Specification Representation and Automatic Test Case Generation using System Model

Ashish Kumari, Noor Mohammad, Chetna

Abstract: Finite State Machine is used to model the requirement specification of the system by formal description languages. In this paper, I have presented a approach which is used to represent the requirement specification and automatically generate all possible test cases which should be executed to test that particular system [5].Requirement specification are represented using extended finte state machine which uses the state transition diagram that shows how system changes states and action and variable used during each transition. Based on information given in the state transition diagram, all possible test cases are generating by traversing the graph using Depth First Search.

KEYWORDS: Regression testing, extended finite state machine, Specification-based testing, State Transitions, path, Data dependency, Control dependency, SDG

I. INTORDUCTION

Specification-based testing confers a number of advantages to the software development process.

A specification provides an exact description of the software's fundamental aspects while excluding more detailed information. This allows a tester to extract the product's basic functionality without wading through inessential details. By deriving tests from the software specification, tests can be produced before the software itself. Since many faults occur during the design phase, early identification of them can reduce total development times and costs [9].

In addition, developing tests forces a detailed look at the specification itself, which may reveal ambiguities and/or inconsistencies. These can then be fixed early in the development cycle at a minimum of cost.

Extended Finite-state machines are comprised of states, transitions, events and actions that emphasize the flow of control from one state to another. Finite-state machines best describe the dynamic behavior of a system, and finite- state model based testing has been studied extensively.

EFSM can be used to represent the behavior of communication protocols, graphical user interfaces and other event-driven systems. Model-based testing approaches can automatically derive executable tests from system model thus providing benefits like systematic testing and test adequacy. Since a key requirement of software testing is to ensure test adequacy, these features make finite state machine based testing very useful.

II. SPECIFICATION REPRESENTATION USING EXTENDED FINITE STATE MACHINE [1]

Manuscript received on July, 2012.

Ashish Kumari, M.Tech (Scholar) S.E.C, Jhunjhunu, RTU, Kota(Rajasthan),India

Noor Mohammad, Assistant. Professor S.E.C, Jhunjhunu, RTU, Kota(Rajasthan),India

Chetna, M.Tech (Scholar) Jagannath University, Jaipur (Rajasthan),India

An **EFSM** is a 5-tuple <*S*, *I*, *O.V*, *T*> where:

- *S* is a nonempty finite set of states with two states designated as *Start* and *Exit* states of the EFSM
- *I* is a nonempty finite set of input interactions, each with a (possibly empty) set of input interaction parameters
- *O* is a nonempty finite set of output interactions, each with a (possibly empty) set of output interaction parameter
- *V* is the nonempty finite set of all variables which is the union of set of all local variables and set of all interaction parameters
- *T* is a nonempty finite set of transitions

Each transition *t of* T is a 6-tuple <s_s, s_t, i, c, o, *a*> where:

- s_s , $s_t \in S$ are the *starting* and *terminating* states of *t*
- $i \in I$ is the input interaction of *t*
- ***c*** is the *enabling condition* of *t* which is a Boolean expression defined over the set of all local variables and set of all input interaction parameters
- $o \in O$ is the output interaction of *t*
- ***a*** is a sequence of actions of *t* expressed as functions *f*: $V \rightarrow V$

EFSM models are graphically represented as graphs where states are represented as nodes and transitions as directed edges between states.

The following elements are associateowith each transition :-
(1) Event
(2) Condition and
(3) Sequence of actions

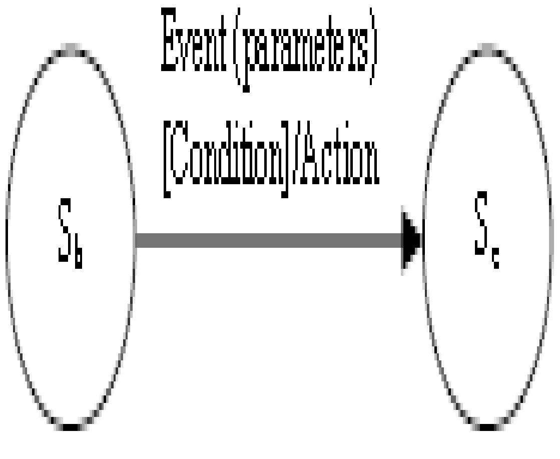

Figure 1 Graphical Representation of an EFSM Transition [1, 2, 11]

In a given EFSM model, it is assumed that every state is reachable from *Start* and *Exit* is reachable from every state. In an EFSM model of a simplified ATM system was given. This EFSM, which is shown in Figure 1, will be used as a running example throughout this paper. Note that we are interested in modifications on transitions instead of states, since transitions represent active elements of the EFSM model.

