications in this way is that it makes testing simpler: the implemented code will, after all, not contain multiple versions of *push* and *pop*.

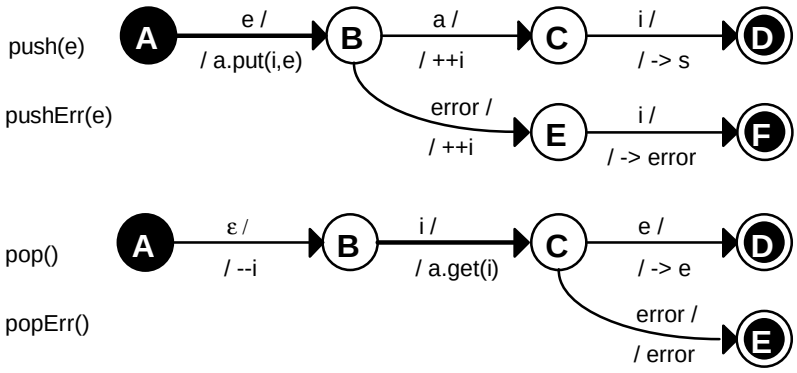


Figure 7: Merged method machines

Figure 7 shows the merged method machines combining the paired correct and error versions of *push* and *pop*. In this, use is made of the returned *error* value to signal the fact that exceptions were raised by the Array collaborator, which are handled by the Stack. The rules for merging transition paths are simple: the nodes and transitions for all variants of a given method are merged for as long as the transitions exhibit the same input/output pairs; but where these diverge, variant transitions lead to distinct states. As a general principle, we find that delaying the reporting of errors results in machines which are more easily merged. The most common case is chosen as the base machine and variant machines are spliced into this.

4.4 Adaptation of the Testing Method

The testing method used is based on the same principles as Chow's method, described in section 2.1. To test a bounded Stack implementation with respect to an Object Machine

specification, it is necessary to show that the states and transitions of both are equivalent. In order to drive the Stack through the test set, a test harness object, StackTester, is used to invoke Stack methods in sequences generated according to the minimal test set algorithm (see section 2.1 and [16, 17]). In order to observe the results of firing single-step transitions, a double policy is adopted in which stub objects are used in place of the Stack's usual collaborators, to observe the effects of *communications*, whereas the StackTester object is used as a privileged oracle (see section 3.2) to observe the results of *primitive computations*. The aim here is to enforce single-step computations, and also provide the equivalent of *distinguishable outputs* (see section 2.2), without interfering directly with the Stack's methods in any way. The stub objects, described below, may also be used to generate parts of the expected test sequence.

Intercepting the Stack's communication with Array using a substituted ArrayStub has many benefits. The ArrayStub object is designed trivially to satisfy the subset of the Array's interface on which the Stack depends, without necessarily having to perform all the associated operations. Instead, the methods of the ArrayStub print simple outputs and supply return values as the tester desires, to drive the Stack through its states. Any Stack method which calls an Array method may be observed and identified by the printed output. Also, since the ArrayStub is under the control of the tester, its methods may wait for control signals from the driving StackTester harness before continuing, so interrupting the normal run-to-completion behaviour of the Stack. The ArrayStub may be designed to ensure that all of the Stack's states are reached in a tractable time interval (see section 3.1), for example, an ArrayStub simulating an Array with n=3 would allow all of the states and transitions in figure 6 to be tested in the shortest time. Finally, the reporting behaviour of the ArrayStub also serves to show that the Stack is communicating correctly with the intended collaborator.

In general, this technique may also be used to handle self-directed messages (a stub substitute for the current object may be used to intercept nested calls), so preventing multi-step run to completion. The substitution of stubs is a simple and minimally-invasive technique that is always available where objects have reference semantics. It requires a design-for-test coding style in which object constructors take arguments which are the intended collaborators. Transitions which perform primitive computations on local memory cannot be observed by intercepting communication. In this case, a different technique is used in which the test harness is granted privileged read-only access to the object's memory state. Privileged access may be granted through *friend* declarations in C++, or using the *basicAt:* instance variable access method in Smalltalk, for example. The equivalent of obtaining a *distinguishable output* is an observation by the test harness, after firing the transition, that the memory of the object has changed in a suitable, unique way.

Following this approach, the Object Machine for the Stack may be completely exercised and the behaviour of the implementation observed. The successful completion of the automatically-generated test series as outlined in section 2.1 guarantees that the integration of methods is correct with respect to the Object Machine specification, on the assumption that the methods are individually correct. A different strategy is required to observe the correct functioning of each method with respect to its Method Machine. Different policies are possible, depending on the tester's confidence that the methods have been correctly implemented [16, 17]. In the examples presented here, the methods are sufficiently simple for category partition testing to be effective [2]. Where methods exhibit sufficiently complex state-dependent behaviour, they may be tested by a further application of Chow's approach. The policy that most directly mimics this is to execute each method with breakpoints for observation after each instruction; in general, this is considered too invasive and too costly, equivalent to observing every instruction in a stepping debugger. However, automated verification of stack-frame values is possible in a language like Java, in which single instructions may be intercepted on the stack of the Java Virtual Machine. A compromise solution is to rely on the communication-steps in methods to provide the breakpoints at which observations are made of the cumulative effects of local memory modifications. Since the stub collaborator objects are under the control of the tester, they may be constructed to refer back to the driving test harness, through which privileged observations of local memory are made. This is an approach that we are currently exploring.

