

## State-based Testing is Functional Testing !

Florentin Ipate, Raluca Lefticaru  
Department of Computer Science and Mathematics  
University of Pitesti  
Str. Targu din Vale 1, 110040 Pitesti, Romania  
Email: florentin.ipate@ifsoft.ro, raluca.lefticaru@gmail.com

### Abstract

*Empirical studies report unsatisfactory fault detection of state-based methods in class testing and advocate the use of functional methods to complement state-based testing. In this paper, we take the view that the modest fault detection of state-based class testing reported in the literature is actually due to the inappropriate state diagram used. We show that functional testing of a class can be reduced to state-based testing, provided that the right state model is produced. We present a strategy for constructing a state diagram for a class method, based on a domain partition derived through functional techniques. We also describe a method for deriving test sequences from the resulting state diagrams, essentially a variant of the W-method. The paper also reports results from an experimental evaluation of the proposed approach, based on mutants generated by MuJava.*

**Keywords:** State-based testing; Test generation; Functional Testing; Finite state machines; Stream X-machines; Mutation testing

### 1 Introduction

State diagrams (statecharts) have been used by many modelling languages and development methodologies [8, 28], most importantly UML [9], for modelling the state behaviour of objects. State diagrams are formally described by *finite state machines* (FSMs). As powerful test selection methods from FSMs exist [29, 19], it is natural to use these state-based models in class testing and a number of coverage criteria for classes and clusters of classes exist in the literature [5, 6, 24, 25]. However, empirical studies show that, while state-based methods are more effective than random testing [25], they are not likely to be sufficient unless the state diagram is straightforward and fully captures the behaviour of the class [2, 10].

Briand et al. [2, 10] report results from a series of experiments to determine the cost effectiveness of one of the most widespread state-based test strategies, the round trip [5], a variant of the well-known *W*-method [12]. In the experiments, two types of oracle are used to determine the outcome of test cases:

- A *precise oracle*, when the *concrete* state is checked. This can be achieved by dumping the values of all class attributes and comparing them to what is expected.
- An *abstract oracle*, as proposed by Meyer [22] which entails checking the state invariants of the states that are expected to be reached during the execution of test cases. In other words, this type of oracle checks the *abstract* state of the model.

The studies conclude that state-based testing is not likely to be sufficient when used in conjunction with an abstract oracle. Fault detection can be significantly improved by a precise oracle, but, on the other hand, precise oracles may often be too expensive. In such cases, the studies advocate the use of functional (black-box) methods, such as category-partitioning [26] to complement state-based testing. Category-partition is used to test class methods *in isolation* and the results show that the fault detection capability of the combined round trip / category-partition strategy is significantly better than that of the round trip. The authors of the studies remark that state-based testing needs to be complemented with functional testing because “the class behaviour is typically not fully captured by statecharts” [10].

Functional methods based on domain partitioning are fairly straightforward to apply when the functionality of the implementation under test is solely determined by inputs. Class methods, on the other hand, are normally dependent on the values of class attributes (the internal state of the object), so functional testing of a method cannot really be performed in isolation; in order to test a method, it is necessary to reach first one or more states, so testing of a method

may depend on other methods. Consequently, a systematic functional strategy for class methods will have to be guided by a state-based model.

In this paper, we take the view that the modest fault detection capability of state-based class testing reported in the literature is actually due to the inappropriate state diagram used. We show that functional testing of a class can be reduced to state-based testing, provided that the right state model is produced. We present a strategy for constructing a state diagram for a class method, based on a domain partition derived through functional techniques. This strategy has come out of a series of experiments we have performed to evaluate the effectiveness of state-based testing. Unlike previous work [4, 11, 13, 14, 21, 23], our approach is to construct a separate state diagram for each method to be tested, thus avoiding the combinatorial state explosion associated with the construction of a composite state diagram for the class and reducing test size. Furthermore, the transitions of the resulting state diagram are closely related to the subdomains of the original partition, and so functional testing is basically incorporated into state-based testing. The paper also presents a method for deriving test sequences from the resulting state diagrams, essentially a variant of the *W*-method. Since any statechart with non-trivial data is actually an *extended* FSM, rather than just a simple FSM, the paper also discusses how the proposed test generation technique can be applied to an extended FSM. The adaptation of FSM based test selection methods to extended FSMs is often neglected in the literature when class testing is discussed. Finally, the papers reports results from an experimental evaluation of the proposed approach, based on mutants generated by MuJava [20, 30].

The remainder of the paper is structured as follows. Section 2 briefly outlines the principles of functional testing, while section 3 introduces FSMs and the *W*-method. The components of the Stream X-machine model are described in section 4, the procedure for constructing the state diagram of a class method is given in section 5 and the associated test generation technique is presented in section 6. Finally, the proposed approach is compared with previous work in section 7, experimental results are reported in section 8 and conclusions are drawn in section 9.

## 2 Functional Testing

Functional testing methods treat the system as a “black box”, and so construct the test sets solely from the information given in whatever specification is supposed to define the behaviour of the system. There are two basic concepts that are fundamental to most functional testing techniques. One is *equivalence partitioning*, the other is *boundary analysis*. The most commonly used functional method, category-partition [26], is based on these two concepts.

The basic idea of equivalence partitioning is to divide the input domain into *equivalence classes* which, according to the specification, are treated identically. Furthermore, the output domain is also considered and “reversed engineered” to be expressed in terms of the input domain [27]. Since all data from an equivalence class should be processed in a similar fashion, a representative from each class can be arbitrarily chosen.

Consider, for example, the *push* method of a bounded stack with maximum  $k$  elements of type *Item*,  $k \geq 2$ . The inputs of the method are the stack itself<sup>1</sup> and the item to be placed on top of it. The output is Boolean, indicating whether the method has been successful or not. There are no obvious choices for equivalence classes when only the inputs are considered. On the other hand, the output clearly divides the input domain into two equivalence classes:  $C_1 = Items[k - 1] \times Item$  and  $C_2 = Items^k \times Item$ , where  $X[k] = X^k \cup \dots \cup \{\epsilon\}$  represent the set of all sequences of length at most  $k$  with elements in  $X$ .

