Abstract—Formal Specification languages are being widely used for system specification and testing. Highly critical systems such as real time systems, avionics, and medical systems are represented using Formal specification languages. Formal specifications based testing is mostly performed using black box testing approaches thus testing only the set of inputs and outputs of the system. The formal specification language such as VDM++ can be used for white box testing as they provide enough constructs as any other high level programming language. In this work, we perform data and control flow analysis of VDM++ class specifications. The proposed work is discussed with an example of SavingAccount.

Keywords—VDM-SL, VDM++, data flow graph, control flow graph, testing, formal specification.

I. INTRODUCTION

The use of formal languages is rising for the systems that are more safety critical such as real time systems, avionics, medicine etc. Formal languages provide an unambiguous and clear representation of the system specification [14]. Formal languages are being used not only for the specification and modeling of the system but they are also a key artifact for testing the system. Test cases are generated from the specification of the system and are applied on the implementation. This provides the conformance of specifications in a system with its implementation [22].

A large number of specification based testing techniques are cited in literature such as using Z specification [21], VDM specification [1], B specifications [20], etc. Formal specification based testing is mostly performed using the black box testing approaches such as boundary value analysis [4, 5, 10], partition analysis [9, 11], classification tree method [23]. As the formal specification languages provide system specifications at a higher abstraction level in declarative form [14].

Formal Specification based test case generation can be performed using white box or code based testing approaches. White box testing is possible only for the languages that provide imperative language constructs such as VDM++. VDM++ is a language that provides completely executable specifications as any other high level programming language. We can use the VDM++ specification to test the system’s code level details at an early stage in development.

Many researchers have proposed approaches for testing of VDM++ [5, 8, 9, 10, 15] specifications. However, in all of the proposed approaches black box testing have been used.

Such as some of the approaches [9, 11] have used partition analysis and some [4, 5, 10] have designed test cases through boundary value analysis. Nadeem et al. [5] used VDM++ to test the inheritance and polymorphic behavior of object oriented systems. We intend to extend the use of VDM++ for data flow and control flow analysis. The data flow shows the definition and use of variables and tells about how the data routes [18]. Similarly, control flow shows how functions are executing and examines the branch and loop structure of the programs. The data flow and control flow analysis is further used for test case generation. We have applied our proposed approach on a case study of Saving Account VDM++ class.

The rest of the paper is organized as follows. Section II gives a review of the literature. In section III we discuss the proposed data and control flow of VDM++ specifications. Section IV provides the proof of work in the form of case study. Section V concludes the paper.

II. LITERATURE REVIEW

The emphasis of our literature review is on testing approaches in VDM and VDM++. Overviews of the approaches that we have surveyed are as follows:

Fitzgerald et al. [8] worked on validation of system level timing properties in formal models of distributed real time embedded systems. The validation of inconsistencies between those distributed applications is the main concern of the approach. The informal model constructed from rules of the system is transformed into VDM++ specification model. The model is then formally tested. The construction of model is automated. Macedo et al. [19] proposed an approach where abstract system specifications of functional and timing properties are added with details. These details are added through intermediate models expressing architecture of the system, concurrency and timing behaviors. The model is then validated through scenario based testing.

Nadeem et al. [9] introduced the technique of testing inheritance relationship using the VDM++ specification. By using synchronization constraints provided by VDM++ all valid sets of sequence of operations of a class are specified. As a result of operation sequence and partition predicate a test model is constructed that are used in test case generation. Another approach by Nadeem et al. [10] has presented a new idea to generate test cases automatically from VDM++ specification. The testing in this approach is based on the fault model presented by subtype inheritance and polymorphism testing presented by Offutt et al. [13]. In the VDM++ specification of a class a trace structure is specified which defines the valid sequence of method invocations of class for an individual object of a class. From these trace structures; test sequence generator constructs valid sequence of operations of
a class. The partition analyzer makes conjunctions of class invariants predicate with precondition predicate of each method in class. With the help of partition analysis, test data is generated for each operation in operation sequences to be tested. Verhoef et al. [15] have proposed the extension to the VDM++ to handle the problems of the system in distributed and real time environments. Initially authors have discussed the existing notation of VDM++ and timed extensions.

