Automated Test Case Generation from IFAD VDM++ Specifications

AAMER NADEEM, MUHAMMAD JAFFAR-UR-REHMAN
Center for Software Dependability
Mohammad Ali Jinnah University
74-E, Blue Area, Islamabad
PAKISTAN

Abstract: - Most of the current research on automatic generation of test cases from formal specifications has been directed towards non object-oriented formal specifications. While object-oriented paradigm is the most widely accepted methodology for software development, generation of test cases from object-oriented formal specifications is still a relatively unexplored area. In this paper, we present a novel framework to automate test case generation from object-oriented formal specifications. We use IFAD VDM++ as the specification language, but the ideas presented can be generalized to other model-based object-oriented formal notations as well. The proposed approach uses a test descriptor to generate valid test sequences, and then generates test data for each method in a test sequence, using a conjunction of method precondition and class invariant to filter the input space. The test data are generated by partitioning the valid input space for each input variable into equivalence classes, and selecting representative values from each class.

Key-Words: - Formal methods, Object-oriented specification, Software testing, Specification-based testing, Automated testing

1 Introduction

Use of formal methods in early phases of software life cycle can help avoid specification errors and ambiguities. Unlike a natural language specification for the system, the precision of a formal specification eliminates ambiguities and misinterpretations. Another advantage of using a formal specification is that aspects of the specification can be rigorously demonstrated using mathematical proofs. However, use of formal methods does not guarantee that implementation will conform to specifications. A formal proof of correctness is not justifiable for most software projects because of the cost involved. Even after a formal proof, testing is usually required to build confidence in the system being developed [10]. Therefore, need for rigorous testing is not eliminated by the use of formal methods. In fact, formal methods and testing complement each other.

However, even for the most trivial systems, exhaustive testing is impossible due to the explosive size of input space, which makes it necessary to find ways to identify a representative set of test cases. For large and complex software systems, manually generating such a set of test cases, executing them, and comparing the results with expected outputs can be a tedious and time-consuming process. Fortunately, testing from formal specifications can be automated: several researchers have proposed techniques for automatic generation of test cases from formal specifications [10] [3] [13] [1] [5].

Over the past decade, object-oriented paradigm has emerged as a promising methodology for software development. It has provided several desirable characteristics, such as abstraction and encapsulation. Abstraction allows modeling of real-world objects and their properties. The physical properties of objects can be captured as attributes and their interactions as routines that may be performed on the attributes. Encapsulation restricts the users to only using the defined routines to access the attributes of the object, thus reducing the likelihood of inappropriate usage of the object.

During past fifteen years, considerable amount of research effort has been directed towards object-oriented analysis and design methodologies for developing software systems. However, testing of object-oriented systems has received relatively much less attention. In particular, little work has been done in the area of developing software testing techniques using object-oriented formal specifications. Though it is theoretically possible to continue using the traditional testing strategies for object-oriented systems, exploiting the object-orientation features of the software can help develop better testing methods. Thus, there is a need to investigate methods to automate testing from object-oriented formal specifications.

In this paper, we present a technique to automate generation of test cases from IFAD VDM++ [7] specifications. Our proposed approach can be generalized to other object-oriented formal notations as well.

The proposed technique requires a VDM++ specification, and a test descriptor, to generate the test cases. The test descriptor defines valid test sequences in an intermediate specification language based on regular expressions.

The rest of this paper is organized as follows: section 2 surveys related work by other researchers; section 3 describes our proposed technique in detail; and finally section 4 concludes the work.

2 Related Work

Various techniques have been proposed to automatically generate test cases from formal specifications. However, most of the research has been directed towards testing from non object-oriented formal specifications.

In [3], a methodology is proposed to convert VDM-SL pre condition expressions into a disjunctive normal form (DNF), so that a solution to each disjunct represents a solution to the entire expression. The state space generated by pre conditions is exhaustively searched using Prolog to generate test cases. In [5], 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.

In [10], a method is proposed 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 [1].

[6] gives an overview of testing based on Z, and proposes transforming Z operation schemas to DNF to generate test cases, but emphasizes the need to automate test evaluation because of the vast amount of data to be processed. [11] describes a test evaluation tool to support automatic test evaluation, by transforming schema predicates into executable forms which are compiled to boolean-valued C functions.