Equivalence partitioning is often used in conjunction with another technique, called boundary analysis, that focuses on a likely source of faults: the boundaries between classes. Once the equivalence partition has been established, the technique derives test data by examining the edges of the partition [27]. Consequently, boundary analysis can be regarded as a strategy for refining the equivalence classes derived through equivalence partitioning.

For the above example, the boundary between the two classes will be composed of stacks with  $k - 1$  and  $k$  elements, so the enriched partition will contain three equivalence classes:  $C'_1 = Items[k - 2] \times Item$ ,  $C'_2 = Items^{k-1} \times Item$  and  $C'_3 = Items^k \times Item$ .

Functional testing is generally straightforward to apply to systems without internal state or when the internal state does not influence the functionality under test. When the state plays a functional role, the behaviour of the system is often described by a state diagram, which can itself be used as basis for test generation.

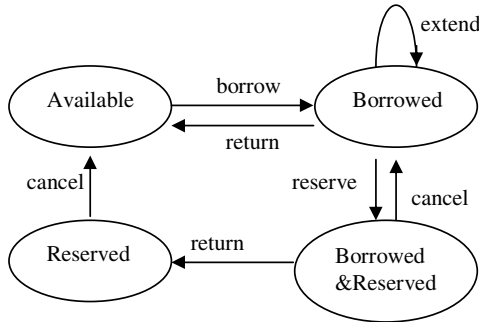
## 3 Finite state machine based testing

*Finite state machines (FSMs)*, or *automata*, are simple mathematical means of describing state diagrams. A FSM  $A$  consists of a finite set of states  $Q$ , of which one is the designated initial state  $q_1$ , and transitions between states labelled by symbols from a finite alphabet  $X$  [19].<sup>2</sup> The FSM model of a simplified *Book* class from a library system is given in Fig. 1 and will be used for illustrations. The model

<sup>1</sup>At this stage we do not differentiate between input parameters and internal state, this distinction will be made latter in our discussion.

<sup>2</sup>In general, transitions may also be labelled by output symbols, but the simpler model presented here is sufficient for our purposes.

is sufficiently self explanatory and so, due to space constraints, we will not provide further details.



**Figure 1. FSM model of a Book class**

Once we have a FSM representation of a system, we can use appropriate techniques to uncover possible errors in the implementation, such as erroneous transition labels, erroneous next-states, missing states, extra states, etc. One of the most general approaches is the  $W$ -method [12], that generates sequences of symbols to reach every state in the diagram, check all the transitions from that state and identify all destination states to ensure that their counterparts in the implementation are correct. Thus the  $W$ -method involves the construction of two sets of sequences:

- A *transition cover*  $T \subseteq X^*$  that reaches every state of the FSM and exercises every transition from that state.  $T$  can be written as  $T = S \cup SX$  for some set of sequences  $S \subseteq X^*$ , called a *state cover*, that reaches every state of the FSM.
- A *characterization set*  $W \subseteq X^*$  that distinguishes between every pair of states in the FSM. In other words, for every pair of distinct states  $q_1$  and  $q_2$ ,  $W$  contains at least one sequence of labels that can be applied from  $q_1$  but not from  $q_2$  or vice-versa.<sup>3</sup>

In Fig. 1, the empty sequence  $\epsilon$  reaches the initial state *Available*, *borrow* reaches *Borrowed*, *borrow reserve* reaches *Borrowed&Reserved*, while *borrow reserve return* reaches *Reserved*. Thus  $S = \{\epsilon, borrow, borrow reserve, borrow reserve return\}$  is a state cover and  $T = S \cup SX$  is a transition cover. On the other hand, it can be observed that *return* partitions the state space into  $\{Available, Reserved\}$  and  $\{Borrowed, Borrowed&Reserved\}$ , while *cancel* distinguishes between the elements of each of the two classes of the partition. Thus,  $W = \{return, cancel\}$  is a characterization set. In general, a transition cover is derived from a

<sup>3</sup>A FSM in which all states are reachable and pairwise distinguishable is called *minimal*. This is not a restriction, however, since, for every FSM, a minimal, functionally equivalent, FSM can be constructed [19].

transition tree constructed in a breadth-first fashion, while a characterization set is constructed by gradually partitioning the state set based on the responses produced by sequences of length  $k \geq 1$ , using the so-called  $P_k$  tables [12].

Suppose we have a FSM specification  $A$  and we have constructed a transition cover  $T$  and a characterization set  $W$  of  $A$ . Naturally, we assume that the implementation of  $A$  can be modelled by an unknown FSM  $A'$ . The only information we need about  $A'$  is an estimation of the maximum possible number of states  $n'$  that it may have. Suppose we denote by  $k$ ,  $k \geq 0$ , the difference between this estimated upper bound and the number of states  $n$  of the specification  $A$ . Then the test suite produced is<sup>4</sup>

$$Y = TX[k]W.$$

The idea is that  $T$  ensures that all the states and all the transitions in  $A$  are also present in  $A'$  and  $X[k]W$  verifies that  $A'$  is in the same state as  $A$  after performing each transition. Note that the latter set contains  $W$  and also all sets  $X^iW$ ,  $1 \leq i \leq k$ . This ensures that  $A'$  does not contain extra states, or even if there are some extra states, they behave the same way as the corresponding specification states. If there were up to  $k$  extra states, then each of them would be reached by some input sequence of length up to  $k$  from the existing states.

Note that the  $W$ -method relies upon the existence of a reliable reset, that correctly puts the system specified by the FSM into its initial state before each test sequence is executed. In the case of a state diagram of a class, discussed in this paper, the class constructor can safely be considered to be a reliable reset.

Variants of the  $W$ -method also exist in the literature. The round trip [5] approach is based on a transition tree constructed in a depth-first fashion. Furthermore, an abstract oracle [22] can be used instead of  $W$  as in [10] to check the state of the implementation under test.

