An Automated Approach to Inheritance and Polymorphic Testing using a VDM++ Specification

Aamer Nadeem*, Zafar I. Malik*, Michael R. Lyu**

*Center for Software Dependability, Mohammad Ali Jinnah University, Islamabad, Pakistan
**Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong S.A.R., China
* {anadeem, zafar}@jinnah.edu.pk, **lyu@cse.cuhk.edu.hk

Abstract- **The use of formal methods is growing with the rapidly increasing applications of safety-critical systems in such fields as aviation, medicine, railways etc. The benefits of using 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. However, most of the work in this area has focused on unit testing only. In object-oriented paradigm, inheritance and polymorphism are powerful features, yet they present new challenges to the testers. In this paper, we present a novel approach to automated generation of test cases from a VDM++ specification. We base our testing technique on Offutt et al.'s fault model for subtype inheritance and polymorphic testing.**

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

I. INTRODUCTION

The past few years have witnessed a rapid growth in the application of safety-critical systems in such fields as aviation, medicine, and railways, which has also led to greater interest in the use of formal methods in software development. However, the use of formal methods, by itself, does not guarantee correctness of an implementation, or its conformance to the specification [1]. The widely recognized role of formal methods in verification is their use in correctness proofs. However, a formal proof of correctness is not feasible for most practical projects, because of the complexity involved. Even after a formal proof, testing is required to build confidence into the system being developed [7]. Fortunately, the existence of a formal specification provides an opportunity to automatically generate test cases. Several researchers have proposed techniques for automatic generation of test cases from formal specifications. However, most of the research in this area has focused on unit level testing only [11]. There is a lack of sufficient research in formal specification based testing of inheritance and polymorphic relationships.

In this paper, we present a novel approach to automate the generation of test cases from a VDM++ specification using Offutt et al.'s fault model for subtype inheritance and polymorphism [21]. However, the proposed approach can be generalized to other object-oriented formal notations as well. The approach is based on generating operation sequences for a class, from the trace structure specified in the VDM++ specification. For each operation in an operation sequence, we partition its input domains using its pre-condition and the class invariant. The test cases are then constructed by combining the operation sequences with the input partitions.

The rest of this paper is organized as follows: section 2 surveys the related work; section 3 gives a background of the VDM++ and the fault model described in [21]; section 4 presents the test generation framework in detail; and finally section 5 concludes the work.

II. 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 specifications is still an open area of research.

Dick and Faivre [4] 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 [4]. The state space represented by the DNF expression is then exhaustively searched using a Prolog tool to generate the test cases. In [12], the authors 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 [7] 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 in [8].

In [9] and [10], a test template framework has been proposed which uses the Z notation to generate test templates. This work has been further extended in [13] for specification-based class testing. The authors have shown their proposed

1-4244-0794-X/06/$20.00 ©2006 IEEE

framework to be flexible by allowing the user to specify a test generation strategy. However, their framework has not yet been fully implemented.

The above work has also been further extended for object-oriented specifications in [6]. It is based on Object-Z notation, and can be partially automated. The proposed framework in their 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 [5]. 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.

In [16], a framework, Korat, has been presented 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.

III. Background

A. Introduction to VDM++

VDM-SL [18] [19], 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++ [14] is an object-oriented extension of the ISO 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 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].

B. Fault Model

Offut et. al. [21] define nine types of faults due to subtype inheritance and polymorphism. However, their fault model is applicable at the code level. An implicit specification in VDM++ is at a higher level of abstraction and does not provide sufficient information to generate test cases for all nine fault types. Our analysis of Offutt et. al.'s fault model shows that only four of the nine fault types can be covered by the test cases generated from a VDM++ specification. These four fault types are described below.

State Definition Anomaly (SDA)

This type of fault can occur if:

- an inherited method m_1 of the superclass defines a state variable v, and
- a method m_2 of the subclass that overrides m_1, does not define the inherited state variable v consistently with the overridden method m_1, and
- an object o of the subclass is assigned to a variable s of superclass type, and method *s.m2* is invoked

State Defined Incorrectly (SDI)

This type of fault can occur if:

- an overriding method of the subclass incorrectly defines an inherited state variable, i.e. the computation performed by overriding method is not semantically equivalent to the overridden method, and
- an object o of the subclass is assigned to a variable s of superclass type, and method *s.m2* is invoked

Incomplete (failed) Construction (IC)

This type of fault can occur if:

- the constructor does not define (or incorrectly defines) a state variable *v*, and
- a method *m* of the class uses the undefined state variable *v*
- an object *o* of the class invokes method *o.m*

State Visibility Anomaly (SVA)

This type of fault can occur if:

- a state variable *v* in the superclass has private access specifier, and
- an overriding method *m* of a sub-subclass cannot define the inherited state variable of super-superclass due to private access specifier.