All of the above-mentioned works focus on testing from non object-oriented formal specifications. Testing from object-oriented formal specifications is still a largely unexplored area. [9] have proposed a framework to automate class testing from Object Z specification. Their work is based on generating a valid input space (VIS) for class methods, and applying 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. However, the framework proposed in [9] has not been implemented.

Another notable work among test generation frameworks for object-oriented formal specifications is the one presented in [2]. They present a framework, Korat, that uses Java Modeling Language (JML) predicates to generate 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 repOk method, are discarded. The repOK() method returns true if an object of the class under test is correctly represented, otherwise it returns false. The authors have implemented their proposed framework, and have shown it to be efficient and effective, but its main limitation is that it is Java-specific.

3 Architecture of the Test Case Generator

The Test Case Generator (TCG) consists of two main components (Fig. 1), i.e.

a. predicate parser, and
b. test generator

It requires a VDM++ specification, and a test descriptor. The predicate parser generates C code for method predicates formed by conjunction of method precondition and class invariant, which is used to filter the test data. The parser also creates a symbol table for the method predicates, which records variable names and their boundary values. The symbol table is used by test generator to generate test data.

The test generator also generates empty test shells from the test descriptor, which are then filled with test data. The test descriptor is an XML file that contains valid sequences of operations defined as regular expressions. The test data are generated from the symbol table by partitioning the input space into equivalence classes, and filtered by method precondition and class invariant.

In the following subsections, we describe functionality of each component of the TCG.

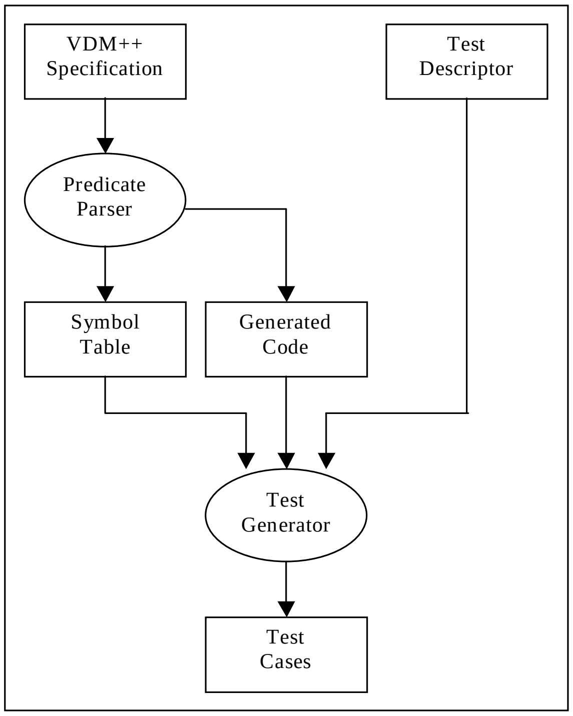

Fig. 1. Architecture of the Test Case Generator (TCG)

3.1 Parser

The specification of an object class in VDM++ contains a class invariant predicate, and method pre and post conditions for each method in the class. The parser constructs a *method predicate* for each method by forming a conjunction of class invariant and method precondition predicates. The method predicate is parsed into a parse tree using a context-free grammar (CFG). A context-free grammar for a simple IFAD VDM++ expression is given in Fig. 2 below.

Nonterminal symbols: Expr, Simple-Expr, Quant-Expr, Implies-Expr, Q-Opr, Or-Expr, And-Expr, Not-Expr, Clause, Rel-Expr, Num-Expr, Set-Expr, Set-Mem, Num, Rel-Opr, Set, Set-Opr, Set-Literal, Elem, Elem-List, Set-Imp

Terminal symbols: ⇔, •, ∀, ∃, !, ⇒, ∨, ∧, ¬, (,), Var, Literal, =, ≠, <, ≤, >, ≥, Set-Var, ⊂, ⊆, ⊃, ⊇, =, ≠, ∈, ∉, ∅, {, }, “,”, “|”

