VDM Specification of an Algorithm for Graph Decomposition

Abdul Huq
Middle East College of Information Technology, Sultanate of Oman
huq@mecit.edu.om

Narayanan T. Ramachandran
Middle East College of Information Technology, Sultanate of Oman
narayanan@mecit.edu.om

JDIM
Journal of Digital Information Management

ABSTRACT: *Complex software systems need a precise specification of their intended behaviour. While graph theory plays an important role in several aspects of building such software systems, formal methods provide a rigorous mathematical framework within which a system can be described. In this paper, an algorithm for decomposing a graph into disjoint paths is presented. Such decompositions are useful, particularly in implementing software testing. VDM representation of the proposed algorithm facilitates automation of testing process without any ambiguity.*

Categories and Subject Descriptors
D.2.6 [Programming Environments]; Graphical Environments: D.2.5 [Testing and debugging]; I.3.6 [Methodology and Techniques]; Graphics data structures and data types

General Terms
Graph Theory, Software Testing, Graph Decomposition

Keywords: Vienna development method, Graphs decomposition algorithm, VDM

1. Introduction

There are several options to describe software properties. Natural languages are expressive but imprecise. Natural language descriptions carry considerable noise, ambiguities and contradictions [16], necessitating alternative techniques. Formal methods based on mathematical specifications have been found quite useful in this context. Extensive studies have been conducted to analyse their potential influence in complex and critical applications [9,10,21].

Discussing the need for and the application of formal methods to complex systems, Alagar et al. [1] present a case study on the specification of a robot based assembly system. **Larsen et al. [14] assess the effectiveness of employing formal methods in different activities of the development of a security critical system.** Emphasising the need for practical guidelines to make the best use of formal specifications, Palshikar provides tips to be used in formal specification in industrial applications[19].

Choice of appropriate formal method is influenced by several factors. These are discussed by Bacherini et al. [2] for an application in the development process of a railway signalling manufacturer. Hui et al. [11] discuss details of two kinds of specification languages viz., TLA and Larch in software development. Industrial applications of the specification language TRIO are illustrated by Ciapessoni et al. [7].

One popular formal method is the Vienna Development Method (VDM). A rigorous approach to this model-oriented specification method is found in [12]. VDM has been explored by several authors for different applications. Pedersen and Klein use it to formalise a communication protocol [20]. Comparing specifications in software cost reduction method and in VDM for a safety-critical system, Williams identifies several key assessment issues [24].

In this paper, an application of VDM is considered for the purpose of implementing an algorithm developed for decomposing a graph. Such a decomposition can be used in problems relating to software testing.

More specifically, a decomposition algorithm that produces several disjoint paths, the union of which results in the original graph has been developed and VDM specification of the process is described. The advantage of specifying the algorithm in VDM is to introduce rigour and to eliminate inconsistencies and ambiguity, facilitating code development for the purpose of automated implementation of the proposed algorithm.

The remainder of the paper is organised as follows: The algorithm is presented in Section 2. In Section 3, an application of the algorithm is described. The next Section discusses the VDM specification. It gives details of the terminology and notations used as well as the data structures and functions.

2. Graph Decomposition Algorithm

Graphs are very useful discrete structures in Computer science. Describing how graph properties are valuable for understanding the characteristics of the underlying software systems, usefulness of graph theory in object oriented systems has been explored in [6].

In the following paragraphs we describe a decomposition algorithm that produces disjoint paths, whose union is the original graph. In other words, given a graph G, with edge set E and node set V, the algorithm yields paths P_1 (with edge set E_1 and node set V_1), path P_2 (with edge set E_2 and node set V_2) …, path P_n (with edge set E_n and node set V_n) such that every member of E is in exactly one of $E_1, E_2, \ldots, E_n$ i.e., $E_1 \cup E_2 \cup \ldots \cup E_n = E$ and $E_i \cap E_j = \emptyset$, for $i \neq j$.

Different paths are obtained by successively collecting adjacent nodes, starting with an arbitrary initial node. The search for adjacent nodes to be included in the path continues until a node is repeated or a final node is reached. However, our search is on for the next path, if any.