C. *Specifying Inheritance and Polymorphism in VDM++*

In VDM++, the top-level system specification consists of a collection of related classes. The body of each class contains the following optional elements,

- type definitions
- value definitions
- instance variable definitions
- operation definitions
- function definitions
- synchronization definitions
- thread definitions

For each class, the header specifies class name and optional super class name(s) using *is subclass of* clause with the class name. For example,

```
class A is subclass of B
        ...
        ...
end A
```

When a class is defined as a subclass of an already existing class, the subclass definition introduces an extended class, which is composed of the definitions of the superclass, and the definitions of the newly defined subclass. The interface to the objects of the subclass is the same as the interface to its superclass extended with the new definitions within the subclass. A subclass inherits from the superclass all of its value and type definitions, instance variables, operation and function definitions, and synchronization definitions.

A class may inherit from more than one superclasses. However, a *name conflict* occurs when two constructs with the same name and of the same kind are inherited from different superclasses. Name conflicts must be explicitly resolved through name qualification, i.e. prefixing the construct with the name of the superclass and a `-sign (back-quote).

Polymorphic behavior cannot be explicitly specified in VDM++. However, a variable *v* of the superclass can be assigned an object of a subclass which allows overriding methods of the subclass to be invoked through *v*. Moreover, in a superclass, it is possible to delegate the responsibility to define an operation to the subclass(es) by using the *is subclass responsibility* clause, e.g.,

```
class A
        ...
        operations
        op1() is subclass responsibility
end A
```

The operation `op1()` is defined by a subclass B derived from `class A`. A superclass containing one or more abstract operations acts as an *abstract base class*.

IV. Framework Architecture

An architectural diagram of the test generation framework is given in Fig. 1.

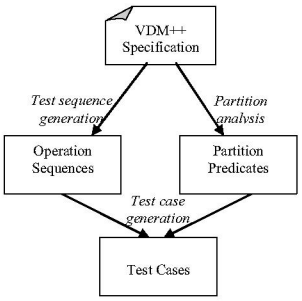

Fig. 1. Architecture of the proposed framework

The following is a brief description of the framework modules, and their functionality:

1. *Test sequence generator* constructs valid sequences of the operations of a class from the *trace structure* specified in VDM++ specification of the class. The trace structure is defined in the synchronization constraints section of a VDM++ class, and defines valid operation sequences in a notation based on regular expressions.
2. *Partition analyzer* combines class invariant predicate with pre-condition predicate of each method in the class, and applies partition analysis strategy on this predicate to construct partition predicates representing equivalence classes of valid inputs. This is done to generate test data for the operations in an operation sequence.
3. *Test case generator* constructs test cases from the test sequences. Each test sequence is a sequence of operation invocations on an object, where each operation invocation is a method call. The test generator sets values of the parameters in method calls, using the partition predicates.

The following subsections describe the framework components in greater detail with a running example.

A. 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 in *synchronization constraints* in a class specification. The synchronization constraints are usually defined as trace structures. A *trace structure* defines valid sequences of method invocations of a class for a particular object of the class.

Trace structures are specified using a language based on the notation of regular expressions, together with special trace structure operators, i.e.,

(i) *; (semi-colon)* denotes sequential execution. This is used to enforce the order of execution of operations or operation traces.

(ii) * *(asterisk)* denotes zero or more times repetition of an operation or an operation trace.

(iii) + *(plus sign)* denotes one or more times repetition of an operation or an operation trace.

(iv) ** *(double asterisk)* denotes the projection operator, and is used to restrict a trace of operations to a subset of the operation alphabet.

(v) *w_ (w underscore)* denotes the weave operator, and is used to perform synchronized interleaving of two operation traces

An implementation of a class with a trace structure specification is correct only if it guarantees that only the specified sequence of invocations can occur. The trace synchronization defines one general trace structure and an arbitrary number of subtrace structures. This scheme allows decomposition of the behavior of an object, in which the general trace structure is built from the subtrace structures.