III. REQUIREMENT ECIFICATION FOR ATM MACHINE

Finite State Machine model to specify the following ATM transaction behavior is given in Figure2. Once a card is inserted and PIN validated, the transactions *deposit*, *withdrawal* may be carried out. If invalid PIN entries are made, there is a limit to the number of re-entries. To generate executable tests from the state transition diagram provided in Figure 1, the events PIN, Withdrawal Amount and ContinueTransaction are modeled as data-flow graphs as shown in figure1. Each time a deposit or withdraw is made into the account, the variable *Balance* is updated

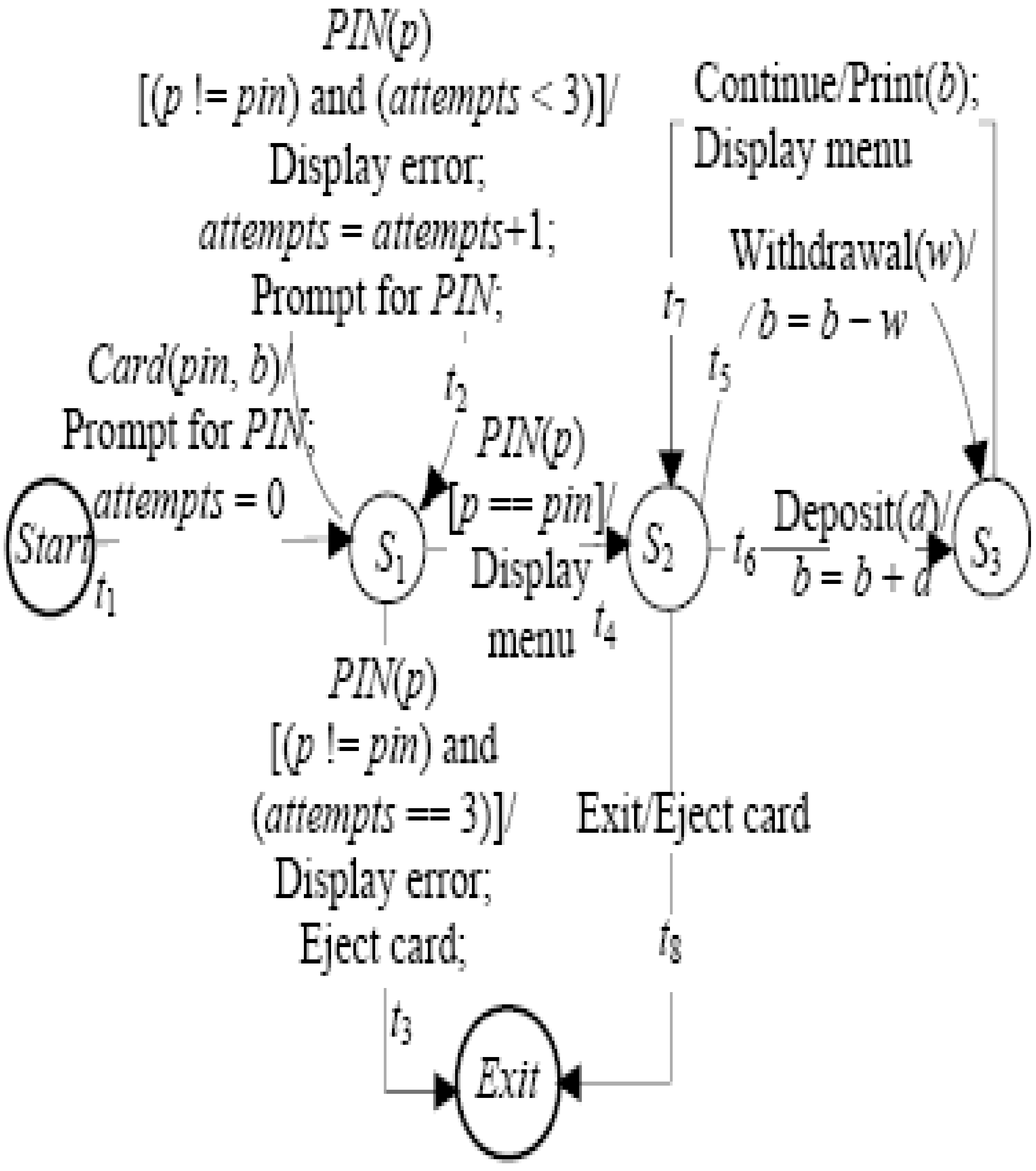
PIN(p) [(p != pin) and (attempts < 3)]/ Display error; attempts = attempts+1; Prompt for PIN;
Continue/Print(b); Display menu
Withdrawal(w)/ /b = b − w
Card(pin, b)/ Prompt for PIN; attempts = 0
t1
t2
t3
t4
t5
t6
t7
t8
Start
S1
S2
S3
PIN(p) [p == pin]/ Display menu
Deposit(d)/ b = b + d
PIN(p) [(p != pin) and (attempts == 3)]/ Display error; Eject card;
Exit/Eject card
Exit

