A technique for model-based testing of a class

Aditya V. Nori *
Tata Research Development and Design Centre
Pune, India
nori@csa.iisc.ernet.in

Ashok Sreenivas
Tata Research Development and Design Centre
Pune, India
ashoks@pune.tcs.co.in

ABSTRACT

Classes form the basic unit of abstraction in the object-oriented (OO) programming paradigm of software development. Therefore, testing of classes is fundamental to testing an OO software system.

In this paper, we present a technique to automatically test a class based on an object-model specification. The novelty of the technique is that it defines a notion of *coverage* for specification-based testing. Two forms of coverage are defined: *edge coverage* and *dependence coverage*. These coverages guarantee that all methods of the class and all pair-wise interactions between methods are tested.

Algorithms to generate test-cases from the specification are presented, and we show that these algorithms guarantee the two kinds of coverage. The algorithms are polynomial in the worst-case and therefore, promise good performance in practice.

1. INTRODUCTION

Object-oriented (OO) programming is currently the most popular paradigm in use for software development. Classes form the basic unit of abstraction in an OO system. Therefore, testing of classes is of fundamental importance to testing such systems.

In the past, [16, 8, 12, 6, 2, 5] have studied the problem of testing OO systems at different levels of abstraction. Algebraic specifications [10] provide a abstract-data type based formalism for the OO paradigm, and much of the literature on specification-based testing of OO systems is based on algebraic specifications. However, algebraic specifications are rarely used in practice to formally specify OO systems. A (semi-)formal notation that has wide acceptance is object-modelling, with the UML [17] being very popular.

*Current address: Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India

In this paper, we present a technique to automatically test a class based on UML-like specifications. We define notions of *coverage* over class specifications, including coverage over interactions of methods in a class. Algorithms which generate method-sequences to guarantee the defined coverages are also given.

In Section 2, we present a formal definition of a class specification, which is based on object-modelling. Section 3 first presents an 'ideal' way to test a class, and then presents a practical alternative by defining two kinds of coverage over the specification. Algorithms to generate test-cases based on the coverages are presented in Section 4. The algorithms are analysed in Section 5. In Section 6, we discuss related work in class-testing and compare our technique with others.

2. CLASS SPECIFICATION

The specification of a class can be broken into two components: a structural component, and a behavioural component. We define each of these below.

DEFINITION 1. *The structural specification $\mathcal{S}_C$ of a class C is a three-tuple (A, M, ι) where*

- *A is set of class attributes $\{a_i : \tau_i\}$, where the a_i are the names of attributes and τ_i are the corresponding attribute types. A also includes attributes inherited from the parent class(es) of the class under consideration.*
- *M is a set of method specifications $\{\langle \pi_{m_i}, m_i :: p_1 : \tau_1 \times \cdots \times p_n : \tau_n \to \tau_i, \psi_{m_i} \rangle\}$. Each triple represents a method with an associated* pre-condition *and* post-condition.

  $m_i :: p_1 : \tau_1 \times \cdots \times p_n : \tau_n \to \tau_i$ is the signature *of method m_i with return type τ_i, parameters p_j and their associated types τ_j. Parameters $p_j : \tau_j$ are partitioned into input parameters, input(m_i) and output parameters, output(m_i).*

  π_{m_i} is the pre-condition of method m_i, and is a boolean expression over the attributes A and input parameters input(m_i). Each π_{m_i} represents a condition over A and input(m_i) that must be satisfied for a valid invocation of m_i.

  ψ_{m_i} is the post-condition of method m_i, and is a boolean expression over A, input(m_i) and output(m_i). ψ_{m_i} must be satisfied at the end of a successful invocation of m_i.

- *ι is an invariant over the class state, and must be satisfied before and after the application of each method m of the class. It is specified as a boolean expression over the class-attributes A.*

This class specification describes the structure of a class in an object model. The post-conditions ψ_m may refer to values of attributes and (input) parameters both *before* the invocation of m and *after* it. To distinguish between the two values of an attribute or parameter a, we use the notation a' in a post-condition to refer to its value before invocation and a to refer to its value after method execution. In a pre-condition expression, a refers to the value before invocation of the method.

Note that the above class specification only considers 'pure' object-oriented specifications, with no 'globals' or 'class attributes'.

A complete description of a class requires a description of its *behaviour* in addition to a structural description (Definition 1). We now define the behavioural specification of a class.

DEFINITION 2. *The behavioural specification $\mathcal{B}_C$ of a class C is given by a set of four-tuples $\{\langle \pi_{M_i}, M_i, \psi_{M_i}, \iota_{M_i} \rangle\}$ where*

- *Each M_i is a finite state machine (FSM) $(Q_i, \Sigma, i_i, f_i, \delta_i)$ representing a behavioural scenario of the class. Q_i is the set of states of M_i where each state corresponds to a 'logical' state of the class. Σ is the input alphabet consisting of the set of methods m_i defined in the structural specification. $i_i \in Q_i$ is the unique start state of M_i and $f_i \in Q_i$ is the unique final state of M_i. $\delta_i : \Sigma \times Q_i \to 2^{Q_i}$ is the (possibly non-deterministic) state transition function. Each transition corresponds to a method invocation which takes the class from one logical state to a set of possible logical states.*

  The graph G defined by M_i has nodes corresponding to states from Q_i and edges labelled with methods m_i defined in the structural specification. G is connected, so that all states are reachable from i_i and that f_i is reachable from all states.

- *π_{M_i} is the pre-condition which must hold true for the scenario depicted by M_i to be valid, and is a boolean expression over the attributes A of the class.*

- *ψ_{M_i} is the post-condition for a scenario that must hold true at the final state of M_i, and is a boolean expression over the attributes A of the class.*

- *ι_{M_i} is the invariant corresponding to the scenario represented by M_i.*

Each FSM M_i represents a 'behavioural scenario' of the class, and the set $\{M_1 \cdots M_n\}$ together represent the behaviour of the class under all possible scenarios. The set of strings accepted by the FSM M_i is the set of *valid* method invocation sequences in the scenario depicted by M_i. Thus, the set of all strings accepted by the FSMs $\{M_1, \cdots, M_n\}$ is the set of all possible valid method invocation sequences over the class.

