A Framework for Inheritance Testing from VDM++ Specifications

Aamer Nadeem

Center for Software Dependability
Mohammad Ali Jinnah University
Islamabad, Pakistan
anadeem@jinnah.edu.pk

Michael R. Lyu

Department of Computer Science and Engineering
Chinese University of Hong Kong
Hong Kong S.A.R., China
lyu@cse.cuhk.edu.hk

Abstract

The benefits offered by the use of formal methods are not limited to avoidance of specification errors and elimination of ambiguities only – a formal specification also provides a sound basis for generating test suites. Inheritance is a powerful mechanism in object-oriented paradigm by which a subclass inherits data and functionality of a super class. Testing of inheritance relationships is crucial in object-oriented testing, as an inheritance error may lead to subtle bugs such as due to overridden functionality. In this paper, we introduce a technique to generate test cases for inheritance testing, using a VDM++ formal specification. The proposed technique is based on the flattening of a VDM++ specification class, and then generating operation sequences from the trace structure specified in the VDM++ specification. The input space for each operation is partitioned, and a test model is constructed from the operation sequences and the input partitions. Test paths are generated from the test model, which cover the different operation sequences as well as the partitions. We also define various coverage criteria for test path generation.

Keywords: Formal specification, Test-case generation, Specification-based testing.

1. Introduction

The use of formal methods in software development helps avoid specification errors and ambiguities in early phases of software life cycle. However, the use of formal methods does not guarantee correctness of an implementation, or its conformance to the specification [1]. While a formal specification can be used as a basis for a correctness proof, or a formal transformation, it becomes practically impossible to perform a proof of correctness as the size and complexity of the software grows. Even after a formal proof, testing is required to build confidence in the system being developed [2]. Besides, testing is considered as the most practical means of verification, despite its major limitation that it can only show presence of errors, and not their absence. Fortunately, several aspects of testing from formal specifications can be automated: several researchers have proposed techniques for automatic generation of test cases from formal specifications. However, most of the research in formal specification based testing has focused on unit level testing only [3]. In object-oriented paradigm, inheritance is a powerful mechanism by which a subclass inherits data and functionality of a super class. Testing of inheritance relationships is crucial in object-oriented testing, as an inheritance error may lead to subtle bugs such as due to overridden functionality. In this paper, we present a novel framework that automates generation of inheritance test cases from a VDM++ [4] specification. The proposed approach can be generalized to other object-oriented formal notations. The approach is based on flattening a derived class and then generating operation sequences for the flattened class, from the trace structure specified in the VDM++ specification. For each operation in operation sequence, we partition its input domains using its pre-condition and the class invariant. By combining the operation sequences with the input partitions, we construct a test model, which forms the basis of test case generation. Each test case is derived as a path through the test model. We also define various coverage criteria for generating test cases from the test model.

The rest of this paper is organized as follows: section 2 surveys the related work; section 3 describes the test generation framework in detail with a running example; section 4 presents coverage criteria for the testing technique; and finally section 5 concludes the work.

2. Related Work

A survey of the literature shows that a large amount of research work has been carried out to automate the generation of test cases using formal specifications. However, most of the research in formal specification based testing has focused on unit testing only – testing of inheritance, polymorphism, and object interactions using formal specification as a basis is still an open area of research.

Dick and Faivre [5] proposed a methodology to convert VDM-SL expressions into a disjunctive normal form (DNF), so that a solution to each disjunct represents a solution to the entire expression [5]. The state space represented by the DNF expression is then exhaustively searched using a Prolog tool to generate the test cases. Helke and Neustupny [6] describe the use of a theorem prover tool Isabelle to automate generation of test cases from Z specifications encoded in Isabelle/HOL. The tool converts Z predicates to DNF, eliminates unsatisfiable disjuncts, and generates valid test cases by searching the state space.

Meudec [2] proposes a method to generate test cases from VDM-SL specifications by converting the pre- and post-condition expressions into DNF, partitioning the DNF into equivalence classes and using boundary value analysis to generate test cases from the equivalence classes. The approach is based on parsing VDM-SL expressions, and is implemented by Atterer [7].