Nadeem et al. [5] proposed a framework which supports the automatic generation and execution of test cases from object oriented formal specification. The framework requires VDM++ specification and its corresponding implementation in C++. With C++ code the symbol table is also constructed, the boundary values of which are used by test generator to generate test cases. After generation of test cases, test driver executes test cases on implementation. An approach of parsing a VDM-SL specification to generate C code and test data, was developed [4], where test case generation is done by converting pre condition and post condition predicate into C function and modify the source code to evaluate each precondition before every function calls. Test cases are generating from pre condition predicate expression by parsing them and partitioning the input domain. The test driver then executes the generated test cases on the modified code and evaluates test results by executing code for post condition.

Droschl [6] proposed an approach for developing the test cases from the collection of valid sequence of events. Test case generator creates test suites which are then submitted to VDM Tools. Then, the VDM Tools supports analysis of specification by animation and test. Author has implemented the approach on a comprehensive case study named as Access control system. The case study is focusing on the features digital video recording and automatic door control of the system. The paper [7] explores the possibilities of automatic black box testing through formal requirements specification. Author has presented the framework focusing the use of formal requirements specification which is used in making the abstract test oracles for concrete implementation. The approach uses retrieve function to map the concrete input and output to abstract representation.

Agerholm et al. [3] presents a report on case study conducted at Dessault Electrinique in which they focused on the suitability of VDM technology for early software development phases before detailed design when requirements are not confirmed and still there need of customer feedback. The example used by the case study is real metro application. Jeremy et al. [11] have presented the methodology of partition analysis in model based specification of VDM. The presented approach is based on partition analysis by using state based specification. Where, the division method is achieved by transforming the relations into disjunctive normal forms (DNF). It is also used to determine test cases for every individual operation.

Scullard [12] have described the validation process of design, adapted by a very large scale integration (VLSI) distributed array processor (VDAP) project. Hardware is designed in this project by using informal design process, but tools and methods of VDM helps in defining testing strategy. Generation of tests in [12] is done by translating the very old level interface specification into VDM.

All of the approaches have used the black box testing strategies. In most of the approaches [3, 4, 5, 6, 7, 8] unit level testing have been done while some [9, 10] have handled the inheritance and polymorphic relationships. For designing test cases different black box testing methods such as boundary value analysis [4, 5, 10] and equivalence partitioning [9, 11] have been used. As a test input some approaches [9, 10, 11] have used DNF expression while some [6, 8] have used the sequence of events. Some of the approaches have mentioned the test coverage criteria [11], test data generation [3, 4, 5, 7, 9, 10, 11, 12] and test case generation [4, 5, 7, 8, 9, 11, 12]. In surveyed approaches analysis of results has been carried out either by a case study [3, 6, 8, 12, 15, 19] or with an example [4, 5, 7, 9, 10, 11]. Additionally for making the approach more understandable structural [4, 5, 7, 8, 9, 10] and behavioral [3, 6, 11, 12] elements have been used. Most of the approaches [5, 6, 7, 8, 9, 10, 15] have used the automatic support of VDMTools.

III. DATA AND CONTROL FLOW ANALYSIS OF VDM++ CLASS SPECIFICATIONS

We use VDM++ specifications for data flow and control flow analysis. This analysis is further used for test case generation. An abstract model of the proposed work is presented in figure 1.

Following is a brief description of the proposed approach activities:

- VDM++ class specifications are used as input artifact for generating control flow graph. A control flow graph is a representation of control transfer within a class.
- The control flow graph is annotated with def-use annotations.
- Extracting def-use associations of data members of a class.
- A coverage criterion is applied on def-use associations to generate test cases. The generated test cases are based on only def-use values.
Now we discuss all the activities with an example of SavingAccount class. The SavingAccount specifications are presented in VDM++ in figure 2. A given SavingAccount class has two instance variables of type real. Afterwards it has an invariant on global variable “bal”. It has two operations of withdraw and postInterest which have preconditions and postconditions for those operations.