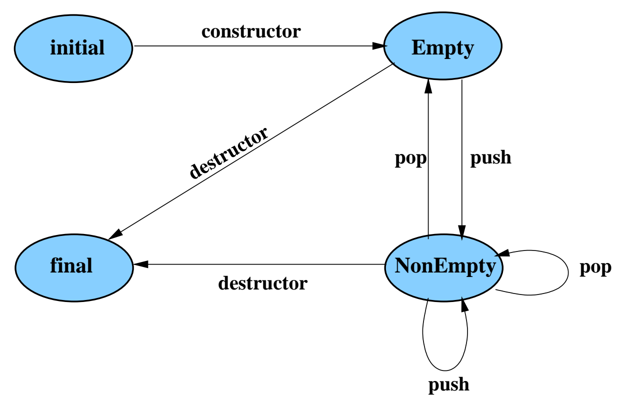

Figure 1: FSM for the unbounded stack example

DEFINITION 3 (CLASS SPECIFICATION). *The specification of a class C is a two-tuple $\langle \mathcal{S}_C, \mathcal{B}_C \rangle$ where $\mathcal{S}_C$ is the structural specification (Definition 1) of the class and $\mathcal{B}_C$ is its behavioural specification (Definition 2).*

EXAMPLE 1. *A simple unbounded stack S may be specified as:*

$$
\begin{array}{ll}
S & = \langle \mathcal{S}_S, \mathcal{B}_S \rangle \\
 & \\
\mathcal{S}_S & = \langle A_S, M_S, \iota_S \rangle \\
\mathcal{B}_S & = \{\langle \pi_{M_0}, M_0, \psi_{M_0}, \iota_{M_0} \rangle\} \\
 & \\
A_S & = \{a : list\ of\ int,\ tos : int\} \\
M_S & = \{m_1, m_2, m_3, m_4\} \\
m_1 & = \langle true,\ constructor :: \bot^1 \to \bot, tos = 0 \wedge a = empty \rangle \\
m_2 & = \langle true,\ destructor :: \bot \to \bot, true \rangle \\
m_3 & = \langle true,\ push :: e : int\ \to \bot, \\
 & \quad (a = a'.append(e)) \wedge (tos = tos' + 1) \rangle \\
m_4 & = \langle tos \neq 0,\ pop :: \bot \to int, \\
 & \quad (a' = a.append(pop())) \wedge (tos' = tos + 1) \rangle \\
\iota_S & = true
\end{array}
$$

The graph shown in Figure 1 represents M_0.

$$
\begin{array}{ll}
\pi_{M_0} & = true \\
\psi_{M_0} & = true \\
\iota_{M_0} & = true
\end{array}
$$

□

3. CLASS TESTING AND COVERAGE

Given the specification of a class C (Definition 3), it may be tested as follows:

- From the FSMs in $\mathcal{B}_C$, construct the set S of all valid sequences of method invocations over C.

[1] $\bot$ represents the 'void' type.

- For each $s \in S$
  - Let $s = m_0 \cdots m_n$ and M_i be the FSM from which sequence s was generated.
  - Generate data to satisfy $\pi_{m_0} \wedge \pi_{M_i}$.
  - Invoke methods in s and ensure that the following conditions are satisfied.
  - π_{m_i} is satisfied before execution of m_i for all $1 \leq i \leq n$,
  - ψ_{m_i} is satisfied after execution of m_i for all $0 \leq i \leq n$,
  - ψ_{M_i} is satisfied after execution of s
  - $\iota_{M_i} \wedge \iota_C$ is satisfied after execution of m_i for all $0 \leq i \leq n$

This form of testing is thorough because it tests every possible sequence of method invocations over C. However, as a strategy, it is impractical, because the number of valid sequences of method-invocations over a class can be infinitely large. For example, arbitrarily long method invocation sequences can be generated for the stack class of Example 1 by alternating between the `push` and `pop` methods.

The above example also illustrates that all possible method invocation sequences *need not* be tested for a class to be tested successfully. A class can be declared to be successfully tested, if the following can be guaranteed:

Criterion 1 Each method in the class has been tested.

Criterion 2 Different interactions between methods have been tested.

Therefore, a pragmatic approach to testing a class requires the selection of an appropriate set of method invocation sequences which would 'exercise' the class such that the above criteria can be met.

In the remainder of this section, we formally define 'interactions' between methods, and define coverage over the class specification which enables selection of such a set of method invocation sequences.

DEFINITION 4 (DEFINITION/USE TRANSITIONS). *Given a class specification $C = \langle \mathcal{S}_C, \mathcal{B}_C \rangle$, a transition or method invocation m in a FSM $M \in \mathcal{B}_C$*[2] *is said to* define *an attribute (or parameter) a, iff a occurs in ψ_m. Similarly m is said to* use *a iff a occurs in π_m, or a' occurs in ψ_m.*

A method m *defines* a if the post-condition of m depends on the value of a after the execution of m. In other words, a plays a role in the way the state is affected by executing m.

Similarly, m *uses* a if the value of a determines whether m can be invoked correctly, i.e. a occurs in m's pre-condition or a' (the value of a before the execution of m) appears in the post-condition of m. The second condition is required to account for the situation where the state after execution of m depends on the pre-invocation value of an attribute (or parameter); implying that m uses the value that a had before m was invoked.

[2] For notational simplicity, we say FSM $M \in \mathcal{B}$ to denote $\langle \pi_M, M, \psi_M, \iota_M \rangle \in \mathcal{B}$, when the other elements of the quadruple are not of interest to us.