## 4 Stream X-machines

A FSM represents a system in terms of states and transitions between states. However, a FSM cannot fully capture the semantics of a statechart of a class. For this, an *extended FSM*, that adds suitable data structures and operations to a FSM, is needed. The *stream X-machine (SXM)* [7, 15, 16, 18] is precisely such an extended FSM.

<sup>4</sup>This formula is given in [12] for the case in which the FSM specification is completely-specified and transitions are labelled by outputs as well as inputs. As explained in [18], when the specification is partially-specified (the case considered in this paper), the test suite will also include some of the prefixes of the sequences in  $Y$ . However, in order to check the result produced by an input sequence, one would normally check the results produced by all its prefixes; consequently, this extra sub-set of prefixes does not explicitly appear in the formula.

A SXM  $Z$  can be described by a finite automaton  $A_Z$ , called the *associated automaton of  $Z$* , in which the transitions between states are labelled by *processing functions* from a finite set  $\Phi$ . The machine has some data store  $M$ , called *memory*, and each processing function will read an input from an input alphabet  $\Sigma$  and, on the basis of the current value of the memory, will produce an output from an output alphabet  $\Gamma$ , while possibly changing the value of the memory. Thus, all processing functions are (partial) functions of type  $M \times \Sigma \rightarrow \Gamma \times M$ .

Now, let us examine how these components can be defined for a SXM that describes the behaviour of a class.

- **Input:**  $\Sigma$  is formed from the *method name* and the domain of *input parameters* (if any) for each method, i.e.  $\Sigma = \bigcup \{f\} \times Input_f$ , with the union extending over all class methods  $f$ .
- **Output:**  $\Gamma$  is formed from the domain of *output parameters* (if any) and of any other *observable outputs* for each method, i.e.  $\Gamma = \bigcup Output_f$ .
- **Memory:**  $M$  is formed from *tuples*, each component of the tuple corresponding to the domain of an *attribute*, i.e.  $M = At_1 \times \dots \times At_n$ .

Consider again a *Book* class with four methods, *borrow*, *return*, *reserve* and *cancel*. Naturally, each of these methods will receive a customer ID as input parameter. Suppose this ID is represented as a positive integer. Then  $\Sigma = \{bor, ret, res, can\} \times (\mathbb{N} \setminus \{0\})$ . There are no output parameters, so  $\Gamma$  will only contain the messages displayed by each method (i.e. confirming if the operation has been successfully completed or not, etc.). The class will have two attributes, the ID of the customer that has borrowed the book and the ID of the customer that has reserved the book. Suppose that the value 0 of the attribute is used to indicate that the book is not currently borrowed / reserved. Then  $M = \mathbb{N} \times \mathbb{N}$ .

Each transition label  $\phi \in \Phi$  will be identified by the name of the corresponding class method and the subdomain of  $M \times \Sigma$  for which the transition fires. In the simplest case, the subdomain can be deduced from the state diagram, e.g. in the *Book* diagram, *borrow* can be applied whenever the book is available, so the subdomain is  $(\{0\} \times \{0\}) \times (\{bor\} \times Input_{bor})$ . In general, however, the current state and the name of method may not be sufficient to determine the next state of the transition. In such cases, the subdomain can be specified (as in UML statecharts, for example) as a guard accompanying the method name, i.e.  $\phi = f[g]$ , with  $f$  the method name and  $g$  the transition guard.

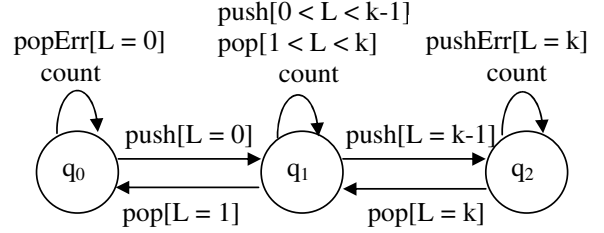


Figure 2. State diagram of a stack

## 5 Constructing the state diagrams

While the identification of the input, output and memory sets is quite straightforward and does not require any major decision-making, the construction of the state diagram will, naturally, depend on the approach used. Very often, different methods may result in quite different diagrams for the same system. Furthermore, the resulting diagram may heavily influence (one way or the other) the effectiveness of the tests derived from it.

Consider, for example, a bounded *Stack* with maximum  $k$  elements of type *Item*,  $k > 3$ , and three methods: *push*, *pop*, and *count* (which, given an integer  $l$ ,  $1 \leq l \leq k$ , finds the number of distinct elements among the last  $l$  elements of the stack); for example, if the stack is *abc* and  $l = 3$  then there are 2 distinct elements among the last 3, so *count* will return 2. The conventional wisdom is to associate a stack with a three-state diagram (one state for when the stack is empty, one for when the stack is full, and one for when the stack is neither empty nor full), as in Fig. 2.<sup>5</sup> Now, consider the application of the *W*-method to this diagram. Suppose that the test generation algorithm will always push the same element on to the stack. This is a perfectly valid algorithm, since there is no restriction in the state diagram to rule out this type of test selection strategy; in fact, it is quite common to test a stack by using only one distinct element. Then, no matter in which state we check the *count* transitions, we will only be able to test the cases in which *count* returns 0 or 1.

The conclusion that can be drawn from the above example is that the state diagram must be constructed in such a way that the test sequences derived from it correspond to the functional test cases of individual class methods. The problem, however, is that different methods may require different test cases. Consequently, a diagram that combines all individual test cases may suffer from state explosion.

Our approach is to construct not a composite diagram for the class, but a diagram for each of the class methods to be tested. Furthermore, the construction of the diagram

<sup>5</sup>In Fig. 2,  $L$  denotes the length of the stack (i.e. the position of the top-most element).

will be based on a partitioning of the domain of the class method in question, thus providing a vehicle for combining state-based and functional testing.

Consider a method  $f$  and a partition of  $M \times Input_f$  derived using some functional technique, specified as a set of mutually disjoint predicates  $p_1, \dots, p_k$  on  $M \times Input_f$ .<sup>6</sup> Then the construction of the state diagram for  $f$  involves the following steps.