Start symbol: Expr

Expr	→ Simple-Expr | Quant-Expr
Simple-Expr	→ Implies-Expr | Implies-Expr ⇔ Simple-Expr
Quant-Expr	→ Q-Opr Expr • Expr
Q-Opr	→ ∀ | ∃ | ∃!
Implies-Expr	→ Or-Expr | Or-Expr ⇒ Implies-Expr
Or-Expr	→ And-Expr | And-Expr ∨ Or-Expr
And-Expr	→ Not-Expr | Not-Expr ∧ And-Expr
Not-Expr	→ Clause | ¬ Not-Expr
Clause	→ (Expr) | Rel-Expr
Rel-Expr	→ Num-Expr | Set-Expr | Set-Mem
Num-Expr	→ Num Rel-Opr Num
Num	→ Var | Literal
Rel-Opr	→ = | ≠ | < | ≤ | > | ≥
Set-Expr	→ Set Set-Opr Set
Set	→ Set-Var | Set-Literal
Set-Opr	→ ⊂ | ⊆ | ⊃ | ⊇ | = | ≠
Set-Mem	→ Elem ∈ Set | Elem ∉ Set
Elem	→ Var | Literal
Set-Literal	→ ∅ | {} | { Elem-List } | Set-Imp
Elem-List	→ Elem | Elem, Elem-List
Set-Imp	→ { Expr “|” Set-Mem • Expr }

Fig. 2. Context-free grammar for a simple VDM++ expression

From the parse tree, the parser generates C language code to evaluate the method predicate. The idea of converting a formal specification into a parse tree and generating C language code from the tree has been discussed in [12]. The parser produces a boolean-valued C function named *classname_methodname_*pre() for each method in the class under test (CUT). This generated function is used by test generator to filter the test data.

The parser also generates a symbol table containing all input variables for the method, and their boundary values, using the parse tree. As an example, consider the following VDM++ (partial) specification of a *BankAccount* class:

```
class BankAccount

   instance variables
      accountNum : int;
      balance : int;
      inv (accountNum > 0) ∧ (balance > 0);

   operations

      Create : (int) ==> ()
      Create (amount) == (balance := amount)
      pre   amount > 0;
      post   balance = amount;

      Withdraw : (int) ==> ()
      Withdraw(amount) == (balance := balance~ - amount)
      pre  (amount > 0) ∧ (amount < balance);
      post   balance~ = balance + amount;

end BankAccount
```

The input space for *Withdraw()* method consists of the implicit parameter, i.e. current *BankAccount* object, and the *amount* parameter. The method predicate for the *Withdraw()* method is formed by conjunction of the class invariant and the method precondition, as below:

$$((accountNum > 0) \wedge (balance > 0)) \wedge$$
$$((amount > 0) \wedge (amount < balance))$$

The parser uses the context-free grammar defined above to parse the method predicate, and generate C language code. The parser also produces symbol tables for class invariant and method precondition as shown in Fig. 3 and Fig. 4 respectively.

If an object is used as an attribute, or a parameter, then a separate symbol will be required to keep boundary values of its attributes.

The symbol tables define boundary values for each input variable of the method to be tested. These boundary values are used by test generator to partition the input space.

Class Name	Instance Variable	Type	Boundary Value
BankAccount	*accountNum*	*int*	*0*
BankAccount	*balance*	*int*	*0*

Fig. 3. Symbol table for BankAccount class

Class Name	Method name	Parameter	Type	Boundary Value
BankAccount	*Withdraw*	*amount*	*int*	*0*
BankAccount	*Withdraw*	*amount*	*int*	*balance*

Fig. 4. Symbol table for Withdraw method

3.2 Test Generator

The test generator component of TCG is further composed of three parts (Fig. 5), i.e.

a) test shell generator
b) test data generator, and
c) test case generator

3.2.1 Test Shell Generator

The test descriptor contains valid sequences of method calls in an intermediate specification language that extends the notation of regular expressions. This intermediate language has been described in [4] and is based on the work of [8]. The test shell generator determines valid test sequences from this test descriptor and forms templates for method calls. A *test template* consists of a *method name*, and its *parameter types*. A *test shell* is a *sequence of test templates* that describes a valid transaction.