Whenever an edge (v_i, v_j) is included in a path, it is deleted from E. Also whenever there is an isolated node we shall delete the same from V. The following algorithm describes more rigorously the operations involved in obtaining different paths which constitute the components of the decomposition.

Algorithm

Input: Graph G = (V, E)

Output: Set of components, whose disjoint union is G

```
1. Set   G ← Graph considered for
             decomposition
         V ← Set of nodes of G
         m ← 0 //path index
2. While (V ≠ ø) do
3.       Set E ← Set of edges in G
          m ← m + 1
4. Choose an initial node j in V
          Set P_m ← < j >
                t ← j
               repeated ← false // repeated is a boolean
      variable that becomes true when any node occurs for the
      second time
5.       While (there is a k such that (t, k)
         is in E and repeated =
         false) do
6.        If k is in P_m then
          Set repeated ← true
          Else
          Add k to P_m
          Remove (t, k) from E
          Set t ← k
7.        EndIf
8.       EndWhile
9.       V← V- {isolated nodes}
10. EndWhile
11. Set Components ← {P_1 , P_2 , …., P_m }
12. Return Components
```

3. An Application of the Algorithm

Testing is an important phase in any software development life cycle. It involves a systematic approach to imagine, explore and locate the weaknesses of a complex system and demonstrate how it needs to be corrected [3,13]. For this purpose software structures have to be abstracted. Graphs have been frequently used for such abstractions. Several authors have attempted a graph theoretic approach to testing issues [4,15,18,23].

In large real time systems, the following scenario is common: one or more modules call another module, or control is passed from one module to another. Considering the number of modules and the transfers of control from one module to another with associated parameters, there is a need to organise a systematic testing strategy. Such a strategy must enumerate and represent all transfers of control as well as find the most appropriate way to test them.

All the modules in a system along with their transfers of control can be represented as a graph with nodes representing modules and edges indicating transfers of control. For the purpose of testing, it is more convenient to study all the sub graphs of a graph than the graph itself. Hence we aim at decomposing a graph into several sub graphs (which are paths in the algorithm of Section 2) such that all the sub graphs collectively represent the original graph and no two sub graphs have an edge in common.

There are several advantages in such a decomposition viz. *a. it provides a step-by-step process to identify and represent all the transfers of control that exist in the system b. it ensures that no transfer of control is repeated; hence none of the transfers of control is tested more than once c. it provides a disjoint list of transfers of control thus facilitating simultaneous testing of different transfers of control d. it provides systematically the order in which the tests need to be carried out.*

4. VDM Specification for the Decomposition Algorithm

Benefits of using VDM in different areas of software engineering are well documented. Studies on the suitability and benefits of applying VDM for specific activities have been carried out, illustrating the use of VDM in different stages of software development [5,22]. Using VDM specifications, Nadeem and Ur-Rehman [17] propose an approach to automated testing that generates the required C code as well as test data. In another study [8], Fenkam et al. report a technique for constructing a CORBA-supported VDM oracle for automated testing starting from a VDM specification.

In this section, we develop VDM specification for the decomposition algorithm presented in Section 2. It demonstrates how a VDM specification represents an algorithm rigorously without ambiguity or inconsistency.

4.1 Terminology and Notations

We begin with a discussion of the data structures required for the specification of the algorithm.

Length of the path is the number of edges in the path, denoted by *len path*. The first element of a path is called its *head* and the rest of the path is called its *tail*. In VDM, head and tail of a path are indicated *hd path* and *tl path* respectively, whereas the element in the ith position of the path is denoted by *path(i)*. If *A* is a set, then *A-set* denotes the power set of *A* and A^* denotes the set of all sequences of the elements of *A* (including the empty sequence), where repetitions are allowed.

More precisely, $A^* = ø \cup A \cup A^2 \cup A^3 \cup \ldots = \bigcup_{i=0}^{\infty} A^i$, where

$A^i = A \times A \times A \times \ldots \times A$, the product taken i times with $A^0 = ø$.

In data structures, a variable is also referred to as an *object*. Sometimes an object may contain in itself other objects. Such an object is called a *composite object* which is similar to the records of Pascal or the structures of C. More precisely, a composite object has a number of fields, each such field is a variable.