In Fig. 2, we present an 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 explanation of the algorithm:

- if an empty expression ε is given as input, the output is the set containing an empty operation sequence.
- if the input expression consists of a single operation *op,* the output is the set containing *op* only, i.e., [*op*].
- if the input expression R is of the form R_1+, then the number of operation sequences formed would be infinite. However, the algorithm generates only a finite number of sequences for up to three iterations of R_1, *i.e.,* R_1, $R_1;R_1$, *and* $R_1;R_1;R_1$.
- if the input expression R is of the form R_1*, then the number of operation sequences formed would be infinite. However, the algorithm generates only a finite number of sequences for up to three iterations of R_1, *i.e.,* ε, R_1, $R_1;R_1$, *and* $R_1;R_1;R_1$.
- 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 R1 and R2, and then all possible combinations of the two sets are formed.

```
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 R1 ** S) then
                OSset := restrict(genOpSeqs(R1), S);
        else if (R is of the form R1 w_ R2) then
                OSset := weave(genOpSeqs(R1), genOpSeqs(R2));
        else if (R is of the form R1 ; R2) then
                OSset := product(genOpSeqs(R1), genOpSeqs(R2));
        else if (R is of the form R1+) then
                OSset := union(genOpSeqs(R1),
                        genOpSeqs(R1 R1), genOpSeqs(R1 R1 R1));
        else if (R is of the form R1*) then
                OSset := union( [ε], genOpSeqs(R1),
                        genOpSeqs(R1 R1), genOpSeqs(R1 R1 R1));
        return OSset;
}
```

Fig. 2. Algorithm for generating operation sequences

As an example, consider the inheritance hierarchy in the UML class diagram for a bank account class, in Fig. 3. 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.

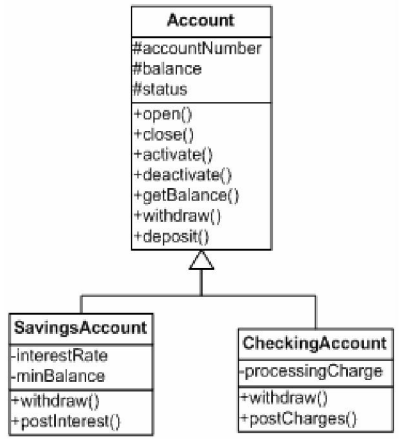

Fig. 3. Class diagram for Bank Account hierarchy

Fig. 4a and Fig. 4b present VDM++ specification for the *Account* and *SavingsAccount* classes, respectively. In an inheritance hierarchy, if synchronization constraints are specified as trace structure expressions in both the subclass and the superclass, the effective trace structure for subclass is obtained by weaving of the two trace structures [14]. The semi-colon operator is used to indicate sequential execution of two operations (or two groups of operations represented by two sub-expressions).

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

Fig. 4a. VDM++ specification for the Account class

In the example synchronization constraint in Fig. 4a, 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*; getBalance*)* ;*
(deactivate; getBalance; activate)*)*; (deactivate*; close)*

The above 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; withdraw; deactivate; close
open; getBalance; deactivate; getBalance; close
etc.

```
class SavingsAccount
instance variables
    interestRate: real;
    minBalance: real;
invariant balance >= minBalance;
operations
    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);
end SavingsAccount
```

Fig. 4b. VDM++ specification for the SavingsAccount class

The trace structure analyzer component of our framework generates all possible operation sequences from the specified trace structure expression, while repeating repetitive operation up to three times. The generation test sequences are saved in a file, and the test case generator selects only those test sequences which are required to be tested for the fault model.

In the Bank Account example, all traces involving *withdraw* operation of the *SavingsAccount* class need to be tested for *SDA* and *SDI* faults, since *withdraw* is an overriding method of the subclass that defines the inherited state variable *balance*.

For *IC* fault, all operation sequences involving object creation and an operation that reads a state variable need to be tested. For the *SVA* fault, the operation sequences that involve an operation of the sub-subclass that reads (or writes) a state variable of the super-superclass are tested.

B. Partition Analysis

In order to generate test data for each operation in the operation sequences to be tested, we use the well-known black box strategy of partition analysis. 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, or 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)* [4], 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,

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

Here k is the number of clauses in the expression E and $n = 2^k$ is the number of disjuncts. The canonical DNF form is unique for a Boolean expression [3]. 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 *Account* class. The conjunction of class invariant and operation pre-condition is, $(balance>=100) \wedge (amount>0)$. DNF for this expression has only one disjunct, i.e., the expression itself. Applying boundary value analysis to partition this predicate,

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

This results in 4 partitions, represented by the predicates,

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

Each of these predicates represents a partition of the input space for the operation *deposit.* The unsatisfiable partition predicates are eliminated by evaluating each predicate for specific values of the variables chosen from their domains. The partitioning of simple relational expressions and the expressions involving finite sets, sequences, and maps can be automated. For instance, in the 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 expression can be evaluated as, $(s_1<y) \wedge (s_2<y) \wedge (s_3<y) \wedge \ldots\ldots \wedge (s_n<y)$. Similarly, an expression with an existential quantifier can be evaluated as, $(s_1<y) \vee (s_2<y) \vee (s_3<y) \vee \ldots\ldots \vee (s_n<y)$

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].