1. Derive (not necessarily disjoint) subdomains  $M_1, \dots, M_k$  of  $M$  by “hiding” the input parameters in the partition of  $M \times Input_f$ , i.e.  $M_i = \{m \in M \mid \exists input \in Input_f \cdot p_i(m, input)\}$ ,  $1 \leq i \leq k$ .
2. Derive the states of the diagram  $q_1, \dots, q_n$  by removing the overlapping of  $M_1, \dots, M_k$ . That is, each state will be associated with some non-empty set  $M'_1 \cap \dots \cap M'_k$ , where  $M'_i$  is either  $M_i$  or  $M \setminus M_i$ ,  $1 \leq i \leq k$ . The initial state  $q_1$  is chosen such that it contains the initial values of the attributes.
3. Construct the transitions of the diagram by considering each pair of states and checking whether the pair is related to a method. That is, for states  $q_i$  and  $q_j$  and method  $h$ , the transition guard  $g$  will be a predicate on  $M \times Input_h$  defined by  $g(m, input)$ ,  $m \in q_i$  and  $input \in Input_h$ , if and only if there exists  $m' \in q_j$  such that  $h$  transforms  $m$  into  $m'$  when given input parameter value  $input$ . If the guard  $g$  induces a non-empty subdomain of  $q_i \times Input_h$  then the diagram will contain a transition from  $q_i$  to  $q_j$  labelled  $h[g]$ . At this step, the methods, other than  $f$ , that do not change the value of the memory may be ignored since they do not influence the testing strategy described in section 6.
4. Refine the transitions involving  $f$  using the predicates  $p_1, \dots, p_k$ . That is, each transition labelled  $f[g]$  will be split into at most  $k$  separate transitions, labelled  $f[g \wedge p_i]$ ,  $1 \leq i \leq k$ ; naturally, only the guards  $g \wedge p_i$  that determine non-empty subdomains of  $M \times Input_f$  will be retained.

Consider again a bounded *Stack* with maximum  $k$  elements of type *Item*,  $k > 3$ , and three methods: *push*, *pop*, and *count*, as above. The stack has one attribute, the stack contents<sup>7</sup>; *push* and *count* have one input parameter each, while *pop* has no input parameters. Thus  $M = Item[k]$ ,  $Input_{push} = Item$ ,  $Input_{count} = \mathbb{N}$  and  $Input_{pop} = \emptyset$ . We illustrate the construction of the state diagram for the

<sup>6</sup>Predicates that only constrain inputs are not considered here; since test cases are not influenced by state, a diagram is not necessary in this case.

<sup>7</sup>When implementing the class, one would normally use three attributes: the stack contents  $s$ , the position  $L$  of the top-most element and the upper bound  $k$ ; since  $L$  can be deduced from  $s$  and  $k$  is fixed, for simplicity, these two are ignored in our discussion.

*count* method. Suppose the equivalence classes for *count* are expressed by the following predicates:<sup>8</sup>

- $p_1(m, l)$  if and only if  $count(m, l) = 0$ .
- $p_2(m, l)$  if and only if  $count(m, l) = 1$ .
- $p_3(m, l)$  if and only if  $count(m, l) = i$ ,  $2 \leq i \leq k - 1$ .
- $p_4(m, l)$  if and only if  $count(m, l) = k$ .

Then  $M_1 = \{\epsilon\}$ ,  $M_2 = M \setminus \{\epsilon\}$ ,  $M_3$  represents the set of stacks of length  $L \leq k$  for which the number  $N$  of distinct elements is at least 2, while  $M_4$  represents the set of stacks for which  $N = L = k$ . Thus the resulting state diagram will have 4 states:

- $q_1$ : the empty stack.
- $q_2$ : stacks for which  $N = 1$  and  $L \leq k$ .
- $q_3$ : stacks for which  $2 \leq N \leq k - 1$  and  $L \leq k$ .
- $q_4$ : stacks for which  $N = L = k$ .

The state diagram derived at step 3 is represented in Fig. 3(a); the notation  $[i \rightarrow j]$  is used to indicate that the number of distinct elements in the stack is changed from  $i$  to  $j$ . The final state diagram, represented in Fig. 3(b) is obtained by splitting the *count* transitions into distinct transitions for the subdomains identified by  $p_1, p_2, p_3$  and  $p_4$ .

Similarly, if the partitions for *push* and *pop* are  $\{Item[k - 2] \times Item, Item^{k-1} \times Item, Item^k \times Item\}$  and  $\{\{\epsilon\}, Item, Item[k] \setminus Item[1]\}$  then the state diagrams are as represented in Figs. 4 and 5, respectively. Since *count* does not change the memory value, it is omitted from these diagrams. The state-diagrams represented in Figs. 4 and 5, also contain the error handling parts of the *push* and *pop* methods, respectively, represented as self-looping transitions. Alternatively, the erroneous use of methods can be represented as transitions leading to an (extra) error state. In general, the error-handling part of a method will appear only on the state diagram for that method.<sup>9</sup>

## 6 Test generation

Suppose we have built the SXM for a class method  $f$ . Then we can generate test sequences from this model using the adaptation of the  $W$ -method described in what follows.

<sup>8</sup>Equivalence classes that correspond to invalid values of  $l$ , e.g.  $l = 0$  and  $l > k$ , are considered separately, since they do not involve the state of the object.

<sup>9</sup>Testing a method that throws exceptions, like *pop* or *push*, is straightforward when the erroneous part appears on the state-diagram. It involves surrounding each method call by a try-catch block and checking the effect: expected exception or result.

