Journal of Universal Computer Science, vol. 11, no. 6 (2005), 962-985
submitted: 4/10/04, accepted: 3/5/05, appeared: 28/6/05 © J.UCS

TESTAF: A Test Automation Framework for Class Testing using Object-Oriented Formal Specifications

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

Muhammad Jaffar-ur-Rehman
(Center for Software Dependability,
Mohammad Ali Jinnah University, Islamabad, Pakistan
jaffar@jinnah.edu.pk)

Abstract: In this paper, we present a novel framework TESTAF to support automatic generation and execution of test cases using object-oriented formal specifications. We use IFAD VDM++ as the specification language, but the ideas presented can be applied equally well to other object-oriented formal notations. The TESTAF framework requires a VDM++ specification for a class, a corresponding implementation in C++, and a test specification, to generate and execute test cases, and evaluate the results. The test specification defines valid test sequences in an intermediate specification language based on regular expressions. The framework uses the formal specification of the class, and the test specification to generate empty test shells, which are then filled in with the test data to create concrete test cases. The test data for a method are generated from the input space defined by the method pre condition and the class invariant. The TESTAF applies boundary value analysis strategy to generate the test data. A test driver then executes the implementation with the test data, and uses a conjunction of method post condition and the class invariant as a test oracle to evaluate the results, while reporting failed test cases to the user.

Key Words: automated testing, formal specification, object-oriented software

Category: D.2.5

1 Introduction

Using formal methods can help avoid specification errors, ambiguities, and misinterpretations in early phases of software life cycle. Unlike natural languages, formal languages are based on sound mathematical principles, and allow aspects of the specification to be rigorously demonstrated using mathematical proofs. However, the use of formal specification methods provides no guarantee that the implementation will conform to the specifications. A formal proof of correctness, although possible, is not cost-effective for most software projects because of the complexity of large software systems. Even after a formal proof, testing is usually required to build confidence in the system being developed [Meudec, 98]. 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 using formal specifications, e.g., [Dick, 93] [Stocks, 96] [Meudec, 98] [Liu, 02]. Some other related works include [Van Aertryck, 97] [Atterer, 00] and [Helke, 97].

Considerable amount of research effort has been directed towards automatic generation of test suites from formal specifications, such as Z notation, the B method, and the VDM-SL specification language. The object-oriented dialects of formal notations help write a specification that uses the same concepts as the implementation in an object-oriented language, and hence provides a good basis for generation of test cases.

In this paper, we present a novel framework TESTAF that automates generation, selection, and execution of test cases from IFAD VDM++ [IFAD, 99] specifications. Although we use VDM++ as the specification language, the approach used in our framework can be generalized to other object-oriented formal notations as well. The major reason to choose VDM++ as the specification language was that it allows method pre and post conditions as well as class invariant to be explicitly specified. In contrast, if Z notation (or an object-oriented dialect of Z) is used as the specification language, the preconditions for operation schemas would have to be calculated.

The TESTAF framework requires a VDM++ specification, a corresponding implementation in C++, and a test specification, to generate and execute test cases, and evaluate the results. The test specification defines valid test sequences in an intermediate specification language based on regular expressions. The TESTAF first matches classes, their attributes and methods in the specification with corresponding classes, attributes and methods in the implementation, and stores the mappings in a file. This process is semi-automated – the TESTAF automatically generates the mappings, however the user is required to confirm or edit the mappings if needed. Alternatively, the user can manually create the mappings file for use by the TESTAF.

Then, TESTAF uses formal specification and the test specification to generate empty test shells. A test shell contains method sequences and templates for input data. The test data for test shells are generated from the input space created from method pre condition and class invariant of each method to be tested. Test data are selected from the generated input space using boundary value analysis, and filled into the test shells. A test driver then executes the implementation with test data, and uses method post condition and the class invariant as test oracle to evaluate the results, while reporting the failed test cases to the user.

The rest of this paper is organized as follows: section 2 surveys related work by other researchers; section 3 describes the working of the TESTAF framework in detail; section 4 presents a case study to show effectiveness of the TESTAF; and finally section 5 concludes the work.

2 Related Work

In this section, we present an overview of related works, and discuss how our framework differs from other closely related works.

Several researchers have proposed techniques to automatically generate test cases from formal specifications. However, most of the research in this area has focused on testing from non object-oriented formal specifications.

One of the well-known works in this area is a methodology proposed by Dick and Faivre to convert VDM-SL precondition expressions into a disjunctive normal form (DNF), so that a solution to each disjunct represents a solution to the entire expression [Dick, 93]. The state space generated by the precondition is then exhaustively searched using a Prolog tool to generate the test cases. In [Helke, 97], 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 proposed 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 [Meudec, 98]. The approach is based on parsing VDM-SL expressions, and is implemented in [Atterer, 00].

In [Hörcher, 95], an overview of testing based on Z is given, and a technique to transform Z operation schemas to DNF is proposed to generate test cases. However, the author emphasizes the need to automate test evaluation because of the vast amount of data to be processed. Mikk describes a test evaluation tool to support automatic test evaluation, by transforming Z schema predicates into executable forms which are then compiled to boolean-valued C functions [Mikk, 95].