For instance, if there are *Create, Delete, Withdraw, Deposit,* and *InquireBalance* methods in our *BankAccount* class, then the following could be valid sequences of operations:

Seq 1:	*Create, InquireBalance*
Seq 2:	*InquireBalance, Withdraw*
Seq 3:	*Withdraw, InquireBalance, Deposit*
Seq 4:	*Create, Deposit, Withdraw, Delete*

etc.

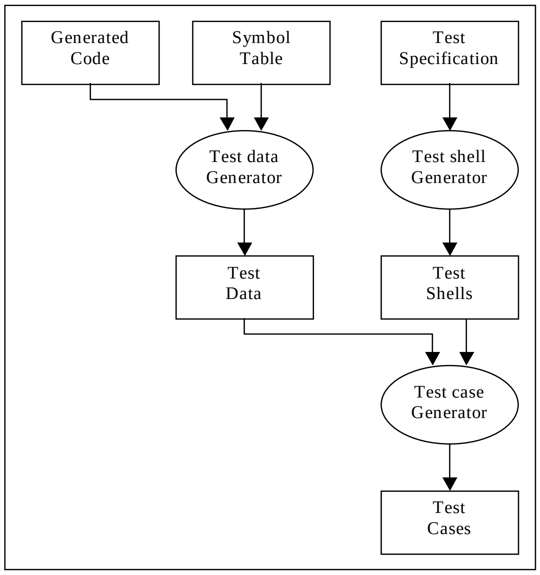

Fig. 5. Architecture of the Test Generator component of TCG

while the following sequences are invalid:

Seq 5: *Create, Delete, InquireBalance*
Seq 6: *Delete, InquireBalance, Withdraw*
Seq 7: *Withdraw, InquireBalance, Delete, Deposit*
etc.

In order to allow only valid sequences of operations, the test descriptor defines a test specification as shown in the example below.

SeqSpec(BankAccount) ⇒ Create · ProcessAccount · Delete
ProcessAccount ⇒ (Deposit , Withdraw) ↔ (InquireBalance)*

The first production states that an account must be first created then processed and finally deleted. The second production defines how an account can be processed. Using the above specification, the test shell generator produces test templates as shown below.

```
BEGIN TEST 1
  Create <>
  Withdraw <int>
  Delete <>
END TEST 1

BEGIN TEST 2
  Create <>
  Deposit <int>
  Withdraw <int>
  Delete <>
END TEST 2

BEGIN TEST 3
  Create <>
  Deposit <int>
  InquireBalance <>
  Deposit <int>
  Delete <>
END TEST 3
```

3.2.2 Test Data Generator

It determines *method inputs* for each method in the CUT. Method inputs consist of parameters of the method, including the implicit *this* parameter. Using boundary values from the symbol table, input space for each parameter is partitioned into equivalence classes. The test data generator generates a random test value for each parameter from each partition. Then, method predicate code is executed to filter the test data – only those data sets are passed to the test case generator which satisfy the method predicate.

In the BankAccount example, a possible set of test values generated by test data generator is shown in Fig. 6 and Fig. 7.

Class Name	Instance Variable	Type	Boundary Value	Test Val 1	Test Val 2	Test Val 3
BankAccount	*accountNum*	*int*	*0*	*-4*	*0*	*7*
BankAccount	*balance*	*int*	*0*	*-5*	*0*	*6*

Fig. 6. Test values generated for instance variables

Class Name	Method name	Parameter	Type	Boundary Value	Test Val 1	Test Val 2	Test Val 3
BankAccount	*Withdraw*	*amount*	*int*	*0*	*-3*	*0*	*2*
BankAccount	*Withdraw*	*amount*	*int*	*-5*	*-10*	*-5*	*-1*
BankAccount	*Withdraw*	*amount*	*int*	*0*	*-2*	*0*	*3*
BankAccount	*Withdraw*	*amount*	*int*	*6*	*4*	*6*	*12*

Fig. 7. Test values generated for method parameters

Using the generated test values, the test data generator forms data sets by constructing all possible combinations of test values of variables. In our example of *Withdraw* method, a total of 108 (=3x3x12) data sets will be generated. The table in Fig. 8 shows first fifteen data sets as an example.