DEFINITION 5 (DEPENDENCE PAIRS OF TRANSITIONS). *Given a class specification $C = \langle \mathcal{S}_C, \mathcal{B}_C \rangle$, methods m_1 and m_n form a dependence pair for an attribute a, denoted as $(m_1, m_n)_a$, iff:*

- *there exists an FSM $M_i \in \mathcal{B}_C$ with transitions labelled m_1 and m_n,*
- *a is defined by m_1 and used by m_n,*
- *there is at least one sequence of transitions $m_1, m_2 \cdots m_n$ in M_i from m_1 to m_n such that none of the transitions $m_2 \cdots m_{n-1}$ defines a.*

A dependence pair $(m_1, m_2)_a$ indicates that methods m_1 and m_2 interact with each other through a in the scenario of the class represented by machine M_i. This is because the existence of such a dependence pair implies that the value of attribute a as determined by m_1 is used unchanged by m_2. A dependence pair of methods $(m_1, m_2)_a$ is analogous to the *def-use* pairs of data-flow analysis [1], with a 'definition' and 'use' being as given by Definition 4 and the 'control flow graph' being replaced by a FSM M_i.

With this definition of dependence-pairs, we now define two kinds of coverage over a class specification.

DEFINITION 6 (EDGE COVERAGE). *Given a class specification (Definition 3), $C = \langle \mathcal{S}_C, \mathcal{B}_C \rangle$, and a FSM $M_i \in \mathcal{B}_C$, a set S of method invocation sequences* edge-covers *M_i iff every transition $m \in M_i$ belongs to at least one sequence $s \in S$.*

In other words, an edge-covering set of sequences S guarantees that all methods involved in a scenario described by M_i belong to at least one of the sequences in S.

DEFINITION 7 (DEPENDENCE COVERAGE). *Given $C = \langle \mathcal{S}_C, \mathcal{B}_C \rangle$, a class specification, and a FSM $M_i \in \mathcal{B}_C$, a set S of method invocation sequences* dependence-covers *M_i iff for every dependence pair $(m_j, m_k)_a$ of M_i, there is some sequence $s \in S$ such that $s = m_0 \cdots m_j \cdots m_k \cdots m_n$ traces a path in M_i and none of the transitions in $m_{j+1} \cdots m_{k-1}$ define a.*

A dependence-covering set of sequences S for a scenario M_i guarantees that all pairs of methods (m_1, m_2) which interact through any attribute a, belong to at least one sequence $s \in S$, i.e. S encapsulates all possible pairs of interacting methods.

We now define a set of covering sequences of a class based on these coverage criteria.

Definition 8 (Covering sequences). *Given a class specification* $C = \langle \mathcal{S}_C, \mathcal{B}_C \rangle$, *a set* S *of method invocation sequences covers* C *iff*

- S *edge-covers* $M_i, \forall M_i \in \mathcal{B}_C$ *and*
- S *dependence-covers* $M_i, \forall M_i \in \mathcal{B}_C$

We denote the subset of S that edge-covers all M_i as S_E and the subset that dependence-covers all M_i as S_D. Note that S_E and S_D need not be disjoint.

A set of method invocation sequences S that covers a class C edge-covers all FSMs M_i of C, and is guaranteed to test all methods that appear as transitions in any M_i. Therefore, all methods in the public interface of the class that appear in at least one scenario are tested. This shows that testing a class using S satisfies criterion 1 mentioned above about testing all its methods.

A set of method invocation sequences S that covers a class C dependence-covers all FSMs M_i of C. Therefore, S is guaranteed to contain sequences to test all possible dependence-pairs (Definition 5) of methods for any attribute and in any scenario. This shows that testing a class using S satisfies criterion 2 mentioned above about testing all possible method interactions.

Therefore, a set of covering sequences S (Definition 8) satisfies both the pragmatic criteria laid down above, and provides a good basis to test a class.

4. GENERATING COVERING SEQUENCES

We now present algorithms to generate covering sequences of method invocations over a class (Definition 8). Algorithm 1 generates a sequence S_E of method invocations guaranteeing edge-coverage (Definition 6) and Algorithm 2 generates a sequence S_D of method invocations guaranteeing dependence-coverage (Definition 7). These algorithms treat the FSMs M_i of the behavioural specification as graphs, with nodes representing the states of the FSM and edges, labelled with the class methods, representing the transitions. To simplify the presentation of the algorithms, we outline two variants dfs' and dfs'' of the standard depth-first search (DFS) algorithm dfs over graphs that returns a DFS tree for the graph.

dfs' is a normal DFS function over a graph (FSM) G which returns a two-tuple $\langle T, E \rangle$ where T is the usual DFS tree, and E is the set of edges in G which are not part of T, i.e. the set of non-tree edges of G.

$dfs''(G, n, a)$ takes a FSM graph G, a state node n and an attribute a as parameters, and builds a DFS tree of G rooted at n but not containing any edge (or transition or method) which defines a (Definition 4). This ensures that the tree returned by dfs'' is 'definition-clear' for attribute a, so any path through this tree does not 're-define' a, and contains all uses u of a reachable from n.

We also define a function *ComputePath*. This function accepts a (DFS) tree, T, a node n in the tree, a 'direction' d which takes values *td* (for 'top-down') or *bu* (for 'bottom-up'). If the direction is *td*, it returns the sequence of edges from the root of the tree to n. Otherwise, it returns the sequence of edges from n to the root of the tree.

We now present the algorithm to generate the set of method invocation sequences S_E guaranteeing edge-coverage over a FSM.

Algorithm 1 (Edge coverage algorithm).