A *make function*, when applied to appropriate values for the fields, yields a value of the composite type. A *make function* is specific to a type; its name is formed by prefixing *mk-* to the name of the type. Thus if *D* is a composite object, with fields f_1, f_2, … ,f_n, then $mk\text{-}D: f_1 \times f_2 \times \ldots \times f_n \rightarrow D$.

Often we may have to impose some restrictions on the objects. In formal specifications, such restrictions are known as *invariants*. They are written as a part of the type definition with key word (*inv*) followed by the character _.

A precise statement of all external characteristics of a system used is called *functional specification* or *implicit definition*. A *direct definition* of a function provides a rule for computing the result of applying the function to its arguments. The *pre-condition* of a function is a truth-valued function, which defines the elements of the domain of a partial function (operation) for which the existence of a result is guaranteed. The pre-condition of a function or operation defines the state/inputs to which it can be applied. The post-condition is a truth-valued function, which defines the required relation between the input and output.

There are several ways of decomposing a composite object. One method uses the name of the *make* function in a context which makes it possible to associate names with the variables. Alternatively, one can use local variables which are introduced with the *let* command.

4.2 Two Composite Objects

Formal specification of the decomposition algorithm begins with the definition of two composite objects viz. *Directed graph* and *Restricted graph*.

The idea of a graph as a pair comprising a set of nodes and a set of edges is brought out in the following definition of the composite object *Graph*:

Graph:: nodes: Z-set

    edges: $(Z \times Z)$*-set*

inv_Graph(mk-Graph(n,e)) $\underline{\Delta}\; e \subseteq n \times n$

The labels of the nodes have been assumed to be integers – negative as well as positive. The invariant stipulates that the nodes connected by the edges of the graph are only those from the given set of nodes. For example, ({a,b,c},{<a,b>,<b,c>,<c,d>}) will not be accepted as a graph since <c,d>Ï n × n, where n ={a, b, c}. We note that the graph, as defined here, is actually a directed graph.

A restricted graph is a directed graph with the additional restrictions that there is exactly one initial node as well as exactly one final node. Hence the following definition of the composite object *RGraph*:

RGraph::

    nodes: Z-set

    edges:: $(Z \times Z)$*-set*

inv_Rgraph(mk-Rgraph(n,e)) $\underline{\Delta}$

$$e \subseteq n \times n \wedge$$

$$\textit{is-unique-initial}(g) \wedge$$

$$\textit{is-unique-final}\ (g)$$

The boolean functions testing the uniqueness of the initial and final nodes are developed in the next section.

4.3 Functions: Uniqueness of Initial and Final Nodes

The process of checking whether the initial node is unique is carried out in two stages. We first collect all the initial nodes, count them, and make sure that the count is one. However, the collection of initials assumes that there is an initial node. For this reason we have the following specification:

exists-initial(g: RGraph) t : B

pre $g \neq \emptyset$

post *let mk-RGraph (n,e)=g in*

$$t \equiv \exists k \in n \cdot (\exists j \in n \cdot (k, j) \in e \wedge$$

$$\neg \exists r \in n \cdot (r, k) \in e\ )$$

The function *exists-initial* returns the value *true* if there is at least one initial node. Otherwise it returns *false.* The post-condition is a predicate expression for the mathematical definition of an initial node, stated as follows: *a node k is an initial node if and only if there is an edge going out of it and there is no edge coming into it.* This boolean valued function provides the pre-condition for the function that collects all the initial nodes given below:

initials(g: RGraph) s: Z-set

pre *exists-initial(g)*

post *let mk-RGraph(n,e)=g in*

$$\forall k \in s \cdot (\$\, j \in n \cdot (k, j) \in e \wedge$$

$$\neg \exists r \in n \cdot (r, k) \in e\ )$$

The definition of an initial node is used again to verify that every element of the output set is an initial node. We are now in a position to specify the function that examines the uniqueness of the initial node:

is-unique-initial(g: RGraph) r : B

pre *exists-initial(g)*

post $r \equiv (\text{card initials}(g) = 1\ )$

The post-condition is made simpler by the use of the pre-defined operator *card* that stands for the *cardinality* of the set.