Stocks and Carrington [8][9] propose a test template framework which uses the Z notation to generate test templates. This work has been further extended by Carrington et. al. [10] for specification-based class testing. The authors have shown their proposed framework to be flexible by allowing the user to specify a test generation strategy. The main limitation of this work is that it is not fully automated.

The above work has also been further extended for object-oriented specifications by Liu et. al. [11]. It is based on Object-Z notation, and can be partially automated. The proposed framework in this work generates a valid input space (VIS) for class methods, and applies a strategy on VIS to generate test data. Valid sequences of execution of methods are determined by constructing a finite state machine (FSM) for the class under test.

Bernard et. al. present a case study on generating test sequences for Smart Card GSM 11-11 standard [12]. The test generation method used in the case study is based on the B notation, and is implemented in the B Testing Tools. Their approach is based on computation of all the boundary states for the B machine (a boundary state is defined as a state in which at least one state variable has the minimum or maximum value), and generating a test path for each boundary state. The test paths (called preambles) ensure that a boundary state is reached from the initial state. The operation to be tested is then invoked from each boundary state and the final state is examined. The authors have demonstrated that the test generation method gives a wide coverage (compared with manually generated tests) and saves 30% of test design time.

In Bernard et. al.'s work, the preamble is computed automatically using a best-first search algorithm on a constrained reachability graph. A major limitation of this approach is that it is based on the assumption of uniformity on the domain of the path. Another limitation is that only the first path discovered by the algorithm is used as preamble. As there can be multiple paths (possibly infinite) leading to a boundary state from the initial state, the single path coverage may not be adequate.

Boyapati et. al. [13] present a framework named Korat that uses Java Modeling Language (JML) predicates to generate the input space, and a Finitization class to bound the input state space. The bounded state space is searched and invalid objects, that do not satisfy a representation constraint, are discarded. The authors have implemented their framework, and have shown it to be efficient and effective, but its main limitation is that it is Java-specific.

3. Framework Architecture

The test generation framework uses a VDM++ specification for the class hierarchy, flattens the derived class (to be tested), and generates test paths from the flattened VDM++ specification class. This section briefly introduces the VDM++ notation and describes framework architecture in detail.

3.1. VDM++

VDM-SL [14] [15], one of the few formal languages whose syntax and semantics have been completely formally defined, is a model-based specification language based on denotational semantics. VDM++ [16] is an object-oriented extension of the VDM-SL. It supports various forms of abstraction, and step-wise refinement of abstract models into a concrete implementation. In VDM++, representational abstraction is supported by mathematical data structures, such as sets, sequences, maps, composite objects, Cartesian products and unions. At a lower level, the language provides various numeric types, the Boolean type, tokens and enumeration types. By using the data-structuring mechanism and the basic data types, compound data types can be formed with a specific mathematical structure. Subtyping is supported by attaching domain invariants to domain definitions.

Operational abstraction is supported in VDM++ by function specification, and the operation specification. Both functions and operations may be specified implicitly using pre and post conditions, or explicitly using applicative constructs to specify functions and imperative constructs to specify operations. Operations have direct access to a collection of global objects – the state of the specification. The state is constructed as a composite object, built from labeled components.

A VDM++ specification typically consists of a collection of classes. Each class has a state description, domain definitions, constant definitions, a collection of operations and a collection of functions. An initial specification should be as abstract as possible. Two techniques are available for the further development of the initial specification: data reification, which addresses the refinement of the state elements, and operation modeling, which addresses the refinement of the functions and operations. Data reification involves the transition from abstract to concrete data types, and a justification of this transition. Choosing a more concrete data model implies a redefinition of all operations and functions on the original model in terms of the new model, a process called operation modeling [17].

An architectural diagram of the test generation framework is given in Fig. 1. The following subsections describe the framework components in detail with a running example. At the end, we define a set of coverage criteria for generating the test cases.