All of the above works focus on techniques to automate testing from non object-oriented formal specifications. However, works focusing on testing from object-oriented formal specifications are relatively fewer. In [Carrington, 94] and [Stocks, 96], a test template framework has been proposed which uses the Z notation to generate test templates. This work has been further extended in [Carrington, 00] for specification-based class testing. The authors have shown their proposed framework to be flexible and by allowing test generation strategy to be specified. However, their framework has not been fully implemented.

The above work has also been further extended for object-oriented specifications in [Liu, 02]. It is based on ObjectZ 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.

Legeard and Peureux present a case study on generating test sequences for Smart Card GSM 11-11 standard [Legeard, 04]. 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 contrast to the Legeard and Peureux's work [Legeard, 04], the TESTAF framework presented in this paper supports testing of a message sequence, as well as an individual method of the class. When testing an individual method, it is only required that the class be in a correct state. The TESTAF achieves this by setting the appropriate values for the instance variables, rather than executing the preamble test path as in [Legeard, 04]. This approach saves the time required to compute and execute the test path.

In Legeard and Peureux'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. The TESTAF, in contrast, requires the user to manually specify the test paths to be tested, using a formal notation based on regular expressions. This makes the TESTAF more flexible, as the user can specify all valid test paths or a subset of it.

In [Boyapati, 02], 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 *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.

In contrast, the TESTAF framework is flexible enough to be easily generalized for other model-based formal notations and implementation languages.

3 Architecture of the TESTAF

The test automation framework consists of four main components (Fig. 1), i.e.

a) configuration matcher
b) predicate parser
c) test generator, and
d) the test driver

It requires a VDM++ specification, a corresponding implementation, and a test specification. The test specification contains valid sequences of operations defined in an intermediate language based on regular expressions [Fletcher, 94], and is defined manually by the user. To start the testing, the user selects the specification class and the corresponding implementation class to be tested. The configuration matcher matches the class configuration in VDM++ specification with that of the implementation. The purpose of this step is to match the class name, instance variable names, and method signatures in the formal specification with those of the implementation. Output of the configuration matcher is a file that contains mappings between names used in the specification and the names used in implementation.

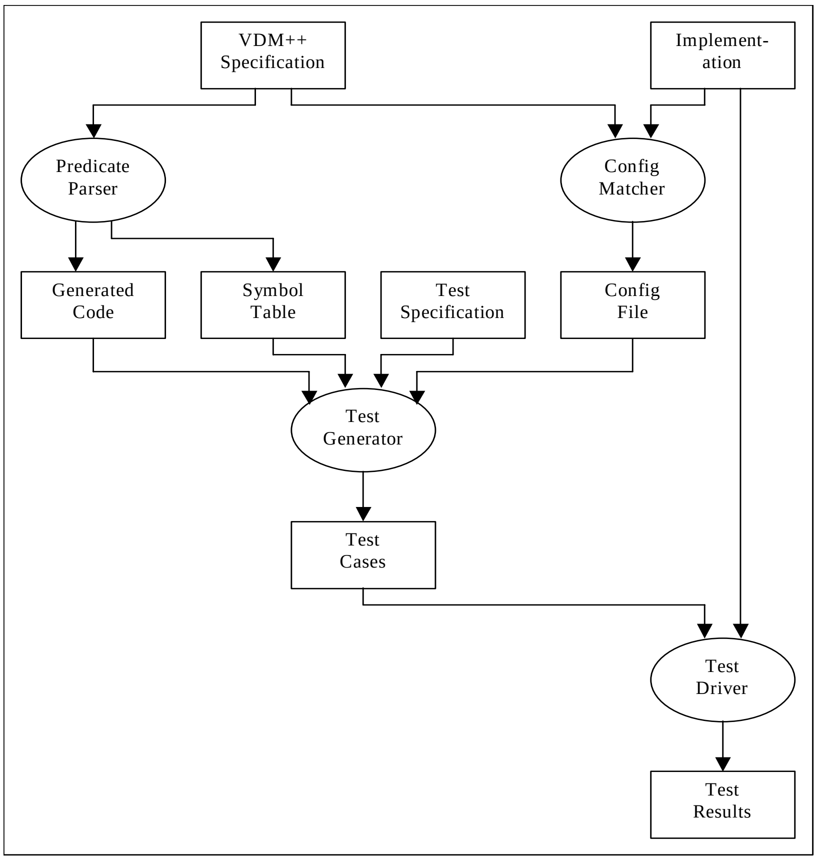

Figure 1: Architecture of the TESTAF

The predicate parser constructs a *method entry predicate* and a *method exit predicate* for each method in the class. The entry predicate is formed by a conjunction of the method precondition and the class invariant predicates. Similarly, the exit predicate is a conjunction of the method postcondition and the class invariant predicates. The parser then generates C++ code for both entry and exit predicates of each method. The generated code for entry predicate is used to filter the input data for a method, while the generated code for exit predicate is used to evaluate the results of method execution. The parser creates a *child wrapper* class as a subclass of the CUT (class under test), and includes the generated code as public methods in this class. The parser also creates a symbol table for the method precondition, which records variable names and their boundary values. This is used by the test generator to generate test data.

