Towards Automated Testing of Multi-Agent Systems Using Prometheus Design Models

Shafiq Ur Rehman[1], Aamer Nadeem[1], and Muddassar Sindhu[2]
[1]Center for Software Dependability, Capital University of Science and Technology, Pakistan
[2]Department of Computer Science, Quaid i Azam University, Pakistan

Abstract: *Multi-Agent Systems (MAS) are used for a wide range of applications. Goals and plans are the key premise to achieve MAS targets. Correct and proper execution and coverage of plans and achievement of goals ensures confidence in MAS. Proper identification of all possible faults in MAS working plays its role towards gaining such confidence. In this paper, we devise a model based approach which ensures goals and plans coverage. A Fault model has been defined covering faults in MAS related to goal and plan execution and interactions. We have created a test model using Prometheus design artifacts, i.e., Goal overview diagram, Scenario overview, Agent and Capability overview diagrams. New coverage criteria have been defined for fault identification. Test Paths have been identified from test model. Test cases have been generated from test paths. Our technique is then evaluated on actual implementation of MAS in JACK Intelligent Agents is a framework in Java for multi-agent system development (JACK) by executing more than 100 different test cases. Code has been instrumented for coverage analysis and faults have been injected in MAS. This approach successfully finds the injected faults by applying test cases for coverage criteria paths on MAS execution. 'Goal plan coverage' criterion has been more effective with respect to fault detection while scenario, capability and agent coverage criteria have relatively less scope in fault identification*.

Keywords: *Goal sub goals coverage, MAS faults identification, model based goal plan coverage*.

Received April 18, 2016; accepted September 19, 2016

1. Introduction

Multi Agent Systems (MAS) have been adopted widely in complex systems due to agent's unique features like reactivity, pro-activity, autonomy and social ability [7]. Autonomous agents are programmed to perform automatically and all of their activities converge towards achieving their defined goals by any possible way [7, 21]. Agents interact with each other to achieve their designed goals. All these features of agents and MAS pose challenges that must be handled and tested before MAS goes into operation.

Testing is aimed at finding inconsistencies between the system's expected output and actual output [4]. Testing can be performed at unit, integration and system level; we are targeting system level testing of MAS. Model Based Testing (MBT) uses system models to generate tests for System Under Test (SUT). Design artifacts exhibit rich information of a multi agent system and its internal working. Testing based on extracting test requirements from system models is useful for revealing faults in multi-agent systems testing.

There are many MAS development methodologies and one of the detailed methodologies is Prometheus [7, 8] that is extensively in use since more than a decade [2]. Prometheus methodology has three phases: system specification, architectural design and detailed design. System specification phase identifies environment, external actors, goals and scenarios with details of actions and percepts involved. Architectural design phase defines agent and interaction protocol involved in system overview. Detailed design phase has plans and capabilities for goals defined in system specification phase [7]. Prometheus has tool support available called Prometheus Design Tool (PDT) which supports all artifacts and can generate skeleton code from detailed design [17]. Based on the richness of Prometheus methodology, we have used Prometheus design artifacts for a test model generation. PDT generates skeleton code for JACK Intelligent Agents is a framework in Java for multi-agent system development (JACK) [5] development environment [18].

Correct and ordered execution and achievement of goals and plans in MAS can assure its correctness. Goal deliberation and goals completeness work has been done by Thangarajah *et al*. [11, 12, 13] and Duff *et al*. [1], but goal and plan coverage and their coverage criteria definition work seems missing. Although some work has been done in [15] covering only single scenario, no system level testing has been performed. Our system level testing approach using MBT will utilize most of the design artifacts in MAS testing. Each design diagram of MAS, i.e., interaction protocol, goal, scenario, process and agent overview etc, contains features that must be covered for reliability. We assume that design models are complete and specified requirements are properly propagated from specification to details design. As design faults are detected and handled in other researches. Based on the importance of goals and plans correct execution for

MAS reliability, we have two research questions which we cover in this paper.

1. What Types of faults can occur in MAS?

This research question relates to the different types of faults that can occur in MAS operations and how different types of artifact interactions can cause faults in MAS?

2. How MBT is effective in MAS fault detection? How models can be used to ensure goal and plans coverage?

This involves answering the effectiveness of design model in MAS testing? How model coverage ensures reliability in MAS? How goals and sub goals and plan coverage is vital to MAS testing? How faults are detected in MAS when goals and plans coverage is performed?