3.2. Class Flattening

The idea of a *flattened class* is suggested in [18] for unit testing of a derived class. We apply this idea to the VDM++ specification. Flattening is the process of expanding an original class by including all the attributes and operations it inherits from the ancestor classes, except the overridden operations. The class invariant of the flattened class is a conjunction of the invariant of the original class and invariants of all ancestor classes in the inheritance hierarchy.

For instance, consider the inheritance hierarchy in the UML class diagram for a bank account class, in Fig. 2a. The parent class *Account* is an abstraction of a bank account with the basic attributes and operations common to all types of accounts. The derived classes *SavingsAccount* and *CheckingAccount* model two common types of bank accounts. Fig. 2b shows the flattened SavingsAccount class.

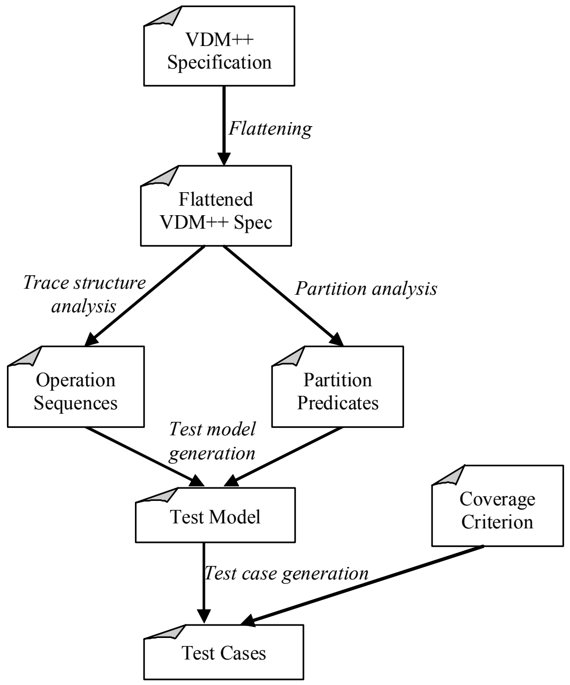

Fig. 1. Architecture of the proposed framework

Fig. 3 gives VDM++ specification of the flattened *SavingsAccount* class.

3.3. Generating Operation Sequences

The correct functionality of a class depends upon the sequence in which its operations are invoked. In VDM++, the set of all valid sequences of operations is specified as synchronization constraints in a class specification. The synchronization constraints are defined as trace structures, as shown in Fig. 3 for the *FlattenedSavingsAccount* class. In a trace structure, the semi-colon operator is used to indicate sequential execution of two operations (or two groups of operations represented by two sub-traces). The effective trace structure of a derived class is the weave of trace structures of the parent class and the child class [4]. In the example synchronization constraint of Fig. 3, the general trace structure T uses the sub-structures X and Y to specify the sequences of operations. If X and Y are replaced in the general trace structure, it becomes the expression,

open;((withdraw;deposit*;postInterest*; getBalance*)*;(deactivate;getBalance*; activate)*)*;(deactivate*;close)*

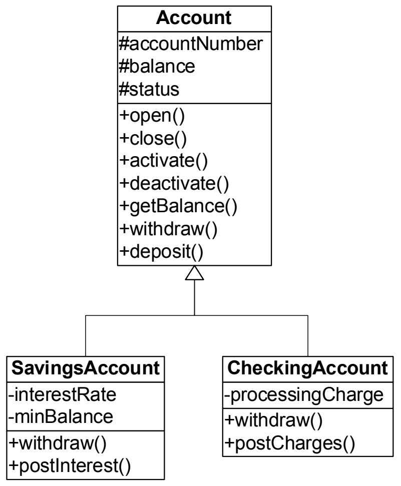

Fig. 2a. Class diagram for *Bank Account* hierarchy

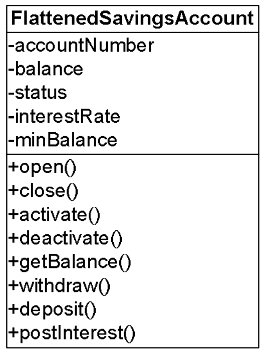

Fig. 2b. The *FlattenedSavingsAccount* class

This expression represents an infinite number of operation sequences, but for testing purposes we can place a limit on the number of sequences, for instance, by restricting the iterative operations to *n* iterations. Some valid operation sequences are:

open; close
open; deactivate; close
open; withdraw; close
open; deposit; close
open; postInterest; deactivate; close
open; getBalance; deactivate; getBalance; close
etc.