The test generator generates empty test shells from the test specification, and the configuration file, which are then filled with test data. The test data are generated from the symbol table created by the parser, and filtered by the method entry predicate.

Finally, the test driver executes the test cases on the implementation by instantiating the child wrapper, and evaluates results by executing code for the method postcondition and the class invariant. In the following subsections, we describe the working of each component of the framework.

In this section, we use an IFAD VDM++ specification for a class *NNcomplex* (a simple abstraction of the non-negative complex numbers) and its corresponding implementation as a running example to show how the TESTAF generates and executes test cases. The VDM++ specification for NNcomplex class and an implementation in C++ are shown in Fig. 2 and Fig. 3 respectively.

The class invariant *(re>=0) & (im>=0)* specifies that both real and imaginary components of the complex number must be non-negative. Four methods called *add, subtract, multiply*, and *divide* have been defined for the *NNcomplex* class, to perform the basic arithmetic operations on an *NNcomplex* object with an integer value.

The precondition for each method ensures that the result of operation will be a complex object with both real and imaginary components as non-negative. Precondition for the divide operation also prevents division of the complex object by zero.

3.1 Matching Specification with Implementation

In order to generate valid test cases for a class implementation, not only types of its attributes and method signatures are required but also names of attributes and methods must be known. As the names used in the implementation may be different from those used in the formal specification, the test generator must maintain mappings between the two to allow test generation from the formal specification.

The *configuration matcher* component of the TESAF is responsible for mapping names used in the specification with those of the implementation. This process is automated, however the user is allowed to modify or manually create the mappings file.

The configuration matcher first matches class names, by comparing number and types of attributes and method signatures. For instance, class A in specification matches with class B in the implementation if both A and B have the same number and types of attributes, as well as the same number of methods with matching signatures.

Class attributes are matched by their types. Likewise, method names are matched by their signatures (i.e., number and types of parameters). The table in Fig. 4 below shows how configuration matcher matches VDM++ types with those of C++. Currently, the TESTAF supports only the VDM++ types shown in Fig. 4. In the matching process, C++ type qualifiers (*long, short, signed, unsigned*) are ignored if type name is specified, otherwise the type name is assumed to be *int* (the default type in C++).

```
class NNcomplex
  instance variables
    re : real;
    im : real;

  inv (re>=0) & (im>=0);

  operations
    init : () ==> ()
    init () == ( re := 0;
                 im := 0; )

    add : int ==> NNComplex
    add(num) == ( re := re+num;
                  return self; )
    pre num >= -re;
    post re = re~+num;

    subtract : int ==> NNComplex
    subtract(num) == ( re := re-num;
                       return self; )
    pre num >= re;
    post re = re~-num;

    multiply : int ==> NNComplex
    multiply(num) == ( re := re*num;
                       im := im*num;
                       return self; )
    pre num >= 0;
    post re = (re~*num) and (im = im~*num);

    divide : int ==> NNComplex
    divide(num) == ( re := re/num;
                     im := im/num;
                     return self; )
    pre num > 0;
    post re = (re~*num) and (im = im~*num);

end NNcomplex
```

Figure 2: VDM++ Specification for NNcomplex class

The strategy of matching specification with implementation using types of attributes and method signatures works well in most cases. However, it may fail if two or more attributes in a class have the same type and scope, or two or more methods have the same signature and scope. For this reason, the TESTAF prompts the user to confirm each mapping before it is saved to the mappings file. Moreover, the file is saved in text format, and the user is allowed to modify its contents later.

Fig. 5 shows mappings file generated by the configuration matcher for the *NNcomplex* class.

```
class Complex {
  private:
    float re;
    float im;

  public:
    void init() {
      re = im = 0;
    }

    Complex add(int x) {
      re += x;
      return this;
    }

    Complex subtract(int x) {
      re -= x;
      return this;
    }

    Complex multiply(int x) {
      re *= x;
      im *= x;
      return this;
    }

    Complex divide(int x) {
      re /= x;
      im /= x;
      return this;
    }
}
```

Figure 3: Implementation of the NNcomplex class in C++

VDM++ Type	**Mapped to (C++ Type)**
bool	bool
int	int
nat	int
nat1	int
real	float, double
rat	float, double
char	char
quote type	enum type
seq and seq1 types	array type
map type	array of struct
object reference type	object reference type

Figure 4: Mappings of VDM++ types to C++ types

```
class NNcomplex -> Complex

  attributes
    re -> re
    im -> im

  methods
    init() -> init()
    add(int) -> add(int)
    subtract(int) -> subtract(int)
    multiply(int) -> multiply(int)
    divide(int) -> divide(int)
```

Figure 5: Mappings identified by the configuration matcher

3.2 Predicate Parsing

A formal specification in VDM++ contains pre and post conditions for each method of the class under test (CUT). The predicate parser constructs *method entry and exit predicates* for each method by forming a conjunction of method pre and post condition predicates with the class invariant predicate, as shown below:

$$method\_entry\_predicate = method\_precondition \wedge class\_invariant$$

$$method\_exit\_predicate = method\_postcondition \wedge class\_invariant$$

In addition to the method precondition and class invariant, a *method entry predicate* also includes *type constraints*. For instance, if an input parameter of the method, or an instance variable is of type *nat*, then it is implicitly implied that its value cannot be negative. The method predicates are parsed into parse trees using a context free grammar for VDM++ expressions. The TESTAF implements a simple LR parser to parse the expressions.