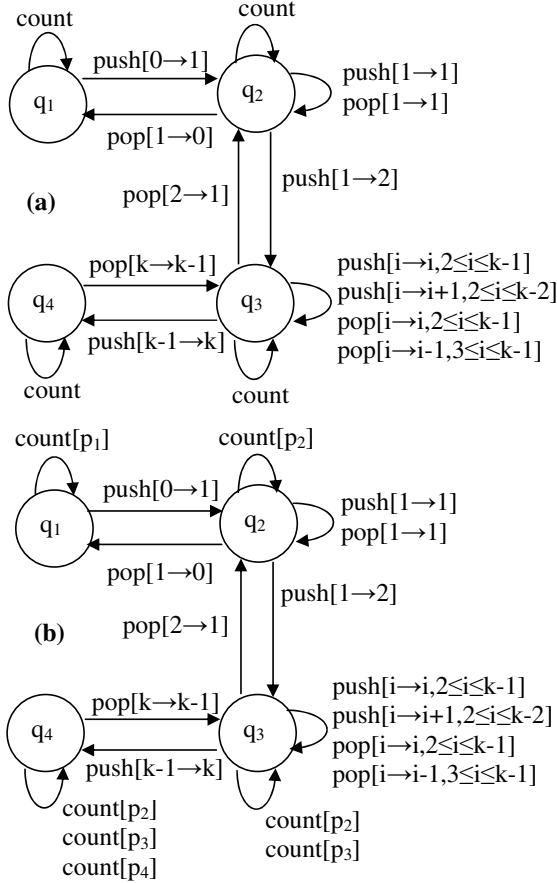


Figure 3. State diagram for *count*

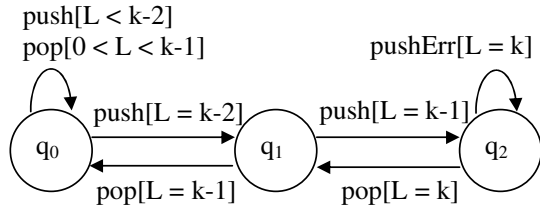


Figure 4. State diagram for *push*

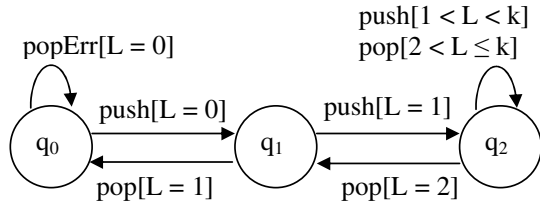


Figure 5. State diagram for *pop*

**Construction of  $T$ :** Since only the method  $f$  is tested, only the transitions involving  $f$  (we will call these  $f$ -transitions) will be considered. For the state diagram of *count* represented in Fig. 3(b), the transitions that need to be checked are  $count[p_1]$  from  $q_1$ ,  $count[p_2]$  from  $q_2$ ,  $count[p_2]$  and  $count[p_3]$  from  $q_3$ , etc. Thus  $T = \{count[p_1]\} \cup \{push[0 \rightarrow 1]\}\{count[p_2]\} \cup \{push[0 \rightarrow 1] push[1 \rightarrow 2]\}\{count[p_2], count[p_3]\} \cup \{push[0 \rightarrow 1] push[1 \rightarrow 2] push[k-1 \rightarrow k]\}\{count[p_2], count[p_3], count[p_4]\}$ .

**Construction of  $W$ :** In practice, an *abstract oracle* is often used instead of  $W$ . An abstract oracle associates with each state  $q$  a predicate capable of deciding if the current state of the object is indeed  $q$ . It is straightforward to construct the predicates for the *push* (Fig. 4) and *pop* (Fig. 5) SXMs; these are  $L < k-1$ ,  $L = k-1$ ,  $L = k$  and  $L = 0$ ,  $L = 1$ ,  $L > 1$ , respectively. An abstract oracle for the *count* SXM is slightly more complex; it involves counting the number of distinct elements in the stack. When an abstract oracle is too expensive, the states can be distinguished by  $f$ -transitions, as explained in what follows. First, let us observe that if the domain of the method  $g$  is partitioned by predicates  $p_1, \dots, p_k$ , one would normally expect  $f$  to produce distinct outputs on the subdomains identified by these predicates. More precisely, for every two distinct predicates  $p_i$  and  $p_j$  and for every memory values  $m_i, m_j \in M$  and parameter value  $input \in Input_f$ , if  $p_i(m_j, input)$  and  $p_j(m_j, input)$  then  $f$  will produce distinct outputs when  $input$  is applied in  $m_i$  and  $m_j$ , respectively. In this case we say that the  $f$ -transitions are *output-distinguishable*.<sup>10</sup>

In the worst case, output-distinguishability can be achieved through suitable debug messages that are removed after testing has been completed. Now, it can be observed that for each pair of distinct states  $q$  and  $q'$ , there is some predicate  $p_i$  that can be satisfied for memory values in  $q$  but not for memory values in  $q'$  (or vice versa). Thus, in order to identify the end state  $q$  of a sequence  $s$  from  $T$ ,  $s$  is concatenated with the  $f$ -transitions defined from  $q$ ; e.g. the sequences leading to  $q_1$  in the *count* SXM are concatenated with  $count[p_1]$ , etc. Therefore,  $W$  can be constructed from  $f$ -transitions.

Once a set of test sequences  $Y = TW$  has been generated, the definitions of  $\Phi$  can be used to identify the sequence of inputs (method names and parameter values) that drives a sequence of transitions, and so each se-

<sup>10</sup>In the SXM literature [7, 15, 16, 18],  $\Phi$  is usually called output-distinguishable when every two distinct processing functions produce distinct outputs on any given memory/input pair. However, this definition is given for the case in which the state and the memory are determined by separate variables and so identical memory values can be attained in different states. Thus, the same memory/input pair can be used to distinguish between states. In a SXM model of a method, the state space is obtained by partitioning the memory values, so the state is dependent on memory. Consequently, distinguishing states entails distinguishing memory values that belong to different classes of the partition.

quence of transition labels from  $Y$  is translated into a sequence of program statements and an appropriate test program is written. For example, the sequence  $push[0 \rightarrow 1] push[1 \rightarrow 2]count[p_3]$  derived from Fig. 3(b) will be transformed into the sequence of program statements  $push(item)push(item')count(2)$ , where  $item$  and  $item'$  represent two distinct elements of  $Item$ . The outputs produced by the test program will then be compared to the expected outputs.<sup>11</sup>

There is one more problem left to be addresses though. It concerns sequences of transition labels that cannot be driven by any sequence of inputs. Consider, for example, the  $push[0 \rightarrow 1] push[1 \rightarrow 2] push[k - 1 \rightarrow k]$  sequence from Fig. 3(b). Clearly, if  $k > 3$  then the sequence can never be exercised even though it is a valid path in the state diagram. The sequences of transition labels which can be driven by inputs are said to be *realisable* and a SXM in which all paths are realisable is called *controllable* [18].