In Fig. 4, we present a recursive algorithm to generate a set of valid operation sequences for a given trace structure. Input to the algorithm is a trace structure expression, and the output is the corresponding set of operation sequences. The following is a brief description of the algorithm:

- if an empty expression ε is given as input, the output is the set containing an empty operation sequence.

```
class FlattenedSavingsAccount
instance variables
    accountNumber: nat;
    balance: real;
    status: <Active> | <Inactive> | <Closed>
    interestRate: real;
    minBalance: real;
invariant balance >= 100 and balance >= minBalance;
operations
    open(amount: real)
        ext wr balance: real;
            wr status: <Active> | <Inactive> | <Closed>;
        pre amount >= 100;
        post balance = amount and status = <Active>;
    close()
        ext wr status: <Active> | <Inactive> | <Closed>;
        post status = <Closed>;
    activate()
        ext wr status: <Active> | <Inactive> | <Closed>;
        post status = <Active>;
    deactivate()
        ext wr status: <Active> | <Inactive> | <Closed>;
        post status = <Inactive>;
    getBalance() bal: real
        ext rd balance: real;
        post bal = balance;
    deposit(amount: real)
        ext wr balance: real;
        pre amount > 0;
        post balance = balance~ + amount;
    withdraw(amount: real)
        ext wr balance: real;
        pre balance >= minBalance + amount;
        post balance = balance~ - amount;
    postInterest()
        ext wr balance: real;
        post balance = balance~ * (1+interestRate);
sync
   subtrace X = <(withdraw*;deposit*;postInterest*;getBalance*),
                 {withdraw, deposit, postInterest, getBalance}>;
   subtrace Y = <(deactivate;getBalance*;activate),
                 {deactivate, getBalance, activate}>;
   general    T = <open;(X*Y*)*;(deactivate*;close),
                 {withdraw, deposit, postInterest, getBalance,
                  deactivate, activate, open, close}>;
end FlattenedSavingsAccount
```

Fig. 3. VDM++ specification for the *FlattenedSavingsAccount* class

- if a single operation *op* is input, the output is the set containing *op* only, i.e., [*op*].
- if the input expression R is of the form R_1+, or R_1*, then the algorithm generates operation sequences for up to three iterations of R_1, *i.e.,* R_1, $R_1;R_1$, *and* $R_1;R_1;R_1$. The empty sequence ε is also generated in the case of * operator.

- if the input expression *R* is of the form $R_1;R_2$, where R_1 and R_2 are sub-expressions, then the output is the product of sets of operation sequences S_1 and S_2 generated from R_1 and R_2 respectively. The product of two sets of operation sequences S_1 and S_2 is defined as the set of all operation sequences formed by concatenating operation sequences of S_1 with operation sequences of S_2.
- if the input expression *R* is of the form R_1**S, where R_1 is a sub-expression and S is an alphabet set, the output sequences are generated from R_1 with only those operations specified in S.
- if the input expression *R* is of the form R_1 w_ R_2, operation sequences are generated from both R_1 and R_2, and then all possible combinations of the two sets are formed.