3.2.1 Generating Code for Method Predicates

From the parse tree, the parser generates C++ code to evaluate each method predicate. The idea of converting a predicate expression into a parse tree and generating C code from the tree, has been described in [Nadeem, 04]. The parser produces boolean-valued C++ functions named *classname_methodname_pre*() and *classname_methodname_post*() for each method in the CUT. Code generated for the Complex class is shown in Fig. 6 below. This code is used by the test generator and the test driver to filter the input data, and to evaluate the results, respectively.

```
bool Complex_init_pre(float re, float im) {
  bool result = true;
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_init_post (float re, float im) {
  bool result = true;
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_add_pre(float re, float im, int x) {
  bool result = true;
  result = result && (x >= -re);
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_add_post (float re, float re_old, float im, int x) {
  bool result = true;
  result = result && (re == re_old+x);
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_subtract_pre(float re, float im, int x) {
  bool result = true;
  result = result && (x >= re);
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_subtract_post (float re, float re_old, float im, int x) {
  bool result = true;
  result = result && (re == re_old-x);
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_multiply_pre(float re, float im, int x) {
  bool result = true;
  result = result && (x >= 0);
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_multiply_post (float re, float re_old, float im, float im_old, int x) {
  bool result = true;
  result = result && ((re == re_old*x) && (im == im_old*x));
  result = result && ((re >= 0) && (im >= 0));
  return result;
}

bool Complex_divide_pre(float re, float im, int x) {
  bool result = true;
  result = result && (x > 0);
  result = result && ((re >= 0) && (im >= 0));
  return result;
}
```

Figure 6: Code Generated by Predicate Parser

A predicate in VDM++ is a well-formed logical expression that involves

- clauses (relational sub-expressions)
- universal and existential quantifiers
- set membership sub-expressions
- logical connectives

At the present, the TESTAF supports only VDM++ predicates involving clauses, set membership sub-expressions, and logical connectives. A quantified expression involves evaluation of a boolean expression with multiple bindings of several variables. Because of the complexity of converting such an expression to a C++ equivalent, we have omitted the quantifiers from the current version of TESTAF. The table in Fig. 7 shows C++ expressions generated by TESTAF for various types of predicates.

VDM++ Predicate	**C++ Expression**
`a=b`	`a==b`
`a<b`	`a<b`
`a>b`	`a>b`
`a<=b`	`a<=b`
`a>=b`	`a>=b`
`a<>b`	`a!=b`
`not a`	`!a`
`a and b`	`a && b`
`a or b`	`a || b`
`a=>b`	`!a || b`
`a<=>b`	`a==b`
`a in set S`	$(a==s_1)$ || $(a==s_2)$ || $(a==s_3)$ ... where s_1, s_2, s_3, ... are elements of S
`a not in set S`	$(a!=s_1)$ && $(a!=s_2)$ && $(a!=s_3)$ ... where s_1, s_2, s_3, ... are elements of S

Figure 7: C++ boolean expressions for VDM++ predicates

3.2.2 Constructing the Symbol Tables

For each method in the CUT, a symbol table is constructed that stores instance variables, method arguments and their boundary values. The boundary values are determined from method entry predicates. The test generator uses symbol tables to generate test inputs for methods. For the *add* method of the *Complex* class, the TESTAF generates the symbol table shown in Fig. 8 below.

Var	Type	Rel. Op.	Boundary Value
re	*float*	>=	*0*
im	*float*	>=	*0*
x	*int*	>=	*-re*

Figure 8: Symbol Table for add() method

As the boundary value of *x* in Fig. 8 is dependent on the value of *re*, so the test generator must first generate test values for the variable *re*. A variable may have more than one boundaries if it appears in more than one clauses of the predicate expression. For instance, in the predicate expression *(a>10) & (a<20)*, the variable *a* has two boundary values, i.e., *10* and *20*. For such variables, there are multiple rows in the symbol table.

3.3 Generating Test Cases

The test generator component of TESTAF is composed of three smaller components, i.e.

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

In this subsection, we describe the working of these components.

3.3.1 Generating Test Shells

As the correct behavior of a class method may depend not only on the current state of the class object, but also on the correct sequence of messages passed to the object [Binder, 99], the TESTAF uses a test specification to determine the valid message sequences. However, if the correct behavior of a class is not dependent on its message sequences – as is the case in non-modal, or quasi-modal classes [Binder, 99] – then the test specification may be omitted. In this case, the TESTAF will independently test each method.