Figure 2 State Transition Diagram for ATM [1]

Details of each transition in an ATM system are given below:

Transition T1:
Variable defined: attempt
Varibale initialized: pin, balance
Variable used: none
Event: card is inserted
Condition: none
Action: prompt for pin

Transition T2:

Variable defined: attempt

Varibale initialized: p
Variable used: p, pin, attempt
Event: pin_no is entered
Condition: pin_no! =p & attempt<=3
Action: if condition is true

1. Display error
2. Attempt=attempt+1
3. Prompt for pin

Transition T3:

Variable defined: none
Varibale initialized: p
Variable used: p, pin, attempt
Event: pin_no is entered
Condition: pin_no! =p & attempt==3
Action: if condition is true

1. Display error
2. Eject card

Transition T4:

Variable defined: none
Varibale initialized: p
Variable used: pin
Event: pin_no is entered
Condition: pin_no==p & attempt<=3
Action: if condition is true

1. Display menu

Transition T5:

Variable defined: balance
Varibale initialized: none
Variabl_used: balance,withdraw_amount
Event: withdraw button is pressed
Condition: amount<= balance
Action: if condition is true

1. balance-=withdraw_amt

Else
Display error

Transition T6:

Variable defined: balance
Varibale initialized: none
Variabl_used: balance, deposit_amount
Event: deposit button is pressed
Condition: none
Action: balance+=deposit_amt

Transition T7:

Variable defined: none
Varibale initialized: none
Variable_used: none
Event: none
Action: display menu

Transition T8:

Variable defined: none
Varibale initialized: none
Variable_used: none
Event: none
Action: eject card

IV. AUTOMATIC TEST CASE GENERATION

An EFSM system model becomes an input to an EFSM test generator that may support a variety of the existing EFSM model-based test generation strategies.Depending on the selected testing strategy, the generator automatically generates a set of tests (paths an initial state to the final state) in the EFSM model satisfies the selected strategy. For each path, appropriate test values (inputs) that lead to the traversal of the selected path are identified. Clearly, a test case consist sequence of events (transitions) with appropriate input values [2, 10]. The following is an example of a test case for ATM system shown of Figure 2 [2]:

Card (1234, 100.00); PIN (1234); Withdrawl (50); Receipt; Exit.

Therefore, the test shown above is represented as the following sequence of transitions:

TI, T4, T5, T7, T8.

Most of the existing EFSM model-based test generation strategies are mainly used to test the whole system, referred to as complete system testing. Several testing strategies exist, e.g., transition coverage, path coverage, and constrained path coverage [2][4][10]:

A.Transition coverage strategy:

This requires that every transition in the model be traversed at least once.

B. Path coverage strategy:

This requires that every path in the model be traversed at least once; this strategy is frequently not practical because of an unacceptable number of test cases generated in the presence of cycles in the model.

C.Modified path strategy /constrained path strategy:

This strategy limits the test explosion by limiting a number of times each transition can be traversed. This strategy requires that every path in the model be traversed at least once where each path can contain at most n "occurrences" of the same transition (any transition can be traversed at most n times in a path).

EFSM having the two types of dependencies between the transition nodes:

1. Data dependency
2. Control dependency

A **data dependency**[6] between two transition Ti and Tk w.r.t. variable v , if Ti transition defined variable v and transition Tk use the same variable v and a path exists between these two transitions Ti and Tk in EFSM model along which variable v is not modified.