One way to achieve controllability is by designing special methods (mutators) to set up the appropriate context for the problematic transitions. In our case, a mutator that produces a stack with  $k - 1$  distinct elements will be implemented and inserted in the sequence, before  $push[k - 1 \rightarrow k]$ . The drawback of this approach is that it may require a non-negligible number of additional methods that will have to be implemented. Furthermore, their implementation must be correct, as the testing of the other methods relies on this. Another solution is to suitably adjust the initial memory value (i.e. the initial values of class attributes). In the above example, it can be observed that the path becomes realisable if tested for  $k = 3$ .

A third alternative is to expand the non-realisable path (and so construct a tree in breadth-first fashion) until the problematic transition is triggered. In our example, the sequence  $push[0 \rightarrow 1] push[1 \rightarrow 2]$  is expanded until  $push[k - 1 \rightarrow k]$  can be applied; the solution found (i.e. the shortest sequence that meets the termination criterion) is  $push[0 \rightarrow 1] push[1 \rightarrow 2] \dots push[k - 1 \rightarrow k]$ . If this approach is to be automated, then the complexity of the associated algorithm can be reduced by using criteria for pruning and discarding non-useful paths in the tree. One criterion is to prune a path if it has encountered the same memory value twice, the rational being that we have returned to a memory configuration that has already been considered. Based on this criterion, in our example all paths that contain a  $push[i \rightarrow i + 1]$  followed by a  $pop[i + 1 \rightarrow i]$  or a  $push[i \rightarrow i]$  followed by a  $pop[i \rightarrow i]$  need not be extended any further and so can be discarded. Similarly, transitions labelled  $count[p_i]$ ,  $2 \leq i \leq 3$ , can be ignored since they do not affect the value of the memory. Another strategy is to discard paths that leave the current state without trigger-

ing the problematic transition. In our example, based on this criterion, the  $pop[2 \rightarrow 1]$  transition from state  $q_3$  can be omitted from the tree. Using these criteria, in our example only the path  $push[0 \rightarrow 1] push[1 \rightarrow 2] \dots push[i \rightarrow i + 1]$ , which leads to the solution, will be extended.

## 7 Related work

The derivation of a state diagram from a (formal) specification for testing purposes has been considered before in the literature [4, 11, 13, 14, 21, 23]. Dick et al. [4, 13] provide a method for constructing a FSM from a model-based specification using partition analysis, by reducing the specification to disjunctive normal form. Hierons [14] describes the generation of a FSM from a Z specification by rewriting the specification as disjointed input and output predicates (preconditions and postconditions). Murray et al. [11, 21, 23] extend the work of Dick et al. by using an object-oriented notation, object Z.

Although this paper also uses the idea of identifying the states of the diagram by partitioning the values of the class attributes, there are some major differences. Firstly, our approach does not require any kind of formal specification, but only a domain partitioning of each class method to be tested, obtained through some functional technique.<sup>12</sup> The original test cases, along with a (possibly informal) description of the inputs, outputs and memory can be regarded themselves as the specification. This goes down very well with agile methodologies [1] such as eXtreme Programming [3]. Other major differences are in the way the state diagram is built. We construct one diagram for each method to be tested, whereas previous approaches derive a composite diagram of the class, by combining the test cases of individual methods. The composite diagram for a class suffers from a combinatorial state explosion problem that may lead to a much larger test set than that obtained by cumulating the test sets for individual diagrams; furthermore, many of the test sequences derived from a composite state diagram are not particularly useful since they exercise methods based on test criteria for other methods. Finally, although the aforementioned previous approaches use domain-partitioning in the construction of the state diagram, they do not make fully use of it. A method is associated with at most one transition between any two given states; the transition is not “refined” according to the original partitioning (as in the 4th step of our approach).

We illustrate the above points with the bounded stack example. The states of the composite diagram are obtained by removing the overlapping of the subdomains of  $M$  that determine the states of the individual diagrams. Thus, if

<sup>11</sup>As discussed earlier, this minimal oracle can be used in conjunction with an abstract oracle.

<sup>12</sup>If test generation is to be automated, naturally, (formal) models of method behaviour have to be available as well.

$k > 4$  then 9 states will be obtained ( $L$  denotes the number of elements in the stack and  $N$  the number of distinct elements):<sup>13</sup>

- $q_1: L = 0, N = 0.$
- $q_2: L = 1, N = 1.$
- $q_3: 2 \leq L \leq k - 2, N = 1.$
- $q_4: 2 \leq L \leq k - 2, 2 \leq N \leq k - 2.$
- $q_5: L = k - 1, N = 1.$
- $q_6: L = k - 1, 2 \leq N \leq k - 1.$
- $q_7: L = k, N = 1.$
- $q_8: L = k, 2 \leq N \leq k - 1.$
- $q_9: L = k, N = k.$

Many of the test sequences generated by applying the  $W$ -method to this diagram are redundant. For example, a *push* transition from  $q_2$  to  $q_4$  is not likely to uncover faults which have not already been found by a *push* transition from  $q_2$  to  $q_3$ ; many more examples can be found. On the other hand, if the diagram has only one loop-back *count* transition, important test cases could be missed out. Indeed, since for input value  $l = 1$  predicate  $p_2$  will hold for any memory value attained in states  $q_3$  and  $q_4$  (i.e. one distinct element will be counted), the  $count[p_3]$  and  $count[p_4]$  transitions may never be exercised.