The test specification 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 [Fletcher, 94] and is based on the work of [Kirani, 94]. For instance, test specification for the *NNcomplex* class can be written as given in Fig. 9 below.

$$SeqSpec(NNcomplex) \Rightarrow init \cdot ProcessComplex$$
$$ProcessComplex \Rightarrow (add, subtract, multiply, divide)^*$$

Figure 9: Test Specification for the NNcomplex class

This states that any valid sequence begins with the *init* method, followed by *ProcessComplex*, where *ProcessComplex* is any combination involving zero or more calls to the methods *add, subtract, multiply,* and *divide.* The *· (dot)* operator specifies that the operations *init* and *ProcessComplex* must be invoked in sequence. The * operator specifies that the operations *add, subtract, multiply*, and *divide* can be repeated zero or more times. The test shell generator determines valid test sequences from this test specification and constructs test shells. A *test shell* is a sequence of test templates, where a *test template* consists of a method name followed by its parameter types. The test shell generator uses mappings from the configuration file to determine method names in the CUT, and saves the generated test shells in a file.

The algorithm employed by TESTAF to generate valid message sequences from a test specification is given in Appendix A. For instance, the following three test shells are constructed from the message sequences generated from the above expression.

```
BEGIN TEST 1
 init <>
 add <int>
END TEST

BEGIN TEST 2
 init <>
 add <int>
 subtract <int>
END TEST

BEGIN TEST 3
 init <>
 subtract <int>
 multiply <int>
 add <int>
 add <int>
END TEST
```

3.3.2 Generating Test Data

The test data generator determines *method inputs* for each method in the CUT. Method inputs consist of parameters of the method, including the implicit *this* parameter. It uses the symbol table (section 3.2) to generate test values for method inputs, and the code for method entry predicate to filter the test values. Using the boundary value analysis strategy, the TESTAF generates the following test values for the *add* method (Fig. 10).

For instance, for the variable *re*, the boundary value is *0*, therefore the generated test values are *0*, *1*, and *5*. While the values *0* and *1* are at the boundary, the value *5* is randomly generated from the space $re > 1$. Similarly, test values are generated for the variables *im* and *x*. A total of 27 sets of test values are thus formed (Fig. 10) for the *add* method. Each of the generated test sets is then executed on the method entry predicate *Complex_add_pre()* to test if it satisfies method entry predicate or not. The unsatisfiable test sets are eliminated. In our example, all 27 test sets are satisfiable. Unsatisfiable test sets may result if there are variables with multiple boundary values. For variables with multiple boundaries, all boundaries are used to generate test data. However, a test set contains only a single value for each variable. The generated test data are used by the test case generator to construct the concrete test cases.

3.3.3 Constructing Concrete Test Cases

The test case generator is responsible for filling the test data in empty test shells. For each input parameter of a method, the test data generator produces multiple test values using boundary value analysis. A *test set* is defined as a set of values of input parameters for a method. The test sets for a method are formed by taking a cross product of test values for the input parameters. The generated test sets are then filtered by executing them on the code for the method entry predicate.

```
1:      re = 0,  im = 0,  x = 0
2:      re = 0,  im = 0,  x = 1
3:      re = 0,  im = 0,  x = 9
4:      re = 0,  im = 1,  x = 0
5:      re = 0,  im = 1,  x = 1
6:      re = 0,  im = 1,  x = 9
7:      re = 0,  im = 8,  x = 0
8:      re = 0,  im = 8,  x = 1
9:      re = 0,  im = 8,  x = 9
10:     re = 1,  im = 0,  x = -1
11:     re = 1,  im = 0,  x = -1
12:     re = 1,  im = 0,  x = -1
13:     re = 1,  im = 1,  x = 0
14:     re = 1,  im = 1,  x = 0
15:     re = 1,  im = 1,  x = 0
16:     re = 1,  im = 8,  x = 12
17:     re = 1,  im = 8,  x = 12
18:     re = 1,  im = 8,  x = 12
19:     re = 5,  im = 0,  x = -5
20:     re = 5,  im = 0,  x = -5
21:     re = 5,  im = 0,  x = -5
22:     re = 5,  im = 1,  x = -4
23:     re = 5,  im = 1,  x = -4
24:     re = 5,  im = 1,  x = -4
25:     re = 5,  im = 8,  x = 6
26:     re = 5,  im = 8,  x = 6
27:     re = 5,  im = 8,  x = 6
```

Figure 10: Test values generated for add() method

For each method in a test shell, the generator generates valid test sets. The empty test shells are then filled in with all possible combinations of test sets for its methods, to form the concrete test cases.

Accessing private variables

As mentioned in section 3.3.2, the inputs to a method are not only its explicit parameters, but also the implicit *this* parameter, which represents state of the current object. When testing a method, the current object's state may also have to be set by setting values of its instance variables. By the principle of encapsulation, the instance variables of a class are kept private, so we must add getter and setter methods to the class under test to access and modify values of its instance variables.