The function *is-unique-final* and the associated functions can be specified in a similar manner as follows:

exists-final(g: RGraph) t: B

pre $g \neq \emptyset$

post *let mk-RGraph(n,e)=g in*

$$t \equiv \exists k \in n \cdot (\exists j \in n \cdot (j, k) \in e \wedge$$

$$\neg \exists r \in n \cdot (k, r) \in e\ )$$

finals(g: RGraph) s: Z-set

pre *exists-final(g)*

post *let mk-RGraph(n,e)=g in*

$$\forall k \in s \cdot (\exists j \in n \cdot (j, k) \in e \wedge$$

$$\neg \exists r \in n \cdot (k, r) \in e\ )$$

is-unique-final(g: RGraph) r : B

pre *exists-final(g)*

post $r \equiv (\ \text{card finals}(g) = 1\ )$

The details are similar as in the case of testing for the uniqueness of initial node. For the identification of the final node, the post-condition in the *exists-final* function stipulates that there exists one node k such that there is an edge coming into it and there is no edge going out of it.

4.4 Functions: Path Extraction

As seen in Section 2, extraction of paths plays a key role in the decomposition algorithm. VDM provides ways of manipulating such paths. We can specify a function that extracts a path. Once a path is extracted, there is a need to delete the edges included in the extracted path. As a result of such a deletion some nodes may become isolated and the structure of the restricted graph may also be violated. To convert the resulting graph into a restricted graph, *dummy nodes* and *dummy edges* are introduced.

Suppose, for example, there occur two initial nodes, say *m* and *n*. A dummy node *k* is introduced at this stage and two dummy edges (*k*, *m*) and (*k*, *n*) are created. To distinguish the dummy nodes from the real nodes, negative integers are used to label the former. Multiple final nodes are handled in the same manner.

Once the path is extracted, the dummy nodes, if any, are removed from the path. We use the path number to label the dummy node consistently. If the graph used for the extraction of k^{th} path has multiple initials, a dummy initial node with label *–(2k –1*) is added; in case the path has multiple finals, the dummy final node added will have the label *–2k*. The process then continues with the extraction of the next path.

We shall specify the function that extracts a path, starting from the initial node of the restricted graph until either the final node of the graph or a node already included is encountered. The function also needs to delete the associated edges included in the extracted path.

Central to the process of decomposition is the function *create (k, g)*. With the path number and the restricted graph as the input parameters, it specifies the conditions to be satisfied by the k^{th} path. We shall now give the specification for *create (k, g)*.

```
create(k : $N_1$, g : RGraph) path : $Z^*$
pre      g $\neq \varnothing$
post ( hd path $\in$ initials (g)
        hd path = - (2*k-1) ) $\wedge$
        let m = len path in
        $\forall$i $\in${1,2,…,m}.
     <path(i),path(i+1) > $\in$ edges (g) $\wedge$
     (path (m+1) $\in$ finals (g)
         j $\in${1,2,…, m}. path(m+1) =
         path(j) path(m+1) = -2*k)
```

It is worth noting that *g* represents the graph after the extraction of k–1 paths.

The process is to be terminated when *g* becomes empty. Hence the pre-condition. The post condition ensures the following: *a. first node of the path is either the initial node or a dummy node b.consecutive nodes of the path correspond to some edge of the graph and c.the last node of the path is the final node or a node already included in the path or a dummy final node.*

Clearly, extraction of paths requires dummy nodes, which, in turn, depend on the path number. Thus the process of decomposition, defined by the function *decompose (g)*, passes the path number 1 along with the graph to the function *decomp (m, g)*, which is executed recursively.

```
decompose: RGraph $\rightarrow$ $N_1^*$-set
decompose (g ) $\underline{\Delta}$
                decomp (1,g)
decomp: $N_1$ × Graph $\rightarrow$ $N_1^*$-set
decomp (m,g ) $\underline{\Delta}$
                if (g = ø )
                then ø
                else
                    let p = create (m,g ) $\wedge$
                    path = remove-dummy (p) $\wedge$
                    $g_1$ = update-Rgraph (g, p) $\wedge$
                    $g_2$ = merge-initials (m, $g_1$ ) $\wedge$
                    $g_3$ = merge-finals (m, $g_2$ ) in
                    {path}$\cup$ decomp (m+1,$g_3$)
```

The function decomp is executed, as long as the graph is non-empty. It performs five sub-tasks: *a. extracts the m^{th} path p b.removes dummy initials and/or dummy finals, if any c.updates the graph by removing the edges of p as well as the resulting isolated nodes from g d.merges the initial nodes, in the event of multiple initials e. merges the final nodes, in the event of multiple finals*.