EdgeCovPathGen :: *FSM* $\rightarrow$ *Set of method invocation sequences*
EdgeCovPathGen(M_i)

1. *Let* $M_i = \langle Q_i, \Sigma, i_i, f_i, \delta_i \rangle$ *and* G *be the graph corresponding to* M_i
2. $\langle T, E \rangle = dfs'(G)$
3. $T^R = dfs(G^R)$ // G^R *is* G *with its edges reversed*
4. $S_E = \{ComputePath(T, f_i,\ td)\}$
5. **forall** $e \in E$
   do
6. $\quad S_E = S_E \cup \{$
   ComputePath $(T, source(e),\ td)$.
   e .
   ComputePath $(T^R,\ target(e),\ bu)$ $\}$
   od
7. **return** S_E

The function **EdgeCovPathGen** first uses dfs' to compute the DFS tree T and the set of non-tree edges E of the graph G corresponding to the FSM M_i. It initialises S_E to a simple tree path going from the root of T (i.e. i_i) to the final state f_i. Then, for each non-tree edge $e \in E$, it adds a path to S_E computed as follows:

- an initial segment from i_i to the source(e), which is got from T,
- a middle segment which is the edge e and
- a final segment which is the path from $target(e)$ to f_i which is got from T^R.

Thus, all paths added to S_E begin at i_i and end at f_i. We present a proof of Algorithm 1 in Appendix A.

The problem of generating a sequence S_D of method invocations guaranteeing dependence-coverage depends upon identifying dependence-pairs of transitions. This problem can be reduced to the *def-use* data-flow analysis problem [1], with the FSM M_i being the 'control-flow-graph', and definitions and uses are as given by Definition 4. The set of def-uses can be obtained using any standard data-flow analysis technique such as the iterative or the elimination techniques. Once the dependence-pairs of methods have been computed, the desired sequences of method invocations which guarantee dependence-coverage can be generated using Algorithm 2 given below.

ALGORITHM 2 (DEPENDENCE COVERAGE ALGORITHM).

DepCovPathGen :: *FSM* $\rightarrow$ *set of method invocation sequences*
DepCovPathGen(M_i)

1. *Let* $M_i = \langle Q_i, \Sigma, i_i, f_i, \delta_i \rangle$ *and* G *be the graph corresponding to* M_i
2. $T = dfs(G)$
3. $T^R = dfs(G^R)$
4. $DU\text{-}set = \phi$
5. **forall** $d \in$ *the set of defining transitions in* M_i
   do
6. $\quad$ $A =$ *the set of attributes defined by* d
7. $\quad$ **forall** $a \in A$
   $\quad$ **do**
8. $\quad\quad$ $DU\text{-}set = DU\text{-}set \cup \{\langle d, a, dfs''(G,\ target(d), a)\rangle\}$
   $\quad$ **od**
   od
9. $S_D = \phi$
10. **forall** $\langle d, a, DU_{(d,a)}\rangle \in DU\text{-}set$
    do
11. $\quad$ **forall** $n \in DU_{(d,a)}$ *such that* $n = source(u)$ *and* $(d,u)_a$ *is a dependence-pair*
    $\quad$ **do**
12. $\quad\quad$ $S_D = S_D \cup \{$
    $\quad\quad\quad$ *ComputePath* $(T, source(d),\ td)$.
    $\quad\quad\quad$ d .
    $\quad\quad\quad$ *ComputePath* $(DU_{(d,a)}, source(u),\ td)$.
    $\quad\quad\quad$ u .
    $\quad\quad\quad$ *ComputePath* $(T^R, target(u),\ bu)$ $\}$
    $\quad$ **od**
    od
13. **return** S_D

Function **DepCovPathGen** works as follows. Firstly, the DFS tree for G and G^R are computed into T and T^R respectively. For all definition-attribute pairs (d, a), it computes a triple in Step 8 consisting of d, a and a tree $DU_{(d,a)}$. $DU_{(d,a)}$ is rooted at d and contains definition-clear paths to all uses u of d. For each such triple $\langle d, a, DU_{(d,a)}\rangle$ constructed, Step 12 considers all uses u of the definition d and paths are added to S_D as follows:

- an initial segment from i_i to the source of d obtained from T,
- followed by the edge d,
- followed by the definition-clear path for attribute a from target(d) to source(u) obtained from tree $DU_{(d,a)}$,
- followed by edge u, and
- finally a segment from target(u) to f_i (obtained from T^R).

This sequence of segments results in a path from i_i to f_i which covers the dependence pair $(d, u)_a$.

Note that the DFS walk of Step 2 need visit only nodes that are targets of defining transitions, and not all nodes in G, as we only use T to compute paths to such nodes. Similarly Step 3 need visit only nodes that are source/target nodes of use transitions, and the DFS walk in Step 8 may terminate once all uses of the definition d have been added to the tree being constructed. Restricting the DFS walks as mentioned above would make an actual implementation more efficient than described here, though the worst-case complexity would remain the same. A proof of Algorithm 2 is presented in Appendix A.

Given a sequence of method invocations $s = m_0 \cdots m_n$, generated by either Algorithm 1 or Algorithm 2 from a FSM $M \in \mathcal{B}_C$, and some test-data ϵ which satisfies the preconditions of both m_0 and M, and the invariants of the FSM and the class i.e.

$$\pi_{m_0}(\epsilon) \wedge \pi_M(\epsilon) \wedge \iota_M(\epsilon) \wedge \iota_C(\epsilon) = true$$

we present a simple method to test the sequence of method invocations s.

ALGORITHM 3 (TESTING A SEQUENCE s).