The operation sequences generated in this phase are input to the test model generator.

```
function genOpSeqs(R : RegExpr): set of OpSeq
{       OSset : set of OpSeq;
        OSset := [ ];
        if (R is ε) then OSset := [ε];
        else if (R is of the form op) then OSset := [ op ];
        else if (R is of the form $R_1$ ** S) then
                OSset := restrict(genOpSeqs($R_1$), S);
        else if (R is of the form $R_1$ w_ $R_2$) then
                OSset := weave(genOpSeqs($R_1$), genOpSeqs($R_2$));
        else if (R is of the form $R_1$ ; $R_2$) then
                OSset := product(genOpSeqs($R_1$), genOpSeqs($R_2$));
        else if (R is of the form $R_1$+) then
                OSset := union(genOpSeqs($R_1$),
                        genOpSeqs($R_1$ $R_1$), genOpSeqs($R_1$ $R_1$ $R_1$));
        else if (R is of the form $R_1$*) then
                OSset := union( [ε], genOpSeqs($R_1$),
                        genOpSeqs($R_1$ $R_1$), genOpSeqs($R_1$ $R_1$ $R_1$));
        return OSset;
}
```

Fig. 4. Algorithm for generating operation sequences

3.4. Partitioning the Input Space

Each operation in an operation sequence can be viewed as a transformation applied to the inputs to produce the outputs. The set of all valid inputs forms the input space. The inputs to an operation consist of two types of parameters: explicit parameters specified in the operation signature, and the implicit parameter *this,* i.e., the object state represented by the values of instance variables. The input space of an operation *m* of class *C* is defined by the set of all possible values of the instance variables of the class object and the explicit parameters of the operation. The *valid* input space, however, is the subset of input space which satisfies the predicate expression,

$$E \equiv inv(C) \wedge pre(m) \wedge type\text{-}constraints$$

where *inv(C)* is the class invariant, *pre(m)* is the pre-condition of the operation *m*, and *type-constraints* are the implicit constraints that arise from refinement of data types in formal specification to those in the programming language. For instance, if VDM++ *nat* type is refined to the C++ *int* type, a type constraint is required to ensure that the value is a positive integer.

The predicate expression *E* is a well-formed Boolean expression that consists of one or more *clauses* joined with the *logical connectives (not, and, or),* and the *constructors* (a type of VDM++ operators used to construct the expressions). Without loss of generality, we can assume that the predicate *prestate(m)* does not have any constructors. A *clause* is either a relational sub-expression, a set membership sub-expression, or a more complex sub-expression involving operators of the types: *combinators, applicators,* and *evaluators.*

The expression *E* above is converted to the canonical *Disjunctive Normal Form (DNF)* [5], as

$$E \equiv D_1 \vee D_2 \vee D_3 \vee \ldots\ldots\ldots \vee D_n$$

where each disjunct is a conjunction of the form,

$$D_i \equiv C_1 \wedge C_2 \wedge C_3 \wedge \ldots\ldots\ldots \wedge C_k$$

The canonical DNF form is unique for a Boolean expression [19]. Each disjunct in the above DNF represents an input sub-domain for the operation *m.* Each sub-domain is divided into partitions by using boundary value analysis, where each partition represents a set of input values (or a sub-domain) for the operation *m.*

As a concrete example, consider the *deposit* operation of the *FlattenedSavingsAccount* class. The conjunction of class invariant and operation pre-condition is,

$$E \equiv (balance >= 100) \wedge (balance >= minBalance) \wedge (amount > 0)$$

DNF for this expression has only one disjunct, i.e., the expression itself. Applying boundary value analysis to partition this predicate,

$$\begin{pmatrix} balance = 100 \\ balance > 100 \end{pmatrix} \times \begin{pmatrix} balance = minBalance \\ balance > minBalance \end{pmatrix} \times \begin{pmatrix} amount = 1 \\ amount > 1 \end{pmatrix}$$

This results in 8 partitions, represented by the predicates,

$balance=100 \wedge balance=minBalance \wedge amount=1$
$balance=100 \wedge balance=minBalance \wedge amount>1$
$balance=100 \wedge balance>minBalance \wedge amount=1$
$balance=100 \wedge balance>minBalance \wedge amount>1$
$balance>100 \wedge balance=minBalance \wedge amount=1$
$balance>100 \wedge balance=minBalance \wedge amount>1$
$balance>100 \wedge balance>minBalance \wedge amount=1$
$balance>100 \wedge balance>minBalance \wedge amount>1$

Each of these predicates represents a partition of the input space for the operation *deposit.* For a concrete value of *minBalance,* some of these predicates would become unsatisfiable, and have to be discarded. For instance, if *minBalance=200,* the first four predicates would be unsatisfiable. The unsatisfiable partition predicates are eliminated by evaluating each predicate for specific values of the variables chosen from their domains. Fig. 5 is a diagrammatic representation of the partitions for the *deposit* operation, where P_i denote the partition predicates.