In order to improve fault detection, stronger variants of state-based criteria, such as full predicate [25], in which all individual clauses in a decision are exercised, and disjunct coverage [10], in which guard conditions are transformed into disjunctive normal form and a separate transition is defined for each disjunct, have been defined and investigated in the literature. Our approach accommodates very well these criteria: the domain partition can be based on the individual guard clauses and so the test sequences derived from the SXM model will check all such clauses. On the other hand, domain partitioning is not only about logical operators. Consider, for example, a simple guard (or clause in a guard)  $x = 0$ . A criterion based on logical operators will yield two values:  $value_1 = 0$  and  $value_2 \neq 0$ . Then, one of the two mutants in which the guard has been replaced by  $x \geq 0$  and  $x \leq 0$ , respectively, will not be killed. On the other hand, tests derived from a conventional partitioning,  $\{\{x \mid x < 0\}, \{x \mid x = 0\}, \{x \mid x > 0\}\}$ , will kill both mutants.

<sup>13</sup>For a slightly finer partitioning of the domains of the three methods, the number of states of the composite diagram will exceed the cumulated number of states of the individual diagrams.

## 8 Experimental results

The approach proposed in this paper has emerged from a series of experiments we have conducted to evaluate the cost effectiveness of state-based testing. We report here the results concerning the two classes used for illustrations in the paper: a bounded stack with 3 methods<sup>14</sup> and a book from a library system. The two classes were implemented in Java and the test suites were executed against mutant programs automatically generated by MuJava [20, 30], with distributions as given in Fig. 6 (for a description of each mutant see [30]).

The implementation of the bounded stack had approximately 90 LOC, for which 219 mutants were generated. Two sets  $T_1$  and  $T_2$  of sequences of transition labels were constructed:  $T_1$  resulted from the application of our method to each of the 3 state diagrams represented in Figs. 3(b), 4 and 5, while  $T_2$  was derived from the application of the  $W$ -method to the (composite) 9-state diagram described in section 7. Depending on the state diagram, the maximum number of elements considered was  $k = 3$ ,  $k = 4$  and  $k = 5$  (i.e. the minimal value allowed in each case). In all cases, minimal oracles, checking the numerical value returned by *count*, the stack element produced by *pop* and a success / fail Boolean for *push*, were complemented by appropriate abstract oracles, checking the state invariants. The results of the experiments are summarised in Table 1; total length represents the total number of method calls, including the methods that check the state invariants; mutation score represents the percentage of killed mutants [10]; test cost measure is calculated as number of mutants killed / total length (this is a variant of the number of mutants killed / number of test cases executed ratio normally used to measure test cost effectiveness [10]). All non-equivalent mutants of *push* and *pop* were killed in each experiment<sup>15</sup>, so the difference in scores was only due to the mutants of *count*. The first row in the table gives the results for a test suite derived from  $T_1$ . The second and third rows provide the results for  $T_2$  in two cases: when in the construction of the test suite the input parameter  $l$  of *count* was always given the value  $l = 1$  (second row) and when  $l$  was always given the value  $l = k$  (third row). For  $l = 1$  only predicates  $p_1$  and  $p_2$  (see section 4) were checked, whereas for  $l = k$  each of the four predicates  $p_1, p_2, p_3$  and  $p_4$  was exercised in at least one state. The fourth row in the table gives the results for an improved test suite derived from  $T_2$ , in which each *count* transition of the 9-state diagram is partitioned according to the predicates  $p_1, p_2, p_3$  and  $p_4$  (as in the 4th step of our approach) and each non-empty class of the partition is checked.

<sup>14</sup>Beside these, the constructor, oracle methods and other private methods have been implemented.

<sup>15</sup>This validates our intuition that the state-diagrams fully capture the behaviour of *push* and *pop*.

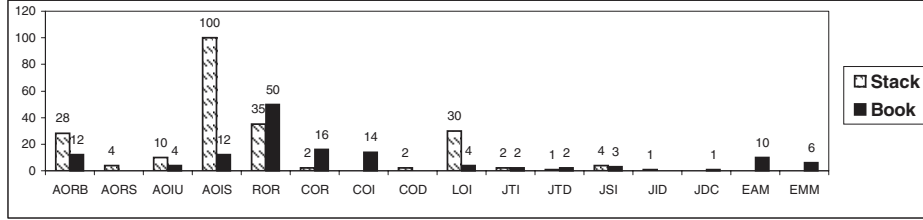


Figure 6. Mutant distribution

These results show that tests derived from the composite diagram for the class are less cost effective than those constructed using our approach. In the best case scenario, however, it appears as though test derived from the composite diagram may achieve a significantly improved fault detection. One possible interpretation of this result is related to the test size: since test suites derived from the composite state diagram are significantly larger, statistically, they should detect more faults, especially when the best case scenario is considered. There is, however, another explanation: the partitioning criterion used in the construction of the state diagram for *count* was too simplistic; it was based only on the result returned by *count*. Based on this criterion, only tests that check stacks without duplicates may be generated (this is the case for the experiment reported in the first row of the table). Consider a stronger criterion, which combines the original criterion with the requirement that both stacks with and without duplicates must be checked. Based on this stronger criterion, the memory subdomains that correspond to states  $q_2$  and  $q_3$  in Fig. 3(b) will be further divided into  $q_{21}$ ,  $q_{22}$  and  $q_{31}$ ,  $q_{32}$ , respectively:

- $q_{21}$ :  $N = L = 1$ .
- $q_{22}$ :  $N = 1$  and  $2 \leq L \leq k$ .
- $q_{31}$ :  $2 \leq N \leq k - 1$  and  $N = L$ .
- $q_{32}$ :  $2 \leq N \leq k - 1$  and  $N < L \leq k$ .

The results of an experiment in which a set  $T_3$  of sequences of transition labels was derived from this 6-state diagram and the *push* and *pop* diagrams represented in Figs. 4 and 5, respectively, are given in the fifth row of the table. It can be observed that the fault detection of the resulting test suite equals that of the improved test suite derived from the composite state diagram of the class. Admittedly, a larger state-diagram of the class can be derived by using the stronger partitioning criterion for the domain of *count* and the resulting test suites may, in the best case scenario, have a slightly better fault detection, but this will be far outweighed by the massive increase in test size.