TestASequence :: *TESTDATA* $\times$ *sequence of methods* $\rightarrow$ $\{$*success, failure*$\}$
TestASequence(ϵ, s)

1. *Let* $s = m_0 \cdots m_n$
2. $\epsilon_0 = \epsilon$
3. **forall** *i from* 0 *to* n
   do
4. $\quad$ **if** $\neg(\epsilon_i$ *respects* $\pi_{m_i} \wedge \iota_M \wedge \iota_C)$
   $\quad$ **then**
5. $\quad\quad$ **print** π *failure at method* m_i *for* ϵ
6. $\quad\quad$ **return** *failure*
   $\quad$ **fi**
7. $\quad$ $\epsilon_{i+1} = m_i(\epsilon_i)$ $\quad$ *//ϵ_{i+1} is the 'state' after m_i*
8. $\quad$ **if** $\neg(\epsilon_{i+1}$ *respects* $\psi_{m_i} \wedge \iota_M \wedge \iota_C)$
   $\quad$ **then**
9. $\quad\quad$ **print** ψ *failure at method* m_i *for* ϵ
10. $\quad\quad$ **return** *failure*
    $\quad$ **fi**
    od
11. **if** $\neg(\epsilon_{n+1}$ *respects* $\psi_M \wedge \iota_M \wedge \iota_C)$
    then

12. **print** ψ_M *failure for* s, ϵ

13. **return** *failure*
else

14. **return** *success*
fi

Algorithm 3 tests a sequence of method invocations given an item of test-data that satisfies the pre-conditions for the sequence and for the first method in the sequence. For a given test-data ϵ, and sequence $s = m_0 \cdots m_n$, it returns success iff all the following conditions are satisfied during **TestASequence**:

1. $\forall i.1 \leq i \leq n.\ \pi_{m_i} = true$
2. $\forall i.0 \leq i \leq n.\ \psi_{m_i} = true$
3. $\psi_M = true$ where M is the FSM for which sequence s was generated
4. $\iota_M \wedge \iota_C = true$ before and after execution of m_i for all $0 \leq i \leq n$.

Appendix A contains a proof that Algorithm 3 returns success iff all the above conditions are met.

Given the specification of a class C (Definition 3), we now present a method to test C based on the algorithms **EdgeCovPathGen**, **DepCovPathGen** and **TestASequence**.

ALGORITHM 4 (METHOD TO TEST A CLASS).

1. Let $C = \langle \mathcal{S}_C, \mathcal{B}_S \rangle$

2. $S_E = S_D = \phi$

3. **forall** $M \in \mathcal{B}_S$
do

4. $S_E = S_E \cup \mathit{EdgeCovPathGen}(M)$

5. Compute dependence pairs of transitions for M

6. $S_D = S_D \cup \mathit{DepCovPathGen}(M)$
od

7. $S = S_E \cup S_D$

8. Fail-set $= \phi$

9. **forall** $s \in S$
do

10. Let $s = m_0 \cdots m_n$

11. Let M be the FSM from which s was generated

12. Generate a set of test-data Υ*, all of whose elements satisfy* $\pi_M \wedge \pi_{m_0} \wedge \iota_M \wedge \iota_C$

13. **forall** $\epsilon \in \Upsilon$
do

14. **if TestASequence**$(\epsilon) =$ *failure*
then

15. *Fail-set* $=$ *Fail-set* $\cup \{\langle \epsilon, s \rangle\}$
fi
od
od

Algorithm 4 first generates the sets of sequences S_E and S_D which respectively edge-cover and dependence-cover all the FSMs in $\mathcal{B}_C$. For each sequence s in $S_E \cup S_D$, it then generates a test-data set in Step 12. The data that is generated represents a valid state of the system to test sequence s, as it satisfies $\pi_{m_0} \wedge \pi_M$ and the invariants ι_M and ι_C. Details of generating the data which satisfies these constraints are outside the scope of this paper and are therefore not discussed. We only state that this data can be generated from $\mathcal{S}_C$ which contains information about C's attributes, m_0's parameters and their associated types. Moreover, this data can be generated so that interesting cases such as boundary-values are covered.

For each element ϵ in the test-data set, the function **TestASequence** is called which checks whether that data passes the test or not. Any data that fails the test is added to a failure-set, which can then be used for problem analysis and bug-fixing. If the failure-set remains empty at the end of Algorithm 4, then the class may be considered to be fully tested, according to the edge- and dependence-coverage criteria.

The correctness of algorithm 4 follows directly from those of Algorithms 1, 2 and 3.

To summarise, a class C that has been successfully tested using Algorithm 4 *guarantees* the following:

- C has been tested on a set of sequences of method invocations which guarantee edge-coverage and dependence-coverage over C. This set of sequences of invocations guarantees that *all* methods have been tested, and all *pair-wise interactions* between methods have been tested.
- Each such sequence $s = m_0 \cdots m_n$, generated from FSM M has been tested on representative data (with respect to types, boundaries, partitions etc.) under the constraint $\pi_M \wedge \pi_{m_0} \wedge \iota_M \wedge \iota_C$
- π_{m_i} of *every* method m_i in *every* sequence s was satisfied for *every* item of test-data, ϵ, used.
- ψ_{m_i} of *every* method m_i in *every* sequence s was satisfied for *every* item of test-data, ϵ, used.
- ψ_M was satisfied for *every* sequence s generated from FSM M, for *every* item of test-data, ϵ, used.
- $\iota_M \wedge \iota_C$ was satisfied before and after every method m_i.

The method outlined above tests the validity of the various possible invocation sequences ‘operationally’. In principle,

one could attempt to prove the validity of the specification. That is, for each FSM M, prove that $psi_{m_1} \Rightarrow \pi_{m_2}$, for all 'adjacent' transitions[3]. Showing that such a property holds for all possible pairs of successive method invocations would help in *model-checking* the specification. However, it would *not* aid in *testing* whether the implementation satisfies the specification. Therefore, it is orthogonal to the discussion in this paper.