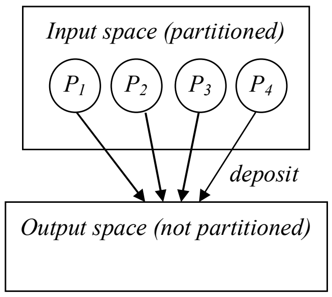

Fig. 5. Partitioning of input space for deposit operation

The partitioning of simple relational expressions, and the expressions involving finite sets, sequences, and maps can be automated. For instance, consider the following set membership expression with a universal quantifier,

forall x in set S & (x<y)

If S is a finite set of elements $s_1, s_2, s_3, \ldots\ldots, s_n$, then the above expression can be evaluated as,

$(s_1<y)$ *and* $(s_2<y)$ *and* $(s_3<y)$ *and* …… *and* $(s_n<y)$

Similarly, an expression with an existential quantifier can be evaluated as,

$(s_1<y)$ *or* $(s_2<y)$ *or* $(s_3<y)$ *or* …… *or* $(s_n<y)$

Expressions which invoke a VDM++ function can also be partitioned automatically, provided that they do not refer to an infinite collection. This limitation is often acceptable for test generation purposes since it is common to replace an unbounded set by a small finite set of enumerated values before testing commences [20].

3.5. Constructing Test Model

The test model is a non-linear structure that represents a collection of operation sequences which differ only by the input partitions of the operations involved. The operations and their order are the same in each operation sequence represented by the test model. The test model is constructed from an operation sequence and the partition predicates for each of the operations in the sequence. Consider, for example, the operation sequence,

open; deposit; withdraw; close

The *open* operation acts as a constructor, and has the pre-condition *amount>=0* which results in two partitions, i.e., *amount=0* and *amount>0.*

Input partitions for the *deposit* operation are given in sub-section 3.3. Similarly, for each of the *withdraw* and *close* operations, *four* and *two* input partitions are generated, respectively. For an operation *op,* let *op:pn* denote execution of the operation *op* for input values selected from the partition *pn,* then the test model for the above operation sequence can be represented as shown in Fig. 6.

$$\begin{pmatrix} open: p1 \\ open: p2 \end{pmatrix}; \begin{pmatrix} deposit: p1 \\ deposit: p2 \\ deposit: p3 \\ deposit: p4 \end{pmatrix}; \begin{pmatrix} withdraw: p1 \\ withdraw: p2 \\ withdraw: p3 \\ withdraw: p4 \end{pmatrix}; \begin{pmatrix} close: p1 \\ close: p2 \end{pmatrix}$$

Fig. 6. Test model for the operation sequence open; deposit; withdraw; close

3.6. Generating Test Paths

A test path is an operation sequence derived from the test model such that exactly one partition predicate is selected for each operation. Thus, the number of test paths that can be derived from a test model equals the product of the number of partition predicates for each operation in the operation sequence. For instance, the number of test paths for the test model of Fig. 6 is, *2x4x4x2 = 64*

The actual test paths are of the form:

open:p1 ; deposit:p1 ; withdraw:p1 ; close:p1
open:p1 ; deposit:p1 ; withdraw:p1 ; close:p2
open:p1 ; deposit:p1 ; withdraw:p2 ; close:p1
open:p1 ; deposit:p1 ; withdraw:p2 ; close:p2
open:p1 ; deposit:p1 ; withdraw:p3 ; close:p1
etc.

Each partition of each operation in the test model contains two types of variables, i.e. *state variables* (instance variables of the class), and *input parameters*. Test data values are required to be generated for the input parameters of all operations using boundary value analysis. However, the values of the state variables are generated only for the first operation in each test path. This is because the values of the state variables need to be set only once, i.e. before start of the test path execution. After that, each operation modifies the object state accordingly.