A fault model has been devised by considering possible occurrence of faults in MAS. Plans are used to contain the steps to fulfill goals completeness. Goals are defined and plans are triggered to achieve desired goals. Goals and plans coverage with respect to their execution and order is critical for testing adequacy. In this case adequate testing can claim reliability of MAS. The test model has ability to cover possible aspects of model coverage. Coverage criteria can ensures testing adequacy. Coverage criteria have been defined for test model upon which test paths have been generated for each coverage criterion. JACK development environment [20] is used for MAS implementation, which is then instrumented to evaluate our testing framework. Test case execution and evaluation shows different types faults identification.

Section 2 presents literature work that is done regarding MAS goal and plans testing with reference to finding faults. Section 3 presents our fault model for multi-agent systems. Section 4 presents our testing framework and process for testing. Section 5 presents results and discussion for faults detected and problems. Finally section 6 concludes the work presented in this paper and references are shown in last section.

2. Literature Review

In this section, existing research and development which have been done so far regarding goal, sub-goals and plan based faults identification in MAS are presented and limitations in literature are discussed.

Thangarajah *et al*. [11, 12] present an approach to quantify goal completeness and level of completeness of goal in Belief-Desire-Intention (BDI) [12] multi-agent system. Completeness has been measured by considering resources consumed by a goal and measure the effect of goal in terms of desired outcomes achieved. Goals and plans coverage criteria have not been defined neither relationship of goal-plan tree with respect to scenarios and protocol diagram. Faults identification has not covered in [11, 12]. Padgham *et al*. [6] present model based test oracle creation for unit testing of agents. Fault model has been created to cover individual units. Event plan tree has been developed but goals and their links to plans and sub-goals are missing. System specification level design diagrams are not used and adequate coverage criteria for coverage of maximum functionality of MAS using Prometheus design artifacts are also missing. Goal and goal-plan related faults identification is also missing.

Model driven architecture for building the multi-agent systems has been presented in [3] but still no verification is performed.

Thangarajah *et al*. [13] present a technique to measure plans coverage by using numeric measures and their overlap for agents. Coverage is measured by number of models or area a plan is applicable. No implementation and validation have been done neither any fault model nor coverage criteria are defined for goals and plans.

Positive and negative goal interactions have been discussed in [10, 14, 18]. Negative interactions are basically conflicts between goals. They have defined resource requirements of a goal by considering all of its plans. Focus of their work is on defining goal plan tree annotated with resources both at start and run time [14]. For goal plan tree modeling, five tuples prolog function node is used in [10]. Faults that may occur if a certain interaction or coverage not covered are not discussed. No coverage metrics has been defined neither design diagrams used for tree construction are elaborated.

Thangarajah *et al*. [15] use scenarios of Prometheus methodology and added a structure in scenario, e.g., added sequence, test descriptor and traceability link. A limitation is that only a single scenario is tested in isolation, no system level traceability is performed. MAS' execution and faults are not considered in this approach. Zhang *et al*. [22] presented an approach for automated testing for units in MAS. Orders of events, plans were defined and test cases were executed with proper test data [22]. Only unit testing is performed, no faults identification model and coverage metrics for goals and plans were discussed.

Partially complete and partial achievement goals have been presented by [9, 23]. No detail has been provided to show whether a goal will be satisfied or not instead only progress is considered with reference to goal achievement. Action and impact of goal modification have not been analyzed. Thangarajah *et al*. [16] defined several criteria for agent consideration or discussion, e.g., time varying utilization, deadline, resource requirement, dependencies, communication with other goals etc., [16]. Main consideration in their work is goal deliberation but no faults that may occur in MAS are identified and detected. How AND, OR constrains between goals and plans are covered are not discussed.

The existing techniques on model based testing of MAS considers only goals completeness or plan

coverage but there remains some faults that occurs in system level interaction between artifacts, i.e., goal triggers plan and plan generate sub-goals and so on. Design diagrams like goal, system overview, scenario and process diagrams contain the information required and utilized in system level operations of MAS. No such technique exists which uses all three Prometheus phases design artifacts in testing. No fault model is presented if certain goals are not achieved or plans for the goal are not triggered. Current work in the literature does not target faults description, identification and complete coverage of goals and plans.

3. Fault Model

MAS have many features, if a feature that should be present is not exhibited or not specified feature is present then there is fault in MAS. A fault can also be an undesired event or action in MAS, e.g., a triggering event may not have been triggered or an action may not have been posted or system reacts undesirably upon receiving a triggering event etc., Goal's and plan's correct and ordered execution is necessary for MAS reliability. In Prometheus Methodology, goals have been defined in goal overview diagram at system analysis phase and are assigned relevant plans in the detailed designed phase. Some goals have more than one applicable plans, all of which must be executed in order for a goal to be considered as achieved, while some goals are achieved if any of applicable plans is executed. Same is the case with plans and their sub-goals. Such type of relationships are handled by 'AND' and 'OR' relationships between goals.