EXAMPLE 2. *We exemplify our technique using the class specification of an unbounded stack given in Example 1. The set of edge- and dependence-covering sequences generated for the stack specification by Algorithms 1 and 2 are given below. As all sequences begin with the constructor method and end with the destructor method, we do not include these in the sequences below.*

- $S_E =$ **EdgeCovPathGen**$(M_0) = \{<>, <push>, <push, push>, <push, pop>, <push, pop, push>\}$
- $S_D(\textit{not in } S_E) =$ **DepCovPathGen**$(M_0) = \{<push, push, pop>, <push, push, push>, <push, pop, pop>\}$
- $S = S_E \cup S_D = \{<>, <push>, <push, push>, <push, pop>, <push, pop, push>, <push, push, pop>, <push, push, push>, <push, pop, pop>\}$

It can be seen that the above set of sequences includes all methods and all dependence pairs of methods. For instance, $(push, pop)$ form a dependence pair for attribute tos, which is set by push and used by pop. This results in the sequence $<push, push, pop>$ being generated by **DepCovPathGen** *for the push and pop methods emanating from the* `NonEmpty` *state. Similarly, the sequence $<push, pop>$ is generated by* **EdgeCovPathGen** *to account for the non-tree edge pop from the* `NonEmpty` *state to the* `Empty` *state.*

□

A node (or state) of the FSM represents an abstract state of the class model, to which many concrete states may map. Thus, in Example 1, any concrete state with a non-zero top-of-stack value maps to the `NonEmpty` abstract state. A method invocation (transition in the FSM) on different concrete states which map to the same abstract state, may lead to concrete states which map to different abstract states. Thus, in Example 1, a `pop` method invoked on two different `NonEmpty` states, one with a single element in the stack and one with more than one elements in the stack, would lead to concrete states which map to different abstract states, `Empty` and `NonEmpty` respectively. This is the cause for non-deterministic FSMs being part of $\mathcal{B}_C$.

While generating the set of covering sequences S, Algorithms 1 and 2 only consider transitions between abstract states as defined by the FSM, and cannot consider the underlying concrete states. Therefore, it is possible that the set of sequences S contains some sequence $s = m_0 \cdots m_n$, which when executed on a test-data results in the failure of some π_{m_i}. Such a situation may occur because the sequence was generated assuming that execution of m_{i-1} leads to a state where m_i is a valid method invocation. However, invocation of m_{i-1} may lead to a state where m_i is not a valid invocation. In Example 2, the sequence $<push, pop, pop>$ gets generated because the algorithms are unaware that the first `pop` results in a transition to the `Empty` state and not to the `NonEmpty` state.

Therefore, the set S of sequences generated may contain certain *spurious* sequences which will never pass the test on a correct implementation. We currently have no characterisation of such sequences. However, the number of such sequences is likely to be much smaller than the number of correct sequences. We claim that S is a *conservative* set of method invocation sequences to test class C in the sense that no possible transitions and dependences are missed, though it may produce certain spurious sequences.

[3] m_1 and m_2 are adjacent transitions if the m_1 transition ends in a state n and the m_2 transition begins in state n.

5. ALGORITHM ANALYSIS

For the purposes of this report, the number of states in an FSM is referred to by $\mathcal{N}_M$, the number of transitions by $\mathcal{E}_M$ and the number of defining transitions is represented by $\mathcal{D}_M$. The number of attributes in the class is denoted by $\mathcal{A}$. The number of states in all the FSMs of the specification together is referred to by $\mathcal{N}$ and the number of transitions in all the FSMs together is referred to by $\mathcal{E}$.

Algorithm 1, **EdgeCovPathGen**, performs two DFS walks over the FSM and one walk over the DFS tree for each call to *ComputePath*. *ComputePath* itself is called twice for each non-DFS-tree edge of the FSM. In the worst-case, this amounts to $\mathcal{O}(3\mathcal{N}_M + \mathcal{N}_M\mathcal{E}_M) = \mathcal{O}(\mathcal{N}_M\mathcal{E}_M)$. However, the number of non-tree edges of the FSM are expected to be much lesser than $\mathcal{E}_M$, and therefore the complexity will be lower in practice. The number of sequences returned by **EdgeCovPathGen** is equal to one more than the number of non-tree edges, i.e. $\mathcal{O}(\mathcal{E}_M)$.

The complexity of computing the dependence-pairs using the standard iterate-to-saturate algorithm of data-flow analysis is at worst $\mathcal{O}(\mathcal{N}_M^2)$.

Algorithm 2, **DepCovPathGen**, performs two DFS walks over the FSM, and for each defining transition e, it performs at most $\mathcal{A}$ DFS walks. In addition, it also performs three DFS tree traversals for each use-transition of each <defining transition, attribute defined by the transition> pair. This amounts to a worst-case complexity of $\mathcal{O}(2\mathcal{N}_M + \mathcal{D}_M\mathcal{A}\mathcal{N}_M + \mathcal{D}_M\mathcal{A}\mathcal{N}_M^2)$, or $\mathcal{O}(\mathcal{N}_M^2\mathcal{A}\mathcal{E}_M)$ since $\mathcal{D}_M \leq \mathcal{E}_M$. The number of sequences computed by **DepCovPathGen** is bounded by $\mathcal{O}(\mathcal{N}_M\mathcal{A}\mathcal{E}_M)$, as the number of <defining transition, attribute defined> pairs are $(\mathcal{E}_M\mathcal{A})$ at worst, and the number of sequences added for each such pair is $\mathcal{N}_M$ at worst. However, note that these worst case scenarios can only occur in the rare cases when all transitions define and use all attributes.

The length of each sequence S returned by either **EdgeCovPathGen** or **DepCovPathGen** cannot be more than $\mathcal{O}(\mathcal{E}_{\mathcal{M}})$. This follows from the fact that the maximum length

of a path, in any spanning tree of the graph describing the FSM, is $\mathcal{E}_M$. Assuming that each method to be tested m takes constant time, and checking for constraint validity takes constant time, **TestASequence** (Algorithm 3) takes time linear in the sequence S given to it, i.e. $\mathcal{O}(\mathcal{E}_M)$.

From the above, we see that the total number of sequences generated by Algorithms 1 and 2 for all the sequences is $\mathcal{O}(\mathcal{NAE})$. Moreover, $\mathcal{N}_M \leq \mathcal{N}$ and $\mathcal{E}_M \leq \mathcal{E}$. Therefore, Algorithm 4 has a complexity of $\mathcal{O}(\mathcal{NE} + \mathcal{N}^2 + \mathcal{N}^2\mathcal{AE} + \mathcal{NAE}^2) = \mathcal{O}(\mathcal{NAE}^2)$ as $\mathcal{N} \leq \mathcal{E}$.