For each test path, the test values are chosen such that the test path can be exercised. Depending on the input values chosen, when an operation sequence is executed, exactly one of these paths is exercised. Since the number of test paths can be large for more complex predicates involving sets and sequences, we define various criteria on the test model to ensure that an effective set of test paths is selected at a minimal cost.

4. Coverage Criteria

Since the number of generated test paths can be extremely large, we define path coverage criteria for maximum desirable coverage while minimizing the cost.

Operation coverage: This criterion is defined as: *For each operation m in a class, there must be at least one test case t such that when the software is executed using t, the operation m is executed at least once.* This is the minimal coverage criterion, and can be satisfied without partitioning the inputs. It simply requires that each operation be executed with some valid inputs. This can be achieved without executing all operation sequences. For example, if a larger operation sequence contains all operations of the class, then only one test path would be sufficient to meet operation coverage.

Operation sequence coverage: Operation sequence coverage criterion is defined as: *For each operation sequence s derived from a trace structure, there must be at least one test case t such that when the software is executed using t, all operations in s are executed in the order specified by s.* This criterion requires that each operation sequence be executed at least once. In case of an iterative operation, each operation sequence contains a distinct number of iterations, thus the number of operation sequences can become quite large. In [21], it is suggested that the loops should be tested at the boundaries. For instance, if an iterative operation can execute a maximum of *n* times, then it should be tested by executing it *0, 1, n-1,* and *n* times. This strategy can significantly reduce the number of test paths for iterative operations.

Operation/Partition coverage: *This is defined as: For each operation m, and for each input partition p_i of m, there must be at least one test case t such that when the software is executed using t, the operation m is executed with some input values from p_i.* This criterion is similar to the operation coverage, but it also caters for each input partition of the operation. The number of test paths in this criterion, critically depends on the number of input partitions of the operations involved.

Operation sequence/Partition coverage: *This criterion is defined as: For each operation sequence s, and for each partition p_i of an operation m of sequence s, there must be at least one test case t such that when the software is executed using t, the operation sequence s is executed and the operation m is executed with some input values from partition p_i.* Again, this criterion is similar to the operation sequence coverage, but also requires coverage of the input partitions. The number of generated test paths depends not only on the number of distinct operation sequences, but also on the number of input partitions for each operation in a sequence.

All-path coverage: This is the most exhaustive coverage criterion. It requires execution of test paths with all possible combinations of input partitions involved in all operation sequences. However, depending upon the object states in a test path, some paths may be infeasible, and therefore, impossible to be exercised. The all-path coverage is satisfied if all feasible test paths are exercised.

Each of the above criteria requires that the data values be generated so that the coverage criterion is met. Also, implementation of this testing technique requires the code to be instrumented for setting and getting state variables and parameter values for each operation.

Fig. 7 shows subsumption relationships among the above-defined coverage criteria. It can be seen from the figure that the operation sequence/partition coverage subsumes all other criteria except the all-path coverage, but it produces a significantly less number of test paths compared with all-path coverage. To further illustrate this point, let us assume that the trace structure defines only one message sequence of *n* operations, and each operation has exactly k input partitions, then the number of test paths for all-path criterion would be k^n, whereas operation sequence/partition coverage yields only k paths for this sequence. In operation sequence coverage, the number of test paths is simply equal to the number of message sequences.

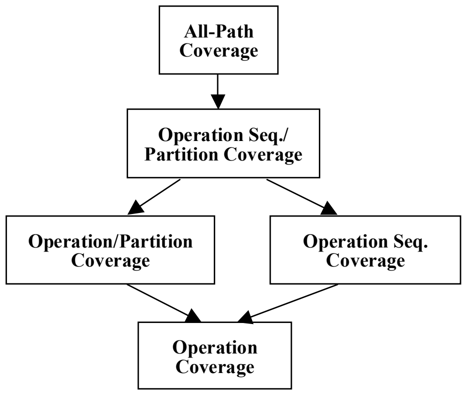
All-Path Coverage
Operation Seq./ Partition Coverage
Operation/Partition Coverage
Operation Seq. Coverage
Operation Coverage

Fig. 7. Subsumption relationships among coverage criteria

5. Conclusion

The contribution of this work is to combine the partition testing approach with testing of operation sequences, and apply the new approach to inheritance testing. A framework has been presented and demonstrated in the paper with a small case study. The main difficulties with the proposed approach are the same ones which arise in partition testing, i.e., the number of partitions can become large in cases where collections, such as sets and sequences, are involved, which results in an exponential number of test paths.