Setting values of instance variables is required only when testing an individual method – the object must be in a correct state to accept the message. For example, the *add* message can be accepted only when the real and imaginary components of

Complex object have defined values, and are non-negative. However, when testing a message sequence, the instance variables are not required to be set. For instance, a valid message sequence requires *add* message to be preceded by *init* message, which will ensure correct object state.

The TESTAF framework supports both individual method testing, and a message sequence testing.

Testing an individual method

When testing an individual method, the values of instance variables *re* and *im* are set via setter methods. For instance, to test the add method of Complex class, using test values of Fig. 10, the following test cases are generated:

```
BEGIN TEST add.1
 set_re <0>
 set_im <0>
 add <0>
END TEST


BEGIN TEST add.2
 set_re <0>
 set_im <0>
 add <1>
END TEST

BEGIN TEST add.3
 set_re <0>
 set_im <0>
 add <9>
END TEST

etc.
```

The number of test cases increases exponentially if there are methods with multiple parameters, because, in such a case, all possible combinations of values of parameters are used to generate the test cases.

Testing a message sequence

When testing a message sequence, the object state is not required to be explicitly set – the correct state of the object for each method in the sequence is ensured by its preceding messages. For the example *Complex* class, using test values from Fig. 10, and test shell 1, the following test cases are generated:

```
BEGIN TEST 1.1
 init <>
 add <0>
END TEST

BEGIN TEST 1.2
 init <>
 add <1>
END TEST

BEGIN TEST 1.3
 init <>
 add <9>
END TEST
```

3.4 Test Driver

For the purpose of testing, a test class can be derived from the CUT as suggested in [Turner, 93]. The derived class is called a *child wrapper*. It inherits all the attributes and methods from the CUT. The extra routines required for testing are added to the child wrapper class, rather than patching an existing class of the system. The test driver instantiates the child wrapper, and invokes its methods to be tested.

The TESTAF implements the strategy described above, i.e., it creates a child class of the CUT and adds its testing methods. Under this mechanism, the class that actually gets tested is the child wrapper rather than the CUT. However, the methods under test are actually implemented in the CUT, so they get tested. This strategy relies heavily on the programming language's inheritance mechanism.

The child wrapper class contains the following additional methods, used for testing:

load(TestCase tc)	– used to load a test case from the file; *tc* is the test case number.
execute()	– used to execute a loaded test case.

The test driver instantiates the child wrapper to create a test object, and then executes each test case with the test object. The *execute()* method of child wrapper invokes each method in a test case in sequence. For instance, for the test case 1.3 of section 3.3.3, the actual method calls made by *execute()* are:

```
init()
Complex_init_post()
add(9)
Complex_add_post()
```

After execution of each method, the method's *exit predicate* (described in section 3.2) is evaluated, and the results are logged in a file. For failed test cases, the execute() method also logs values of variables for which exit predicate failed.

4 Case Study

In this section, we present a case study that demonstrates effectiveness of TESTAF using an example of specification for an address book that maps names to addresses.

4.1 Setup for the Case Study

Setting up the case study for TESTAF requires a VDM++ specification of the CUT, an implementation for the CUT, and a test specification for the CUT. In Fig. 11, and Fig. 12, we present a VDM++ specification, and an implementation, respectively, for the AddressBook case study.

```
class AddressBook
  types
    Name = seq of char;
    Address = seq of char;

  instance variables
    book : map Name to Address;

  operations
    init : () ==> ()
    init () == ( book := {|->}; )

    insert : Name * Address ==> ()
    insert(name, addr) ==
          ( book := book munion {name |-> addr}; )
    pre name not in set dom book;
    post name in set dom book &
         book(name) = addr;

    update : Name * Address ==> ()
    update(name, addr) ==
          ( book := book ++ {name |-> addr}; )
    pre name in set dom book;
    post book(name) = addr;

    delete : Name ==> ()
    delete(name) == ( book := {name} <-: book; )
    pre name in set dom book;
    post name not in set dom book;

    lookup : Name ==> Address
    lookup(name) == ( return book(name); )
    pre name in set dom book;

end AddressBook
```

Figure 11: VDM++ Specification for an AddressBook class

```
class AddressBook
  private:
    struct AddrBook {
      char[20] Name;
      char[50] Address; };

    AddrBook[100] book;
    int top;  // keeps track of number of entries in
              // the address book

  public:
    void init () { top = -1; }

    void insert(char[] name, char[] addr) {
      top++;
      book[top].Name = name;
      book[top].Address = addr;
    }

    void update(char[] name, char[] addr) {
      int i=0;
      while ((book[i].Name != name)&&(i<=top)) i++;
      if (book[i].Name==name)
        book[i].Address = addr;
    }

    void delete(char[] name) {
      int i=0;
      while ((book[i].Name != name)&&(i<=top)) i++;
      if (book[i].Name==name) {
        while (i<top) {
          book[i].Name = book[i+1].Name;
          book[i].Address = book[i+1].Address;
          i++;
        }
        top--;
      }
    }

    char[] lookup(char[] name) {
      int i=0;
      while ((book[i].Name != name)&&(i<=top)) i++;
      if (book[i].Name==name)
        return book[i].Address;
    }

end AddressBook
```