Data set #	accountNum	balance	amount
1	*-4*	*-5*	*-3*
2	*-4*	*-5*	*0*
3	*-4*	*-5*	*2*
4	*-4*	*-5*	*-10*
5	*-4*	*-5*	*-5*
6	*-4*	*-5*	*-1*
7	*-4*	*-5*	*-2*
8	*-4*	*-5*	*0*
9	*-4*	*-5*	*3*
10	*-4*	*-5*	*4*
11	*-4*	*-5*	*6*
12	*-4*	*-5*	*12*
13	*-4*	*0*	*-3*
14	*-4*	*0*	*0*
15	*-4*	*0*	*2*

Fig. 8. First fifteen data sets produced by test data generator

Out of these 108 data sets only 3 (*data set # 99, 105 and 106*) satisfy the method predicate. This is determined by test data generator by executing the generated data sets on the C code for the method predicate. The negative data sets (which do not satisfy method predicate) are eliminated, and only positive ones are passed to the test case generator.

Data set #	accountNum	balance	amount
99	*7*	*6*	*2*
105	*7*	*6*	*3*
106	*7*	*6*	*4*

Fig. 9. Data sets that satisfy method predicate for *Withdraw*

3.2.3 Test Case Generator

The test case generator is responsible for filling the test data in empty test shells. Each test case is formed by a set of values of instance variables (representing the current object *this*) and a test template with parameter values for all the methods. The following are the test cases generated for test template TEST 1 of our BankAccount example:

```
BEGIN TEST 1-1
  accountNum = 7
  balance = 6
  Create <>
  Withdraw <2>
  Delete <>
END TEST 1-1

BEGIN TEST 1-2
  accountNum = 7
  balance = 6
  Create <>
  Withdraw <3>
  Delete <>
END TEST 1-2

BEGIN TEST 1-3
  accountNum = 7
  balance = 6
  Create <>
  Withdraw <4>
  Delete <>
END TEST 1-3
```

4 Conclusion

In this paper, we have presented a novel framework to automatically generate test cases for a class from its IFAD VDM++ specification. The ideas presented in the paper can be generalized to other model-based object-oriented formal specification languages as well. The proposed technique requires a VDM++ specification, and a test descriptor, to generate test cases.

References:

[1] Atterer, R.: "Automatic Test Data Generation from VDM-SL Specifications"; The Queens University of Belfast, April 2000.

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

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

[4] Fletcher, R. S.: "Testing of Object-oriented Software using Formal Specifications"; Masters Thesis, Department of Software Development, Monash University, April 1994.

[5] Helke, S., Neustupny, T., Santen, T.: "Automating Test Case Generation from Z Specifications with Isabelle"; in proceedings of the 10th International Conference of Z Users, 1997, Springer-Verlag.

[6] Hörcher, H.M.: "Improving Software Tests using Z Specifications"; in proceedings of 9th International Conference of Z Users, 1995, Springer-Verlag.

[7] "VDMTools: The IFAD VDM++ Language"; IFAD, Forskerparken 10A, DK-5210, Odense M., 1999; http://www.ifad.dk.

[8] Kirani, S., Tsai, W. T.: "Specification and Verification of Object-oriented Programs"; Technical Report, Computer Science Department, University of Minnesota, Minneapolis, December 1994.

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

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

[11] Mikk, E.: "Compilation of Z Specifications into C for Automatic Test Result Evaluation"; in proceedings of the 9th International Conference of Z Users, 1995, Springer-Verlag.

[12] Nadeem, A., Rehman, M. J.: "A Framework for Automated Testing from VDM-SL Specifications"; in proceedings of 8th IEEE-INMIC Conference, 24-26 December 2004.

[13] Van Aertryck, L., Benveniste, M., Le Métayer, D.: "CASTING: A Formally Based Software Test Generation Method"; Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM'97), 1997.

[14] Turner, C. D., Robson, D. J.: "A Suite of Tools for the State-based Testing of Object-oriented Programs", TR-14/92, Technical Report, Computer Science Division, School of Engineering and Computer Science (SECS), University of Durham, Durham, England, April 1993.