6. References

[1] Clarke, E.M., Wing, J.M., et. al., "Formal Methods: State of the Art and Future Directions", *ACM Computing Surveys*, Vol. 28, No. 4, December 1996.

[2] Meudec, C., "Automatic Generation of Software Test Cases From Formal Specifications"; *Ph.D. dissertation*, The Queen's University of Belfast, May 1998.

[3] Offutt, A.J., "Software Testing: From Theory to Practice", *IEEE AES Systems Magazine*, March 1998.

[4] *VDMTools: The VDM++ Language*, version 6.8.1, CSK Corporation, 2005, http://www.csk.co.jp/index_e.html.

[5] Dick, J., Faivre, A., "Automating the Generation and Sequencing of Test Cases from Model-based Specifications", *Proceedings of FME '93: Industrial-Strength Formal Methods*, Pages 268-284, Springer-Verlag, 1993.

[6] Helke, S., Neustupny, T., Santen, T., "Automating Test Case Generation from Z Specifications with Isabelle", *Proceedings of the 10th International Conference of Z Users*, 1997, Springer-Verlag.

[7] Atterer, R., "Automatic Test Data Generation from VDM-SL Specifications", *Diploma dissertation*, The Queens University of Belfast, April 2000.

[8] Carrington, D., Stocks, P., "A Tale of Two Paradigms: Formal Methods and Software Testing"; *ZUM '94, Z User Workshop*, Springer-Verlag, pp. 51-68, 1994.

[9] Stocks, P., Carrington, D., "A Framework for Specification-Based Testing", *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777-793, Nov. 1996.

[10] Carrington, D., MacColl, I., McDonald, J., Murray, L., Strooper, P., "From Object-Z Specifications to Classbench Test Suites", *Journal on Software Testing, Verification and Reliability*, Vol. 10, No. 2, pp. 111-137, 2000.

[11] Liu, L., Miao, H., Zhan, X., "A Framework for Specification-Based Class Testing", *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02)*, 2002.

[12] Bernard, E., Legeard, B., Luck, X., Peureux, F., "Generation of Test Sequences from Formal Specifications: GSM 11-11 Standard case study", *The Journal of Software Practice and Experience*, Wiley-InterScience, 2004.

[13] Boyapati, C., Khurshid, S., Marinov, D., "Korat: Automated Testing Based on Java Predicates"; *ACM ISSTA 2002.*

[14] Dawes, J., *The VDM-SL Reference Guide*, Pitman, London, 1991.

[15] Jones, C.B., *Systematic Software Development using VDM*, 2nd Edition, Series in Computer Science, Prentice-Hall, New Jersey, 1990.

[16] Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M., *Validated Designs for Object-oriented Systems,* Springer-Verlag, 2005, ISBN 1-85233-881-4.

[17] Plat, N., Katwijk, J.V., Toetenel, H., "Application and Benefits of Formal Methods in Software Development", *Software Engineering Journal*, September 1992.

[18] Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns and Tools,* Addison-Wesley Object Technology Series, 1999.

[19] Weyuker, E.J., Jeng, B., *Analyzing Partition Testing Strategies*, 2nd Edition, Van Nastrand Ranhald, 1990.

[20] Legeard, B., Peureux, F., Utting, M., "A Comparison of the BTT and TTF Test-Generation Methods", LNCS 2272, pp.309-329, Springer-Verlag, 2002.

[21] Beizer, B., *Software Testing Techniques*, 2nd Edition, Van Nastrand Ranhald, 1990.