C. Test Case Generation

The test case generator selects those test sequences which are required to be tested for the fault model. In the Bank Account example, the *withdraw* operation of the base class defines the state variable *balance*, and this variable is also defined by the overriding *withdraw* operation of the subclass. Thus all operation sequences involving the *withdraw* operation are selected to test the *SDA* and *SDI* faults. A *SavingsAccount* object *s* is created and assigned to a variable *a* of the *Account* class. The selected test sequences are required to be executed on variable *a*.

The *IC* fault requires testing of all operation sequences involving object creation and an operation that reads a state variable. The *SVA* fault cannot occur in our example, since there is no sub-subclass relationship in our model.

V. 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 and polymorphic testing using the fault model given by Offutt et al. We have presented a framework to demonstrate our approach on a running example. The main difficulty that we experienced in applying Offutt et al.'s fault model to specification based testing was that the specifications are at a higher level of abstraction and as such they cannot be used to generate test cases to reveal all types of faults in the fault model. The major advantages of our testing approach are that it is highly automatable, and it can be applied earlier in the software life cycle since it is based on the specifications rather than the code.

References

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

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

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

[4] J. Dick, A. Faivre, "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.
[5] E. Bernard, B. Legeard, X. Luck, F. Peureux, "Generation of Test Sequences from Formal Specifications: GSM 11-11 Standard case study", *The Journal of Software Practice and Experience*, Wiley-InterScience, 2004.
[6] L. Liu, H. Miao, X. Zhan, "A Framework for Specification-Based Class Testing", *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'02)*, 2002.
[7] C. Meudec, "Automatic Generation of Software Test Cases From Formal Specifications"; *Ph.D. thesis*, The Queen's University of Belfast, May 1998.
[8] R. Atterer, "Automatic Test Data Generation from VDM-SL Specifications", *Diploma thesis*, The Queens University of Belfast, April 2000.
[9] D. Carrington, P. Stocks, "A Tale of Two Paradigms: Formal Methods and Software Testing"; *ZUM '94, Z User Workshop*, Springer-Verlag, pp. 51-68, 1994.
[10] P. Stocks, D. Carrington, "A Framework for Specification-Based Testing", *IEEE Transactions on Software Engineering*, vol. 22, no. 11, pp. 777-793, Nov. 1996.
[11] A.J. Offutt, "Software Testing: From Theory to Practice", *IEEE AES Systems Magazine*, March 1998.
[12] S. Helke, T. Neustupny, T. Santen, "Automating Test Case Generation from Z Specifications with Isabelle", *Proceedings of the 10th International Conference of Z Users*, 1997, Springer-Verlag.
[13] D. Carrington, I. MacColl, J. McDonald, L. Murray, P. Strooper, "From Object-Z Specifications to Classbench Test Suites", *Journal on Software Testing, Verification and Reliability*, Vol. 10, No. 2, pp. 111-137, 2000.
[14] *VDMTools: The VDM++ Language*, version 6.8.1, CSK Corporation, 2005, http://www.csk.co.jp/index_e.html.
[15] R.V. Binder, "Testing Object-Oriented Systems: Models, Patterns and Tools"; Addison-Wesley Object Technology Series, 1999.
[16] C. Boyapati, S. Khurshid, D. Marinov, "Korat: Automated Testing Based on Java Predicates"; ACM ISSTA 2002.
[17] N. Plat, J.V. Katwijk, H. Toetenel, "Application and Benefits of Formal Methods in Software Development", *Software Engineering Journal*, September 1992.
[18] J. Dawes, *The VDM-SL Reference Guide*, Pitman, London, 1991.
[19] C.B. Jones, *Systematic Software Development using VDM*, 2nd Edition, Series in Computer Science, Prentice-Hall, New Jersey, 1990.
[20] B. Legeard, F. Peureux, M. Utting, "A Comparison of the BTT and TTF Test-Generation Methods", LNCS 2272, pp.309-329, Springer-Verlag, 2002.
[21] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, C. Hutchinson, "A Fault Model for Subtype Inheritance and Polymorphism", The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE '01), pages 84-95, Hong Kong, PRC, November 2001.