Functions have been defined to carry out each of these subtasks. Before we discuss them, it is necessary to note the data type of the input g coming into the function *decomp*. While g is a restricted graph initially, introduction of dummy nodes produces a graph. Thus g is taken to be a graph. Also, since the dummy nodes are removed before the path is added to the decomposition, the output set is still of type *–set.*

```
remove-dummy: $Z^*$ $\rightarrow$ $N_1^*$
remove-dummy(p) $\underline{\Delta}$ let
        $p_1$ = remove-dummy-initial(p) $\wedge$
        $p_2$ = remove-dummy-final($p_1$) in
        $p_2$
remove-dummy-initial: $Z^*$ $\rightarrow$ $Z^*$
remove-dummy-initial(p) $\underline{\Delta}$
        if (hd p > 0 )
        then p
        else tl p
remove-dummy-final: $Z^*$ $\rightarrow$ $N_1^*$
remove-dummy-final(p) $\underline{\Delta}$
        let m= len p in
        if ( p(m+1) > 0 )
        then p
        else subseq (p ,1, m)
```

Removal of the dummy initial is straightforward. If the first node has a positive label, the path received is returned without any change; otherwise the rest of the path is returned. Removal of the dummy final involves deleting the last node. This is achieved by the standard VDM function for subsequence.

After the removal of the dummy initial, the path may still have a negative node as a final. Hence the output of the function *remove-dummy-initial* is a sequence of integers. However, the removal of the dummy final will result in a sequence of positive integers, which is the reason for the output of the functions *remove-dummy* and *remove-dummy-final* to be N_1^*.

Once the dummy nodes are removed from the path extracted, the path can be added to the output of *decomp* function.

4.5 Functions: Updating the Original Graph

Now we turn our attention to the problem of updating the graph by removing the edges of the path just extracted and the resulting isolated nodes.

```
update-RGraph: RGraph × $N_1^*$ $\rightarrow$ Graph
update-RGraph (g ,p) $\underline{\Delta}$ let
        e = edges(g) $\wedge$
        n = nodes(g) $\wedge$
        $e_1$ = new-edges(e,p) $\wedge$
        $n_1$ = n - isolated-nodes(n,$e_1$) $\wedge$
        $g_1$ = mk-RGraph ($n_1$ ,$e_1$) in
        $g_1$
```

Note that the selector functions *edges* and *nodes* are used to decompose g. It is equivalent to *mk-RGraph (n ,e) = g*.

The function *new-edges (e,p)* is invoked to remove the edges included in the path *p*. The set of all nodes that become isolated as a result of such a removal are returned by the function *isolated-nodes (n,e).*

```
new-edges: ($N_1$× $N_1$ )-set x $N_1^*$ $\rightarrow$
                                ($N_1$× $N_1$ )-set
new-edges(e, p) $\underline{\Delta}$
                if len p =0
                then e
                else let
                $e_1$ = e - < p(1 ), p(2 ) > $\wedge$
                $p_1$ = tl p in
            new-edges($e_1$, $p_1$ )
```

The function is defined recursively. We remove the first edge of the path p *from the edge set and the process is repeated with the tail of* p *until there is no edge for deletion.*

```
isolated-nodes: $N_1$-set × ($N_1$ × $N_1$)-set $\rightarrow$
                                        $N_1$-set
isolated-nodes(n,e) $\underline{\Delta}$
            if ( n = { } )
            then { }
            let x $\in$n in
            elsif is-isolated-node(x, e)
            then {x} $\cup$ isolated-nodes(n-{x},e)
            else isolated-nodes(n-{x},e)
is-isolated-node: $N_1$ × ($N_1$× $N_1$)-set $\rightarrow$ **B**
is-isolated-node(x,e) $\underline{\Delta}$
            if ( e = [ ] )
            then true
               let <a, b> $\in$e in
            elsif x = a x = b
            then false
            else is-isolated-node(x, e-< a ,b>)
```

We again have two recursive functions. An arbitrary node of the node set *n* is tested to see whether it is isolated. If so, it is included along with the other such nodes to be collected from the rest of the nodes in *n*; otherwise the process continues with the selection of another arbitrary node from *n*. It is repeated until there is no more node in *n*. The boolean function *is-isolated-node (x, e)* is used to test whether a node *x* is isolated by selecting an edge from *e* arbitrarily. If either end of the edge is the same as *x*, then *x* is not isolated and the testing ends; otherwise another edge is selected arbitrarily and the test is repeated. If there is no more edge to be tried, then *x* is isolated and the testing terminates.