Control dependence

Means that one transition may affect the traversal of another transition. Control dependence between transitions is defined in terms of the concept of *post-dominance.* Suppose that S1 and S2 are two distinct states, and *t* is an outgoing transition from S1 in an EFSM. Then, S2 post-dominates S1 if and only if S2 is on every path from S1 to exit and S2 post -dominates *t* if and only if S2 is on every path from S1 to exit through *t.*

State coverage, Transition coverage, Path coverage, constrained path coverage [1, 2, 4] are the various technique which are used to generate the test cases in EFSM with the help of dependency. Data dependencies and control dependencies between the transitions are the main elements which serve as the input for designing the Static Dependency Graph (SDG).

EFSM system model is already shown in Figure1. And constrained path strategy is frequently used testing strategy in model-based testing.

A tester decides to generate a complete system test suite using a constrained path coverage testing strategy. The resulting complete system test suite contains 64 tests for n = 3; for n = 4, the complete system test suite size is 160. In this paper we are implementing the modified path strategy or constrained path coverage technique.

V. ALGORITHIM FOR FINDING DATA DEPENDENCY AND CONTROL DEPENDENCY

```
Data_ dependency (Struct transition T1, Struct transition T2)
Begin
    • For each defined variable d in T1
    • For each used variable k in T2
    • If (d==k)
        • For each path p from T1 to T2
        • Search for a transition in path p which is defining the variable
        • If (! found) ,Then
            • Display "Transition T1 has data dependency with transition T2 for variable d"
            • Insert the entry into data dependency matrix.
            • Go to first step.
  End
```

```
Post_ dominate (struct transition T1, struct transition T2)
Begin
If (! post_ state (T2.start, T1.start) and post_ transition (T2.start, T1))
• Display "transition T1 post dominate transition T2"
• Insert the entry for control dependency into dependency matrix
    Else
• Display "transition T1 does not post dominate transition T2"
    End
```

The algorithm for finding out whether T2.Start post dominates T1.start:

```
Post_ state (State S2, State S1)
Begin
    1. Find out all the transitions t1,t2 …….tn whose starting state is S1
    2. For each transition t in set t1,t2,…tn
       2.1 For each the possible path
           for transition t to exit state
           2.1.1 Search for the
                 transition in path
           2.1.2 If! (Found)
                 Then
                 Set post_ dominate=0
                 and go to step3.
    3. If post_ dominate=0
       Then
            Display "state S2 post dominate state S1"
                 Return 1
       Else
            Display "state S2 does not post dominate state S1"
            Return 0;
End
```

The algorithm for finding out whether T2.start post dominates transition T1:

```
post_ transition(State S1, Transition T1)
Begin
    1. Find out all the destination state S of T1.
    2. Find out all the transition t1, t2 ….tn whose starting state S.
    3. For each transition t in set t1,t2…..tn
       3.1 For each possible path P from transition t to exit state
           3.1.1 Search for the transition whose starting state is S1
           3.1.2 If (!found)
                 Then
                 Set Post_ dominate=0and go to step 4.
    4. If post_ dominate=0
       Then
            Display "state S2 post
            dominate state S1"
            Return 1
       Else
            Display "state S2 does
            not post dominate
            state S1"

            Return 0;
End
```

VI. ALGORITHIM FOR GENERATING ALL POSSIBLE PATH/TEST CASES USING CONSTRAINED PATH STRATEGY

The algorithim define below based on depth first search and gives details to findout the possible path or test cases from a finite state machine represented using state transition diagram. The algorithim path_generate is invoked on the start state of the finite state machine.

Transition of the ATM state transition diagram contains the following details [12]:

```
Struct transition
{
int no;
int source;
int dest;
int no_of_variable_used;
char *action[no_of_action];
char *events[no_of _events];
char *var[no_of _var_used];
int accurance;
}
```

```
Algorithim:
Path_generation (struct transition T1)
{
If (T1.occurance<n)
Insert this particular transition T1 into the stack.
T1.occurance+=1; /* Transition would be traversed upto n times in path when there is cycle to avoid infinite no. of possible test cases. */
```

```
 If (T1.dest==exit_state)
 Display the contents of the stack array from 0 to top of the stack.
 /*which consist of the sequence of transition traversed in executing this particular path or test case.*/
 Else
 Repeat the step for all adjacent transition T to transition T1.
/* adjacent transitions are those whose source state is same as    destination state of T1.*/
Path_generation (T)
            /*call path_generation for the next adjacent transition to T1*/
 Pop ();
 /* pop out the last transition from the stack. This algo is based on depth first   search so after finding out the one path the. We backtrack to second last node of the path and findout another node who is adjacent. */
 }
```

Test Cases generated for ATM machine when n=3

This section contains all test cases or paths which are generated by the algorithm path _ generate. Path contains only the sequence of the transition which is traversed during the execution of that particular path or test cases. As this will find out all possible path so condition associated with the transition is evaluated as true and false.