A. VDM++ Control Flow Graph

A control flow graph (CFG) is a directed graph in which node represents block of statements while the edges represent the control flow between statement blocks [18]. In constructing a control flow graph the emphasis is on control transfer within a class. A control flow graph of a class SavingAccount is presented in figure 3. We have constructed the following control flow graph by considering statement of specification as nodes and their control transfer as edges. For example after an entry node, at second node we have definition of variables and branch shows the viability and disagreement of a condition.
B. Annotation of CFG with Def-use

Data flow graph represents the definition of variables and their use in the program [18]. In data flow graph occurrence of variable is classified as definitional occurrence (def), computational-use(c-use) occurrence and predicate use (p-use) occurrence. Where assignment statement contains the c-use of variable followed by def of variable and input statement also contains the def of variables. Similarly an output statement contains c-use of variable while the conditional transfer statement contains p-use of variable [18]. We use the def use analysis for VDM++ specifications. For doing def-use analysis we have considered an input and definition variables as def (variable) in an appropriate statement. While an output statement such as “bal = bal - amt” have a c-use of a data member “amt” and “bal” followed by variable “bal”. Additionally a conditional statement contains a p-use such as “bal > minbal” have p-use (bal, (sourcenode, targetnode)). In figure 4 we have presented the data flow graph of the example mentioned above.

Algorithm

```plaintext
EDGE { char edgelabel[ ]; EDGE *edge; }
DataFlowGraph() {
    N[i][j] : string
    create EDGE e
    for i=0; i<=sizeof(N[ ][ ]); i++, repeat step 4
    for j=0; N[i][j] != Null; j++ repeat step 5
    if N[i][j] == assignment || N[i][j] == def
        create e : EDGE
        e.edgelabel = def_use(var, e)
        else if N[i][j] == output || N[i][j] == c-use
            then create e : EDGE
            e.edgelabel = c_use(var)
        else if N[i][j] == predicate || N[i][j] == p-use
            then create e : EDGE
            e.edgelabel = p_use(var, (var, e))
        else
            create e : EDGE
            e.edgelabel = end
    }
}
```

Here we present the algorithm used to generate the def-use analysis of VDM++ class specifications.

Fig. 3: Control flow graph of class SavingAccount

Fig. 4: Data flow graph of class SavingAccount
C. Def-use Associations

Def-use association means an association between the definition of a variable and the node where it is used. The use of variable may be either computation or predicate. Table shows the definition and use associations of variables in a class SavingsAccount.

Consider the following statements of specification,

```
Balance:int
if Balance>=minBalance
  Balance=Balance-amount
```

The first statement contains the definition (def) of data member “Balance” and second statement has predicate use (p-use) of data member Balance. While, the third statement contains the c-use of Balance and amount followed by def-use of data member Balance.

<table>
<thead>
<tr>
<th>Variables</th>
<th>d-c-use</th>
<th>d-p-use</th>
</tr>
</thead>
<tbody>
<tr>
<td>instrRate</td>
<td>[6]</td>
<td></td>
</tr>
<tr>
<td>minBal</td>
<td>[6]</td>
<td>[6]</td>
</tr>
<tr>
<td>bal</td>
<td>(2,3),</td>
<td>(2,7),</td>
</tr>
<tr>
<td></td>
<td>(4,5),</td>
<td>(4,6)</td>
</tr>
<tr>
<td>Amt</td>
<td>[5]</td>
<td></td>
</tr>
<tr>
<td>Bal</td>
<td>(2,3),</td>
<td>(2,7),</td>
</tr>
<tr>
<td></td>
<td>(4,5),</td>
<td>(4,6)</td>
</tr>
<tr>
<td>bal</td>
<td>(2,3),</td>
<td>(2,7),</td>
</tr>
<tr>
<td></td>
<td>(4,5),</td>
<td>(4,6)</td>
</tr>
</tbody>
</table>

VI. CONCLUSION

In this paper, we present the def-use analysis of VDM++ class specifications. VDM++ specifications are formally used for testing approaches such as partition analysis and boundary value analysis. VDM++ specifications provide enough constructs for def-use analysis. Def-use analysis provides inner details of a class that is specific to testing. The def-use of VDM++ can be performed like any high level programming languages. We present the approach with the help of a well used example of a SavingsAccount.

As a future work we intend to work on def-use analysis at an integration level that involves many classes. We are also working on automation of the proposed approach.

REFERENCES