Different types of faults can occur in MAS some of them are discussed by Padgham *et al*. [6] like incorrect belief, incorrect context etc, but there are certain aspects of MAS that are still missing and can cause MAS to behave unexpectedly. In our fault model we have also captured this relationship along with other possible faults that may occur when a MAS is running. Following are our defined fault types and their description covering maximum faults occurrence in MAS:

- *Inaccurate goal achievement*: if more than one plans are required to execute in order to fulfill a certain goal then missing any of the plan can cause inaccurate goal achievement fault in MAS. This could occur when a certain goal has an AND relationship with all of its plans.
- *Plan Failure*: certain plans have more than one sub-goals to achieve; sub-goals have an AND relationship with the plan. Missing any of such sub-goals can cause plan not to produce desired output.
- *Internal Agent fault*: such faults can occur if a certain agent or its capability has not been executed. Non execution of a certain capability cannot reveal its agent's operations and contribution to meet system goals.
- *Missing functionality*: such type of faults can occur if a goal has more than one alternative plans. These plans have an OR relationship with the goal; so non-coverage or non-execution of all of OR plan branched/arcs can cause missing functionality faults.
- *Scenario Fault*: Scenario contains sequence of steps to perform in MAS in the form of goal, action and percepts. If a scenario is not covered properly then there could occur a scenario fault in MAS.
- *Deliberate Fault*: desired output of the MAS can be obtained only by correct execution order of the plans and sub-goals. If an agent triggers the incorrect plan which should not be executed as required then deliberate faults can occur. Correct communication within and between agents should be required. Such types of faults could also occur due to wrong implementation with respect to design.

4. Testing Framework and Process

This section describes our testing framework and testing process of our approach for system level testing of MAS. Our target is to ensure thorough coverage of plans and goal for MAS. Figure 1 shows overall testing process in which maximum utilization of Prometheus design artifacts in test model construction is done. For each scenario there exists a goal overview diagram. Each goal has associated plan(s) or capability in detailed design phase called process diagrams. The test model is constructed by considering all scenarios, goal diagrams and process diagrams. Fault occurrence can cause the MAS to deliver an unexpected outcome. Identify coverage criteria and then apply on test model for test paths generation. Test paths are generated automatically against each coverage criteria. Test paths will lead to the generation of test cases and semi-automatic generation of test data. Expected output is calculated manually for test results evaluation. Actual executable code of MAS is managed in JACK development environment. We have instrumented MAS code to get execution traces when test cases are executed. Our testing process identifies faults that occur because of wrong or ambiguous implementation of MAS design into code.

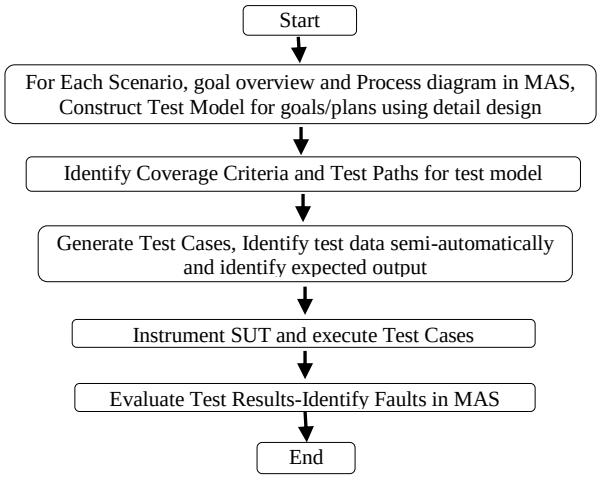

Figure 1. Overview of testing process for goal and plans coverage.

Figure 2 shows testing framework having five main processes, e.g., goal-plan graph generator, test paths generator, test case generator, test case executor and test result evaluation. We have used design artifacts of Prometheus methodology as it is a rich methodology for MAS designing. Sub-sequent sections elaborate each process, i.e., test model generation, coverage criteria definition, test paths generation, test case generation, execution and evaluation for MAS testing with reference to goals and plans. Test paths are used for generation of test cases, elaborated in subsequent sections. Test cases are generated for MAS implementation and they are executed to reveal injected faults. When a test case has correct output as expected then it is considered as pass. Failed test cases have incorrect output. Failed test cases are further discussed with the reason why path is deviated that caused wrong output. Test result evaluation is performed manually.