5. Conclusions

A complete functional testing method for object-oriented components has been presented, in which minimal test sequences are generated from Object Machine specifications and applied using Chow's testing approach. The outcome of a successful test is a proof of correct integration of the object's methods. The methods may be unit-tested independently before their integration, using a variety of approaches. The outcome of a successful test is a proof that the method communicates with the intended collaborator objects, whose correct behaviour is assumed, and leaves the current object in the expected local memory state. The approach is applied recursively to collaborating objects, in a divide-and-conquer fashion, down to the smallest trusted component. The integration guarantee provides a true measure of test *effectiveness*, rather than coverage metrics which merely measure test *effort*: it supports a truly modular testing strategy in which systems may incorporate trusted components, in which all remaining possible faults are necessarily to be found. This testing method therefore makes *definite* statements about the quality of tested code.

A novel aspect of this work is the derivation of an *Object Machine*, a concrete specification that closely mimics the coding styles used by programmers, by automatic transformation from

a more abstract *Protocol Machine*, the natural specification style used by designers. This transformation was intended to help in satisfying the design-for-test conditions discovered for the earlier Stream X-Machine testing approach, but serendipitously has great potential for future automatic code generation approaches based on *model refinement*. The prospect of generating code directly from checked specifications, using formally-verified transformation rules, may in future provide another route to completely trustworthy code.

Another novel theoretical finding is how the notion of object state can be defined in up to three different ways: *protocol states* are motivated from the modes in which distinct subsets of an object's methods are partial functions, *memory states* are abstractions over the object's concrete data attributes and *wait states* are the reactive states in which an object expects an external stimulus. It is a plausible conjecture that *protocol states* may always be determined from *memory states*; but an object's *wait states* are provably different (section 4.2). We are currently investigating the idea that "external stimuli" may include both message requests and also method responses. A wait state is then one in which an object expects a new request, or a response from a collaborator. A communicating state machine designed around such wait states would subsume an Object Machine and the communication steps of its associated Method Machines, but hide internal primitive computations. Call-back invocation and other forms of mutually-recursive invocation could be handled transparently in such a machine; this is the subject of future work.

Acknowldegement: This research was sponsored by EPSRC GR/M56777.

References

[1] R V Binder, Object-oriented testing: myth and reality, *Object Magazine, 5(2)*, 73-75, 1995.

[2] British Computer Society Special Interest Group in Software Testing, *Standard for Software Component Testing, working draft 2.0*, October, BCS SIGIST, 1994.

[3] D Hamlet and R Taylor, Partition testing does not inspire confidence, *IEE Trans. Soft. Eng., 16(12)*, 1402-1411, 1990.

[4] C Morgan, *Programming from Specifications, 2nd ed.*, Prentice Hall, 1994.

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

[6] W M L Holcombe, An integrated methodology for the formal specification, verification and testing of systems, *Software Testing, Verification and Reliability, 3(3/4),* 149-163, 1993.

[7] W M L Holcombe, X-machines as a basis for dynamic system specification, *Software Engineering J., March,* 69-76, 1988.

[8] W M L Holcombe and F Ipate, Another look at computability, *Informatica, 20,* 359-372, 1996.

[9] T Chow, Testing software design modeled by finite state machines, *IEEE Trans. Soft. Eng., SE-4 (3),* 178-187, 1978.

[10] R V Binder, Testing object-oriented systems: a status report, *American Programmer, 7(4),* 22-28, 1994.

[11] S A Schuman and D H Pitt, Object-oriented subsystem specification, in: *Program Specification and Transformation,* Elsevier Science, 1987.

[12] R V Binder, The FREE approach to object-oriented testing: an overview, (a synthesis of four previously published articles), *http://www.rbsc.com/pages/FREE.htm,* 1996.

[13] Y G Kim, H S Hong, D H Bae and S D Cha, Test cases generation from UML state diagrams, *IEE Proc. Softw., 146(4),* 187-192, 1999.

[14] J D McGregor and D M Dyer, A note on inheritance and state machines, *Software Engineering Notes, 18(4),* 61-69, 1993.

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

[16] F Ipate, *Theory of X-Machines with Applications in Specification and Testing,* PhD Thesis, Department of Computer Science, University of Sheffield, 1995.

[17] K E Bogdanov, W M L Holcombe and H Singh, Automated test set generation for Statecharts, in: *Applied Formal Methods (FM-Trends '98),* eds. D Hutter, W Stephan, P Traverso and M Ullmann, *Lect. Notes in Comp. Sci., 1641,* Springer Verlag, 107-121, 1999.

[18] S Eilenberg, *Automata, Languages and Machines, vol. A,* Academic Press, 1974.

[19] M Holcombe and F Ipate, *Correct Systems: Building a Business Process Solution*, ISBN 3-54-076246-9, Springer Verlag, Berlin 1998.

[20] M Holcombe, M Fairtlough, F Ipate, C Jordan, G Laycock and Z Duan, Using an X-machine to model a video cassette recorder, *Current Issues in Electronic Modelling, 3*, 141-151, 1995.

[21] A J H Simons, On the compositional properties of UML statechart diagrams, *Proc. 3rd Conf. Rigorous Object-Oriented Methods*, Bradford, 2000.

[22] J D McGregor, *Testing Object-Oriented Software*, Tutorial, European Conf. Obj.-Oriented Prog., Jyväskylä, 1997.

[23] National Institute of Standards and Technology, *The Secure Hash Algorithm (SHA-1)*, NIST FIPS PUB 180-1, Secure Hash Standard, U.S. Dept. Commerce, April 1995