An interesting observation concerns the mutants that remain undetected. These fall into three categories: equiva-

Experiment	Test suite	No. of sequences	Total length	Mutation score	Test cost measure
$E_1$	$T_1$	15	64	82%	2.81
$E_2$	$T_2$	31	153	74%	1.07
$E_3$	$T_2$	31	153	89%	1.29
$E_4$	$T_2$	31	160	94%	1.28
$E_5$	$T_3$	18	75	94%	2.75

Table 1. Mutation scores for stack

lent mutants, mutants for which stronger criteria are needed and class mutants [20]. There are 7 equivalent mutants out of the total 219. Since these are functionally equivalent to the original implementation, they can actually be removed from the counting. One mutant, missed by all tests, could have been killed by invalid boundary values:  $l \leq 0$ . Similarly, another mutant for  $E_5$  could have been picked by boundary values  $l \geq k > L > 0$ . For other mutants, 3 for  $E_4$  and 1 for  $E_5$ , respectively, more specialised stack configurations (e.g. 'abb' and 'aba') would have been needed. Finally, the 3 class mutants that remain undetected could be killed by fairly simple tests in which more than one instance is used.

A second case study compared the approach proposed in this paper with full predicate coverage [25]. The experiment used the Java implementation (approximately 110 LOC) of a slightly more complex version of the *book* class described in sections 3 and 4, in which two types of books and customers (regular and premium) may exist; 136 mutants of this implementation were generated. Our approach yielded three 4-state diagrams (for *borrow*, *extend* and *reserve*) and two 2-state diagrams (for *return* and *cancel*). The tests based on the full predicate coverage ( $E_1$ ) were derived from a 4-state transition diagram. In both experiments abstract oracles were used to check the state of the *book*. The results are summarized in table 2. Out of the 19% mutants that remained undetected by our method ( $E_2$ ), approximately 8% were equivalent mutants, 2% were class mutants that could be killed by using two separate instances and 9% were

Experiment	No. of sequences	Total length	Mutation score	Test cost measure
$E_1$	27	83	79%	1.28
$E_2$	23	67	81%	1.64

**Table 2. Mutation scores for book**

caused by an attribute, dueDate, for which no obvious partitioning criterion was found and, consequently, did not influence the state diagrams. In general, such mutants can only be picked by a precise oracle. Similar results were obtained for a Java implementation of a vending machine.

## 9 Conclusions

This paper presents an approach for constructing state diagrams for class methods, based on domain partitioning, and also a technique for generating tests from the resulting diagrams. The technique is simple, it does not require extensive formal training and so could be easily adopted in industrial software development. Furthermore, experimental evidence show that it is more cost effective than previous approaches based on the construction of a composite state diagram of the class and/or stronger state-based criteria.

Further work concerns the derivation of hierarchical and concurrent statecharts. Generalisations of the *W*-method for such compound diagrams have been devised [17] and could be adapted to the approach given here. Test generation from state diagrams for object-oriented features, such as polymorphism and inheritance, will also be investigated.

## References

- [1] S. Ambler. *Agile Modeling*. John Wiley, 2002.
- [2] G. Antonioli, L. C. Briand, M. D. Penta, and Y. Labiche. A case study using the round-trip strategy for state-based class testing. In *Proc. ISSRE '02*. IEEE Computer Society, 2002.
- [3] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.
- [4] J. Bicarregui, J. Dick, B. Matthews, and E. Woods. Making the most of formal specification through animation, testing and proof. *Sci. Comput. Program.*, 29(1-2):53–78, 1997.
- [5] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools, Object Technology*. Addison-Wesley, October 1999.
- [6] K. Bogdanov and M. Holcombe. Statechart testing method for aircraft control systems. *Softw. Test., Verif. Reliab.*, 11(1):39–54, 2001.
- [7] K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, and S. Vanak. Testing methods for X-machines: a review. *Form. Asp. Comput.*, 18(1):3–30, 2006.
- [8] G. Booch. Object-oriented development. *IEEE Trans. Softw. Eng.*, 12(2):211–221, 1986.
- [9] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.
- [10] L. C. Briand, M. D. Penta, and Y. Labiche. Assessing and improving state-based class testing: A series of experiments. *IEEE Trans. Softw. Eng.*, 30(11):770–793, 2004.
- [11] D. A. Carrington, I. MacColl, J. McDonald, L. Murray, and P. A. Strooper. From Object-Z Specifications to ClassBench Test Suites. *Softw. Test., Verif. Reliab.*, 10(2):111–137, 2000.
- [12] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.
- [13] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME*, pages 268–284, 1993.
- [14] R. M. Hierons. Testing from a Z Specification. *Softw. Test., Verif. Reliab.*, 7(1):19–33, 1997.
- [15] R. M. Hierons and M. Harman. Testing conformance of a deterministic implementation against a non-deterministic stream X-machine. *Theoretical Comput. Sci.*, 323(1-3):191–233, 2004.
- [16] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer Verlag, 1998.
- [17] F. Ipate. Test selection for hierarchical and communicating finite state machines. Submitted to *The Computer Journal*.
- [18] F. Ipate. Testing against a non-controllable stream X-machine using state counting. *Theoretical Comput. Sci.*, 353(1-3):291–316, 2006.
- [19] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.
- [20] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.
- [21] I. MacColl, L. Murray, P. A. Strooper, and D. A. Carrington. Specification-based class testing: A case study. In *ICFEM*, pages 222–231, 1998.
- [22] B. Meyer. Design by contract. *IEEE Computer*, 25(10):40–51, 1992.
- [23] L. Murray, D. A. Carrington, I. MacColl, J. McDonald, and P. A. Strooper. Formal derivation of finite state machines for class testing. In *ZUM*, pages 42–59, 1998.
- [24] A. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *UML*, pages 416–429, 1999.
- [25] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Softw. Test., Verif. Reliab.*, 13(1):25–53, 2003.
- [26] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, 1988.
- [27] M. Roper. *Software Testing*. McGraw-Hill, 1995.
- [28] J. E. Rumbaugh, M. R. Blaha, W. J. Premerlani, F. Eddy, and W. E. Lorenzen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [29] D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Trans. Softw. Eng.*, 15(4):413–426, 1989.
- [30] <http://www.ise.gmu.edu/~offutt/mujava/>. MuJava home page.