In practice, the numbers and sizes of FSMs defining a single class are fairly small, as are the number of attributes in a class. Moreover, all the complexities stated above are really pathological worst cases which are unlikely to occur in practice. For instance, the unbounded stack example (Example 1) has $\mathcal{N} = 4, \mathcal{E} = 7, \mathcal{A} = 2$. But the number of sequences generated to achieve edge and dependence coverage is 8 as against the worst case figure of $4 \times 7 \times 2 = 56$.

Hence, we claim that the method presented in Algorithm 4 is a pragmatic and practical approach to testing the correctness of the object-state of a class *guaranteeing* edge and dependence coverages.

6. RELATED WORK

[15, 4, 11, 18, 16, 8, 12, 6, 3, 2, 5] is a representative list of previous research on specification-based testing. Of these [16, 8, 12, 6, 3, 2, 5] concentrate on specification-based testing of OO systems. As this paper focuses on testing of OO systems, we will discuss our work in relationship to them.

[16] presents an experience of using object-models for testing OO software, and the gains to be had by this approach. As it is more of an experience paper, it does not present any details of test-case generation or coverage issues. Moreover, it does not discuss the important area of automatic verification of test results.

Many of the other papers [8, 12, 6, 2, 5] work on an algebraic specification of the system to be tested, which provides a strong theoretical basis to discuss properties of testing. However, algebraic specification-based techniques have not scaled up to be used in real-life software applications. This has hindered the adoption of these techniques in the industry.

Doong and Frankl [8] present an approach called ASTOOT based on state equivalence. They generate pairs of method invocation sequences from the axioms that are part of the algebraic specifications. The two elements of a pair of sequences are semantically equivalent, i.e. their execution results in an 'observably equivalent states'. They assume that the set of axioms, when viewed as a term-rewrite system, have the Church-Rosser property and use re-writing as the means to generate the test-cases. The ASTOOT approach cleanly divides the problem of automated testing into two distinct parts, that of deriving test-cases and that of actually testing the software. This division of concerns enables a user to take advantage of automatic testing even if the specifications are not formal. This approach does not include testing that non-equivalent sequences of methods lead to non-equivalent states.

Chen et al discuss testing of individual classes in [6] and extend their technique to test clusters of classes in [5]. Their technique to test a single class is an extension of the ASTOOT work. [5] extends the technique to generate pairs of non-equivalent sequences of method invocations, to facilitate testing that non-equivalent sequences of methods do indeed lead to non-equivalent states. However, the number of such pairs is very large, and may not be practically feasible. [5] also presents techniques to test a *cluster* of classes, rather than a single class, using the notion of a class's *contract* or interface to define testing of a cluster. This is an interesting idea which needs further exploration.

Ball et al present an approach in [2] by which container classes can be tested for correctness. Their approach hinges on selecting a set of states to be tested, and for each selected state, 'taking' an object to the desired state through a series of method-invocations, and testing that the object behaves correctly in the selected state. The selection of the set of states, and deciding on the set of methods to take objects to the appropriate state is left to the test-designer. Testing is then done by checking individual methods in the selected state, and not a sequence of methods.

[3] presents an approach to testing classes based on specifications in Object-Z. Their approach is to automatically extract FSMs from the specification, convert these FSMs into 'test-graphs', which are then used to generate test-sequences using a testing framework called *ClassBench*. However, as with algebraic specifications, specifying a system in Object-Z is not yet a practical option in the industry. The method invocation sequences as generated by *ClassBench* achieves edge-coverage, but there is no equivalent of dependence-coverage in the *ClassBench* framework.

Our technique of testing classes has the following advantages over previous research discussed above.

- It is based on a specification technique that is popular in the developer community, namely, object-modelling.
- It formally defines coverage over the specification, and presents algorithms to generate sequences of method invocations which guarantee the coverage. Different criteria of coverage have been defined for structural (or white-box) testing. These include control-flow criteria such as branch-coverage [9], condition-coverage [14], path-coverage [13], and data-flow criteria such as all-uses criterion and k-tuples criterion [7]. No such criteria of coverage have been defined for specification-based testing.
- The definitions of coverage ensure that *all methods* of the class and *all pair-wise* interaction between methods are tested.
- In the worst case, the algorithms are polynomial in the size of the class specification, and therefore, practical to use in real life applications.

7. CONCLUSIONS

This paper presents a technique for automatically testing a class from its object-model specification. Object-modelling was the chosen specification mechanism because of its popularity in the software development community. We defined two kinds of coverage over the specification, edge-coverage and dependence-coverage, which guaranteed that all methods and pair-wise method interactions get tested.

Based on these notions of coverage, we presented algorithms to generate method sequences over the class which guarantee the coverage. The algorithms have a polynomial worst-case complexity. Therefore, we believe that our technique presents an efficient and pragmatic way to test a class, while guaranteeing coverage.

Testing a class is only the first step towards testing object-oriented systems. Different relationships may exist between classes, and different interactions may take place between classes that form part of a larger system. In addition, different *components*, each consisting of many classes, may interact with each other in a component-based system. This work needs to be extended to test the more complex system specifications mentioned above.

8. REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Mass., 1986.

[2] T. Ball, D. Hoffman, R. W. F. Ruskey, and L. White. State generation and automated class testing. *Software Testing Verification and Reliability*, 10(3):149 – 170, 2000.

[3] D. Carrington, I. MacColl, J. McDonald, J. Murray, and P. Strooper. From Object-Z specifications to ClassBench test suites. *Software Testing Verification and Reliability*, 10(2):111 – 137, 2000.

[4] J. Chang, D. Richardson, and S. Sankar. Structural specification based testing with ADL. In *ACM International Symposium on Software Testing and Analysis*, pages 62 – 70. ACM, 1996.