With this, the third sub-task of the *decompose( )* function is completed. The problem of multiple initials and multiple finals remain to be tackled.

```
merge-initials : $N_1$ × Graph $\rightarrow$ Graph
merge-initials (m, g )
if is-unique-initial(g)
then g
        let x = initials(g ) $\wedge$
          $m_1$ = - (2*m-1) $\wedge$
          $n_1$ = { $m_1$ } $\cup$ nodes(g) $\wedge$
          $e_1$ = add-initial-edges( $m_1$, x ) $\cup$
                                  edges(e) $\wedge$
           $g_1$ = mk-RGraph($n_1$ ,$e_1$) in
else $g_1$
```

If there is only one initial, nothing needs to be done. If not, all the initials are collected first. A dummy node with label -(2m–1) is created and included in the set of nodes. To the set of edges, we add new initial edges connecting this node with all the initial nodes. The new initial edges are created as follows:

```
add-initial-edges: Z × $N_1$-set $\rightarrow$
                                        (Z × $N_1$)-set
add-initial-edges(k, x) $\underline{\Delta}$
if (x = { } )
then { }
 let a $\in$x in
else <k,a> $\cup$ add-initial-edges (k, x-{a})
```

The dummy initial k is connected to each of the initial nodes which are selected arbitrarily one by one. The resulting edges are collected recursively.

```
merge-finals: $N_1$ × Graph $\rightarrow$ Graph
merge-finals(m, g ) $\underline{\Delta}$
if is-unique-final(g)
then g
        let x = finals(g ) $\wedge$
          $m_1$ = - 2*m $\wedge$
          $n_1$ = { $m_1$ } $\cup$ nodes(g) $\wedge$
$e_1$ = add-final-edges( $m_1$ x )$\cup$ edges(g) $\wedge$
$g_1$ = mk-RGraph($n_1$ ,$e_1$) in
else $g_1$
```

```
add-final-edges: $Z \times N_1$-set $\rightarrow$ $(Z \times N_1)$-set
add-final-edges(k,x) $\triangleq$
        if (x = ø )
        then ø
          let a $\in$ x in
        else
          <a, k> È add-final-edges(k, x-{a})
```

The two functions defined above work the same way as the corresponding functions for merging initials. This completes the VDM specifications and definitions of the decomposition algorithm.

5. Conclusion

In this paper we have developed an algorithm for decomposing a graph into disjoint paths and represented the process in VDM. Our primary aim is to demonstrate the use of VDM in representing a process without ambiguity. Two composite objects, required for the VDM specification, have been defined and discussed. Necessary function specifications and definitions have been developed.

Representation of the algorithm using VDM has brought in rigour and robustness, facilitating easy translation of the algorithm into a programming language.

Acknowledgements

The authors would like to thank Middle East College of Information Technology, Sultanate of Oman for the support provided in making this research possible and the referees for their valuable comments and suggestions.

References

[1] Alagar, V.S, Periyasamy, K.,Ramanathan, K (1994). Formal specification techniques for complex software systems*, In*: Proc. Of TENCON 1994. IEEE Region 10's Ninth Annual International Conf., V. 2 p. 1008 – 1013.

[2] Bacherini, S., Fantechi, A., Tempestini,M., Zingoni,N (2006). A Story About Formal Methods Adoption by a Railway Signaling Manufacturer. Lecture Notes in Computer Science, FM 2006: Formal Methods, Springer Berlin / Heidelberg, Volume 4085/ 20. p.179-189

[3] Beizer, B (1990). Software Testing Techniques. London: Int'l Thompson Computer Press.