Figure 12: An Implementation of the AddressBook class in C++

Using the intermediate specification language described in [Fletcher, 94], the valid sequences of execution for the *AddressBook* class were defined by the specification given in Fig. 13 below.

$$\boxed{\begin{array}{l} SeqSpec(AddressBook) \Rightarrow init \cdot ProcessAddressBook \\ ProcessAddressBook \Rightarrow ((insert \cdot update^* \cdot delete)^* \leftrightarrow (lookup)^*) \end{array}}$$

Figure 13: Test Specification for the AddressBook class

This specification is interpreted as follows: a test sequence begins with a call to *init*, followed by *ProcessAddressBook,* where *ProcessAddressBook* has two parts:

- the first part *(insert · update* · delete)** states that every valid processing of address book begins with *insert* operation, which may be followed by zero or more calls to *update* operation, which in turn is followed by an optional call to *delete* – and the whole sequence can be repeated zero or more times. The *· (dot)* operator specifies that the operations *insert*, *update**, and *delete* can only be invoked in a sequence. A sequence may, however, omit the *delete* operation, or both *update** and *delete* operations. The * operator specifies that the operation can be repeated zero or more times.

- the second part consists of the *lookup* operation, which may be called at any time during the first part. The ↔ operator specifies that the operation(s) following it can be invoked at any time. The * operator specifies that the lookup operation can be called any number of times.

To summarize, the following constraints are imposed on message sequences that can be sent to an *AddressBook* object:

- every message sequence begins with the init message, and this message can be sent only once.

- the lookup message can be sent any number of times anywhere after the init message

- in any valid sequence there will be at least as many insert messages as delete messages – a delete message can only appear in a sequence if a corresponding insert message has appeared.

- update message can appear any number of times, but it must be preceded by at least one insert message.

4.2 Results and Discussion

The TESTAF read the test specification from a text file, and generated empty test shells. Some of the test shells generated by TESTAF are shown below:

```
BEGIN TEST 1
 init <>
 insert <char[], char[]>
 delete <char[]>
 lookup <char[]>
END TEST

BEGIN TEST 2
 init <>
 lookup <char[]>
 insert <char[], char[]>
 insert <char[], char[]>
 update <char[], char[]>
END TEST

BEGIN TEST 3
 init <>
 insert <char[], char[]>
 lookup <char[]>
 update <char[], char[]>
 delete <char[]>
END TEST
```

The generated test shells were saved. The test generator then generated the test data for each method using the symbol table produced by the parser. The boundary value analysis strategy was applied to generate the test data. For instance, for the *delete* operation, the precondition is:

name in set dom book

This states that the value of *name* must exist in the domain of the map *book*. For character strings, there is no value at the boundary – all values are either in the boundary, or out of boundary. So the test generator computes domain of the *book*, then selects a random value from the domain, and assigns to the variable *name*. For the insert operation, the precondition is:

name not in set dom book

So, a valid value for *name* would be any value which is not already in the domain of *book*. Initially, the *AddressBook* is empty, so the first *insert* message can accept any value for the *name* parameter.

The generated test sets for each method were then filtered by method entry predicates, and concrete test cases were formed. The test driver executed all the test cases on the implementation and reported no bugs. The results of execution were logged in a file.

The tests generated and executed by the TESTAF for the case study covered all possible message sequences (as specified in test specification) with loop coverage of up to two iterations. This is due to the default setting of *loop coverage* in TESTAF, which can be changed if greater coverage of the loops is required. Each message sequence was tested with all generated test values.

Increasing the loop coverage exponentially increases the number of test cases, since each loop iteration gets tested with all possible test values generated for the input variables. Furthermore, an iterative message can appear at multiple positions in a message sequence, and there can be more than one iterative messages in a test specification, resulting in the generation of multiple message sequences. For instance, the message *lookup()* in the case study can appear anywhere in a message sequence.

The boundary value analysis used to generate test values ensures that each method is tested at the boundary values of its inputs where there are greater chances of errors. However, for the character strings there are no values at the boundaries – each value is either in the boundary or out of the boundary. In the AddressBook case study, the inputs to all the methods (except the *init()* method) were character string type parameters, which resulted in a significantly lower number of generated test cases. This was because only one input value was generated for each input variable.

5 Conclusion and Future Work

In this paper, we have presented and demonstrated a framework TESTAF that automatically generates test cases from a VDM++ specification, executes the test cases on the implementation, and evaluates the results. Test cases for a method are generated from a conjunction of its precondition and the class invariant predicates, which define the input space of the method. Results are evaluated by evaluating the conjunction of method postcondition and the class invariant predicates.

The major contribution of this work is the presentation of a framework that integrates most of the testing activities, at the unit testing level. The framework allows automated testing of a class with minimal human intervention. The TESTAF framework uses VDM++ as the specification language, and C++ as the implementation language, but its design is flexible enough to allow support for other specification and implementation languages in the future.