Here n=3 means that atmost three times a transaction can be traversed in a path if there is cycle encounter in the path among the transition.

As we know “start” is starting state for STD and “exit” is exit state. So every test case transition whose source state is “start” and ends with transition whose destination state is the “exit state.”

Now the following test cases like:

Test case - T1T2T4T8

T1 (CARD INSERTED) **T2** (WRONG PIN ENTERED) **T4** (CORRECT PIN ENTERED) **T8** (EXIT WITHOUT ANY TRANSACTION**).**

Here I have write without any transaction bcoz T8 is selected just after the T4.

Test case-T1 T2 T2 T4 T6 T7 T8

T1 (CARD INSERTED**) T2 (**WRONG PIN ENTERED) **T2** (AGAIN WRONG PIN ENTERED) **T4** (CORRECT PIN ENTERED) **T6** (3000rs DEPOSITED) **T7** (RECEIPT) **T8** (EXIT).

Using this way we can expand all possible test cases given below.

List of Test cases or all possible paths in ATM’s state transition diagram:

1 T1 T3
2 T1 T2 T3
3 T1 T4 T8
4 T1 T2 T2 T3
5 T1 T2 T4 T8
6 T1 T2 T2 T2 T3
7 T1 T2 T2 T4 T8
8 T1 T4 T5 T7 T8
9 T1 T4 T6 T7 T8
10 T1 T2 T2 T2 T4 T8
11 T1 T2 T4 T5 T7 T8
12 T1 T2 T4 T6 T7 T8
13 T1 T2 T2 T4 T5 T7 T8
14 T1 T2 T2 T4 T6 T7 T8
15 T1 T4 T5 T7 T5 T7 T8
16 T1 T4 T5 T7 T6 T7 T8
17 T1 T4 T6 T7 T5 T7 T8
18 T1 T4 T6 T7 T6 T7 T8
19 T1 T2 T2 T2 T4 T5 T7 T8
20 T1 T2 T2 T2 T4 T6 T7 T8
21 T1 T2 T4 T5 T7 T5 T7 T8
22 T1 T2 T4 T5 T7 T6 T7 T8
23 T1 T2 T4 T6 T7 T5 T7 T8
24 T1 T2 T4 T6 T7 T6 T7 T8
25 T1 T2 T2 T4 T5 T7 T5 T7 T8
26 T1 T2 T2 T4 T5 T7 T6 T7 T8
27 T1 T2 T2 T4 T6 T7 T5 T7 T8
28 T1 T2 T2 T4 T6 T7 T6 T7 T8
29 T1 T4 T5 T7 T5 T7 T5 T7 T8
30 T1 T4 T5 T7 T5 T7 T6 T7 T8
31 T1 T4 T5 T7 T6 T7 T5 T7 T8
32 T1 T4 T5 T7 T6 T7 T6 T7 T8
33 T1 T4 T6 T7 T5 T7 T5 T7 T8
34 T1 T4 T6 T7 T5 T7 T6 T7 T8
35 T1 T4 T6 T7 T6 T7 T5 T7 T8
36 T1 T4 T6 T7 T6 T7 T6 T7 T8
37 T1 T2 T2 T2 T4 T5T7 T5T7T8
38 T1 T2 T2 T2 T4 T5 T7 T6 T7 T8
39 T1 T2 T2 T2 T4 T6 T7 T5 T7 T8
40 T1 T2 T2 T2 T4 T6 T7 T6 T7 T8
41 T1 T2 T4 T5 T7 T5 T7 T5 T7 T8
42 T1 T2 T4 T5 T7 T5 T7 T6 T7 T8
43 T1 T2 T4 T5 T7 T6 T7 T5 T7 T8
44 T1 T2 T4 T5 T7 T6 T7 T6 T7 T8
45 T1 T2 T4 T6 T7 T5 T7 T5 T7 T8
46 T1 T2 T4 T6 T7 T5 T7 T5 T7 T8
47 T1 T2 T4 T6 T7 T6 T7 T5 T7 T8
48 T1 T2 T4 T6 T7 T6 T7 T6 T7 T8
49 T1 T2 T2 T4 T5 T7 T5 T7 T5 T7 T8
50 T1 T2 T2 T4 T5 T7 T5 T7 T6 T7 T8
51 T1 T2 T2 T4 T5 T7 T6 T7 T5 T7 T8
52 T1 T2 T2 T4 T5 T7 T6 T7 T6 T7 T8
53 T1 T2 T2 T4 T6 T7 T5 T7 T5 T7 T8
54 T1 T2 T2 T4 T6 T7 T5 T7 T5 T7T8
55 T1 T2 T2 T4 T6 T7 T6 T7 T5 T7 T8
56 T1 T2 T2 T4 T6 T7 T6 T7 T6 T7 T8
57 T1 T2 T2 T2 T4 T5 T7 T5 T7 T5 T7 T8
58 T1 T2 T2 T2 T4 T5 T7 T5 T7 T6 T7 T8
59 T1 T2 T2 T2 T4 T5 T7 T6 T7 T5 T7 T8
60 T1 T2 T2 T2 T4 T5 T7 T6 T7 T6 T7 T8
61 T1 T2 T2 T2 T4 T6 T7 T5 T7 T5 T7 T8
62 T1 T2 T2 T2 T4 T6 T7 T5 T7 T5 T7 T8
63 T1 T2 T2 T2 T4 T6 T7 T6 T7 T5 T7 T8
64 T1 T2 T2 T2 T4 T6 T7 T6 T7 T6 T7 T8