[4] Bertolino, A.,Mirandola, R., Peciola, E (1997). A Case Study in Branch Testing Automation, Special issue on achieving quality in software. *Journal of Systems and Software*, 38 (1) 1997. 47-59.

[5] Bjorner, D (1987). On the use of formal methods in software development, *In*: Proc. of the 9th international conference on Software Engineering, p.17-29.

[6] Chatzigeorgiou, A.,Tsantalis, N., Stephanides, G (2006). Application of graph theory to OO software engineering, *In*: Proceedings of the 2006 international workshop on interdisciplinary software engineering research, Shanghai, China p.29 - 36.

[7] Ciapessoni, E.,Mirandola, P., Coen-Porisini, A.,Mandrioli, D.,Morzenti, A (1999).From formal models to formally based methods: an industrial experience, *ACM Transactions on Software Engineering and Methodology*, 8 (1) 79-113.

[8] Fenkam, P., Gall, H., Jazayeri, M (2002). Constructing CORBAsupported oracles for testing: a case study in automated

software testing, *In*: Proc. of 17th IEEE International Conference on Automated Software Engineering, p.129-138.

[9] Gerhart, S., Craigen, D., Ralston, T (1994). Experience with Formal methods in Critical Systems, *IEEE Software*, p.21-28.

[10] Hinchey, M.G., Bowen, J.P (1995). Applications of Formal Methods.Prentice-Hall, Englewood Cliffs, N.J.

[11] Hui, J., Dong, L., Xiren, X (1997).Using formal specification language in industrial software development, *In*: Proc. of the IEEE International Conference on Intelligent Processing Systems, 1997, V. 2 p.1847 – 1851.

[12] Jones, C.B (1986). Systematic Software Development Using VDM. London: Prentice Hall International (UK) Ltd.

[13] Kit,E (1988). Software Testing in the Real world: improving the process. Addison Wesley.

[14] Larsen, P.G., Fitzgerald, J., Brookes, T (1996). Applying formal specification in industry, *IEEE Software*, 13 (3) 48 – 56.

[15] McCabe, T. J (1976). A Complexity Measure, *IEEE Trans. Software Eng.*,2 (4) 308-320.

[16] Meyer, B (1985). On Formalism in Specification, IEEE Software, 6-26.

[17] Nadeem, A., Ur-Rehman, M.J (2004). A framework for automated testing from VDM-SL specifications, *In*: Proc. of 8th International Multitopic Conference, Dec.2004, p.428- 433.

[18] Ntafos, S.C.,Hakimi, S.L (1979).On path Cover Problems in Digraphs and Applications to Program Testing, *In*: IEEE Transactions on Software Engineering, 5 (5) 520-529.

[19] Palshikar, G.K (2001). Applying Formal Specifications to Real-World Software Development, *IEEE Software*, 18 (6) 89-97.

[20] Pedersen, J.S., Klein, M.H (1988). Using the Vienna Development Method(VDM) to formalize a communications protocol, Technical Report CMU/SEI-88-TR-26,ESD-TR-88-027, Software Eng.Res. Inst.,Carnegie Mellon Univ., Pittsburgh, PA.

[21] Pfleeger, S.L.,Hatton,L (1997).Investigating the Influence of Formal Methods, *IEEE Computer*, 30 (2)33-43.

[22] Plat, N., Van Katwijk, J., Toetenel, H (1992). Application and benefits of formal methods in software development, *Software Engineering Journal*, 7 (5) 335 – 346.

[23] Stickney, M.E (1978). An application of graph theory to software test data selection, *In*: Proc. of the software quality assurance workshop on Functional and performance issues, p.111 -115.

[24] Williams, L.G (1994) Assessment of safety-critical specifications, *IEEE Software*, 11 (1) 51 – 60.

Abdul Huq received his PhD from the University of Madras, India. He is currently Head of the Department of Computing, Mathematics and Applied Sciences at Middle East College of Information Technology, Sultanate of Oman. His research interests include Artificial Intelligence, Software Engineering and Syntactic Pattern Recognition. He has published several papers and authored books on Computing.

Narayanan T Ramachandran received his PhD from the University of Madras, India. He is currently Dean of Middle East College of Information Technology, Sultanate of Oman. His research interests include Simulation studies, Optimisation techniques and Software Engineering. He has published several papers and authored books on Computing.