We have tested our framework using various specification/implementation classes as the inputs, and found that it effectively generates, and executes test cases. A representative case study showing effectiveness of the framework is given in section 4 of this paper.

The quality of specification-based testing is, however, limited by the quality of the specification itself. The goal in specification-based testing is to demonstrate that the implementation conforms to the specification. However, if the specification is incorrect, or does not meet user requirements, then demonstrating that an implementation conforms to the specification would not be of much use. Another issue pertaining to the specification-based testing is that the functional specifications usually describe what the system must do when valid inputs are given (or certain conditions are satisfied), but they usually omit a description of what the system should do when invalid inputs are given. For this reason, only positive testing can be performed. The TESTAF also uses method precondition and class invariant to filter the generated test data for a method, therefore only positive test cases are generated.

An obvious direction for the future work is to enhance the framework to allow testing of class interactions and hierarchies. We believe that the TESTAF can be used as a solid foundation for making such enhancements in the future.

References

[Atterer, 00] Atterer, R.: “Automatic Test Data Generation from VDM-SL Specifications”; Diploma thesis; The Queens University of Belfast, April 2000.

[Binder, 99] Binder, R.V.: “Testing Object-Oriented Systems: Models, Patterns and Tools”; Addison-Wesley Object Technology Series, 1999.

[Boyapati, 02] Boyapati, C., Khurshid, S., Marinov, D.: “Korat: Automated Testing Based on Java Predicates”; ACM ISSTA 2002.

[Carrington, 94] 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.

[Carrington, 00] 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.

[Dick, 93] 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.

[Fletcher, 94] Fletcher, R. S.: “Testing of Object-oriented Software using Formal Specifications”; Masters Thesis, Department of Software Development, Monash University, April 1994.

[Helke, 97] 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.

[Hörcher, 95] Hörcher, H.M.: “Improving Software Tests using Z Specifications”; in proceedings of 9th International Conference of Z Users, 1995, Springer-Verlag.

[IFAD, 99] “VDMTools: The IFAD VDM++ Language”; IFAD, Forskerparken 10A, DK-5210, Odense M., 1999; http://www.ifad.dk.

[Kirani, 94] Kirani, S., Tsai, W. T.: “Specification and Verification of Object-oriented Programs”; Technical Report, Computer Science Department, University of Minnesota, Minneapolis, December 1994.

[Legeard, 04] Legeard, B., 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.

[Liu, 02] 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.

[Meudec, 98] Meudec, C.: “Automatic Generation of Software Test Cases From Formal Specifications”; Ph.D. thesis, The Queen’s University of Belfast, May 1998.

[Mikk, 95] 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.

[Nadeem, 04] Nadeem, A., Rehman, M. J.: “A Framework for Automated Testing from VDM-SL Specifications”; in proceedings of the 8th IEEE-INMIC Conference (INMIC 2004), Lahore, Pakistan, December 2004.

[Stocks, 96] Stocks, P., Carrington, D.: "A Framework for Specification-Based Testing"; IEEE Transactions on Software Engineering, vol. 22, no. 11, pp. 777-793, Nov. 1996.

[Turner, 93] 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.

[Van Aertryck, 97] 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.

Appendix A: Test Sequence Generation Algorithm

The following algorithm is implemented in the TESTAF to generate valid test sequences from a test specification. The following points may be noted about the algorithm:

- the * operator in a test specification implies that the message (or message sequence) can be repeated any number of times in a valid sequence – however, in the algorithm, we repeat the message up to two times only, to avoid generation of infinite number of test sequences.

- the function insert(TSset$_1$, TSset$_2$) constructs a set of test sequences by forming all possible combinations in which test sequences of TSset$_2$ are inserted into test sequences of TSset$_1$.

- The + operator in the algorithm is used as separator to separate two consecutive messages in a test sequence.

```
function generateTestSeqs (S : TestSpecification): set of TestSeq {

        TSset : set of TestSeq;
    TSset := [ ];

        if (S is in set of message names)
                TSset := TSset U S;

    else if (S is of the form $S_1$ , $S_2$) then
        TSset := TSset U generateTestSeqs($S_1$)
                                        U generateTestSeqs($S_2$);

    else if (S is of the form $S_1 \cdot S_2$) then
        TSset := TSset U generateTestSeqs($S_1$)
                                        U generateTestSeqs($S_1$+$S_2$);

    else if (S is of the form $S_1$*)
        TSset := TSset U generateTestSeqs($\varepsilon$) U generateTestSeqs($S_1$);
                                    U generateTestSeqs($S_1$+$S_1$);

    else if (S is of the form $S_1 \leftrightarrow S_2$) {
        TSset$_1$, TSset$_2$ : set of TestSeq;
        TSset$_1$ := generateTestSeqs($S_1$);
        TSset$_2$ := generateTestSeqs($S_2$);
        TSset := insert(TSset$_1$ , TSset$_2$);
        }

    return TSset;
}
```