[5] H. Chen, T. Tse, and T. Chen. TACCLE: a methodology for object-oriented software Testing At the Class and Cluster LEvels. *ACM Transactions on Software Engineering and Methodology*, 10(4), Dec. 2000.

[6] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250 – 295, July 1998.

[7] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil. A formal evaluation of data flow path coverage criteria. In R. Poston, editor, *Automating specification-based software testing*, pages 155 – 195. IEEE computer society press, 1996.

[8] R. Doong and P. Frankl. The ASTOOT approach to testing OO programs. *ACM Transactions on Software Engineering and Methodology*, 3(4):101 – 130, Dec. 1994.

[9] N. Gupta, A. Mathur, and M. Soffa. A new approach to generate test data for branch coverage. In *ACM seventh international symposium on foundations of software engineering*. ACM, 1999.

[10] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch family of specification languages. *IEEE Software*, 2(5):24–36, Sept. 1985.

[11] W. Howden. The theory and practice of functional testing. In R. Poston, editor, *Automating specification-based software testing*, pages 49 – 60. IEEE Computer society press, 1996.

[12] M. Hughes and D. Stotts. Daistish: Systematic algebraic testing for OO programs in the presence of side-effects. In *ACM International Symposium on Software Testing and Analysis*, pages 53 – 61. ACM, 1996.

[13] B. Korel. Automated software test data generation. *IEEE transactions on software engineering*, 16(8):870 – 879, Aug. 1990.

[14] G. Myers. *The Art of Software Testing.* Wiley, New York, 1979.

[15] D. Peters and D. Parnas. Generating a test oracle from program documentation. In *ACM International Symposium on Software Testing and Analysis*, pages 58 – 65. ACM, 1994.

[16] R. Poston. Automated testing from object models. *Communications of the ACM*, 37(9):48 – 58, Sept. 1994.

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language User Guide.* Addison-Wesley, Massachusetts, 1999.

[18] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. In R. Poston, editor, *Automating specification-based software testing*, pages 75 – 85. IEEE Computer Society Press, 1996.

APPENDIX

A. PROOFS OF ALGORITHMS

A.1 Proof of Algorithm 1

We show that every edge $e \in M_i$ is part of some sequence $s \in S_E$ returned by **EdgeCovPathGen**. By Definition 2, the graph G is connected and therefore, every state belongs to both T and T^R.

Every edge e must be either in T or in E as computed by Step 2 of the algorithm.

Case $e \in E$**.** This means e is necessarily covered as a sequence is generated for all $e \in E$ in Step 6 of the algorithm.

Case $e \in T$. Consider a path $e_0, \cdots, e_k = e, \cdots, e_n$ from the initial state i_i to the final state f_i.

If there exists a path such that all of $e_k \cdots e_n \in T$, then the tree path from i_i to f_i includes e. Therefore, Step 4 of the algorithm generates a sequence for e.

Let all paths be such that at least one of $e_{k+1} \cdots e_n \in E$. Let j be the smallest number such that $j > k$ and $e_j \in E$. All edges $e_k \cdots e_{j-1} \in T$, and the tree path to source(e_j) includes e ($= e_k$). Therefore, the sequence generated for e_j by Step 6 of the algorithm includes e, because it includes the path from i_i to source(e_j).

A.2 Proof of Algorithm 2

By contradiction. Let there be a dependence pair $(d, u)_a$ that is not covered by the set of paths S_D computed by algorithm 2. This situation could arise if any of the following is true.

1. There is no sequence $s \in S_D$ containing transition d.
2. There is no sequence $s \in S_D$ containing transition u.
3. There is no sequence $s \in S_D$ with a definition-clear path for attribute a from target(d) to source(u).

If none of the above cases is true, it is easy to see that, for *every* dependence-pair $(d, u)_a$, there is a sequence $s \in S_D$ which contains a definition-clear path for attribute a from d to u, thus proving the correctness of Algorithm 2. We refute each of the above cases below.

1. d is a defining transition which has a corresponding use transition u for some attribute a, otherwise d is not of interest. In such a case, *DU-set* (Step 8) contains a triple $\langle d, a, DU_{(d,a)} \rangle$, where $DU_{(d,a)}$ contains the use u. It is easy to see that step 12 adds a path to S_D which contains d.
2. Similar to the above case.
3. For a definition and attribute combination (d, a), dfs'' $(G, target(d), a)$ visits *all* use nodes of the form source(u) or target(u) where a gets used. This is because, by Definition 5, there *must* exist a path in G from d to such u's without any intermediate definition of a. Therefore, the element inserted in Step 8 for the (d, a) combination, contains a tree $DU_{(d,a)}$ with a path from $target(d)$ to the corresponding use node. Therefore, step 12 guarantees that a path is inserted into S_D such that it contains a path from d to u along which no definitions of a occur.

A.3 Proof of Algorithm 3

We first show that, for each method invocation m_i in s, ϵ_i represents the corresponding system state. This can easily be shown by induction on i. ϵ_0 is the initial state (Step 2), thus proving the base case. ϵ_i is set from ϵ_{i-1} in Step 7, proving the inductive case. Similarly, ϵ_{i+1} represents the system state after method m_i. We now prove the four conditions to be satisfied by the algorithm.

1. Algorithm 3 checks to see if ϵ_i respects π_{m_i} (Step 4), else it returns failure. Therefore, it cannot return success if any of the π_{m_i} is not satisfied.
2. Algorithm 3 checks if ϵ_{i+1} satisfies ψ_{m_i} in Step 8, and returns failure if it does not succeed. Therefore, it cannot return success if any of the ψ_{m_i} is not satisfied.
3. In Step 11, Algorithm 3 returns failure if the state represented by ϵ_{n+1} does not respect ψ_M. Therefore, it does not return success when ψ_M is not satisfied.
4. Obvious from Steps 4, 8 and 11 of the algorithm.

From the above, it is clear that Algorithm 3 returns success iff all the conditions listed in Section 4 are met. If any of them fails, it returns failure.