VII. CONCLUSION

In this paper, we have presented a novel approach for specification representation and automatic test case generation. Implementation of the above said approach is developed in C language.This approach automatically generates the path for all possible test cases which are executed to test the system. In the future, I plan to perform an experimental study to investigate the presented approach of path generation for different types of system models, including industrial models, to determine the effectiveness of the presented approach even with models with large no of states. At present we have generate a sequence of transition which should be followed for a test case. In future I plan to generate the automatic test case script which can automate the testing.

REFERENCES:

1. Ynaping Chen, Robert L. Probert and Hasan Ural, "Regression Test Reduction Using Extended Dependence Analysis" in SOQUA'07, September 3-4 2007, ACM Transaction, Dubrovnik, Croatia, 2007.
2. Korel, B., Tahat, L.H., and Vaysburg, B., "Model-based regression test reduction using dependence analysis", In *Proc. of ICSM'02* (Montréal, Canada, October 3-6, 2002). IEEE Computer Society Press, Washington, DC, 2002, 214-223.
3. Chen, Y., Rosenblum, D., VO, K., "Testtube: A System for Selective Regression Testing," Proceedings of the 161h International Conference on Software Engineering.
4. Tahat, L., Vaysburg, B., Korel, B., Bader, A., "Requirement-Based Automated Black-Box Test Generation," Proceedings of the 25th Annual IEEE International Computer Software and Applications Conference (COMPSAC), Chicago, IL, pp. 489-495
5. Vaysburg, B., Tahat, L., Korel, B., Bader, A., " Automating Test Case Generation from SDL Specifications," Proceedings of the 18th International Conference on Testing Computer Software (TCS), Bethesda, MD, pp. 130-139.
6. Dick, J., Faivre, A., "Automating the Generation and Sequencing of Test Case from Model-Based Specification," Proceedings of the Industrial Strength Formal Methods, 51h International Symposium on Formal Methods, pp. 268-284, Springer-Verlag, Apri11992.
7. Dssouli, R., Saleh, K., Aboulhamid, E., En-Nouaary, A., Bourhfir, C., "Test Development For Communication Protocols: Towards Automation," Computer Networks, 31, pp. 1835-1872, 1999
8. Ferrante K., Ottenstein K., Warren J., "The Program Dependence Graph and its Use in Optimization," ACM Transactions on Programming Languages and Systems, 9(5),pp. 319-349, 1987.
9. Ryan Voigt, Kareem Fazal, Hassan Reza,"Specification-based Testing Method Using Testing Flow Graphs" ICSEA '07 Proceedings of the International Conference on Software Engineering Advances, ISBN:0-7695-2937-2
10. Vaysburg, B., Tahat, L., Korel, B., "Dependence Analysis in Reduction of Requirement Based Test Suites," to appear in Proceedings of IEEE International Symposium on Software Testing and Analysis (ISSTA), Rome, Italy, 2002.
11. Ashish Kumari, Dr. Rahul Rishi, "Specification Representation and Test Case Reduction by Analyzing the Interaction Patterns in System Model", Proceedings of IJCSMS, Vol. 12, Issue 01, January 2012, ISSN (Online): 2231-5268.