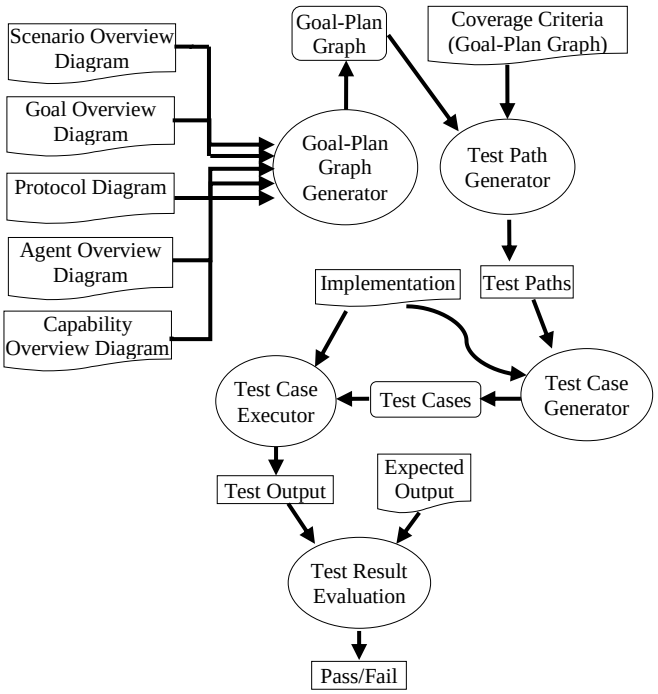

Figure 2. Testing framework for goals and plans coverage.

4.1. Test Model Generation

Goals and plans are the factors used to measure the correctness of MAS working. Every MAS has goal diagram for each scenario. Scenario contains goal, actions and percepts that occurs specific to scenario.

We have taken the case study of Multi Currency Banking System described in [5]. It has three agents, e.g., Bank Account agent, Currency Exchange agent, and Communicator agent [5] which work together to create account, debit account, credit account, debit and credit account with same and different currency and currency conversion. We have design artifacts of MAS which will be used to generate test model. Notations used in all design artifacts are standard notations used in Prometheus Design Tool (PDT) [17]. Figure 3 presents scenario and goal overview diagram of our case study. MAS have three main scenarios which consist of a sequence of goals, actions and percept to perform. As depicted in Figure 3 the operate account scenario has two sub scenarios to handle, e.g., credit account scenario and debit account scenario. Credit account and debit account scenario has an OR constraint with three sub-goals, any of its sub goal's successful execution can lead to positive contribution to its main goal achievement, e.g., debit or credit account. Currency exchange goal has an AND constraint with its sub goals like set exchange rate goal and perform exchange goal. Perform exchange goal has a need to achieve compute rate. Compute rate has an OR constraint with Identify rate and Two Step Exchange goal. Two Step Exchange goal is triggered if two step currency conversions are required. Goals and sub goals have their plans in detailed design. Plans are defined in process diagrams and each plan has goals to satisfy. Every plan has exactly one triggering goal and multiple sub-goals (steps) in the plan.

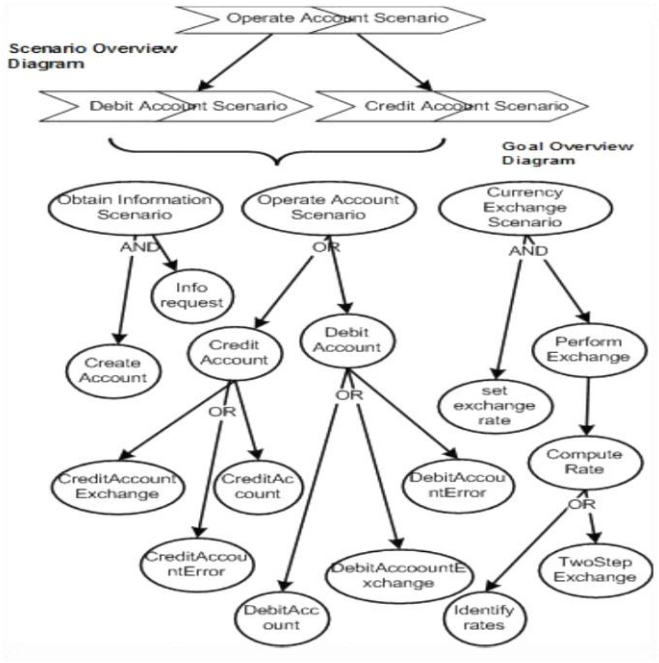

Figure 3. Scenario and goal overview diagram of MAS.

Satisfaction of all sub-goals in a plan means the plan is satisfied, and therefore its triggering (sub) goal is achieved. Sub-goals are specified in a plan while designing the MAS. The steps that need to be executed as part of a plan are determined and included as sub-goals.

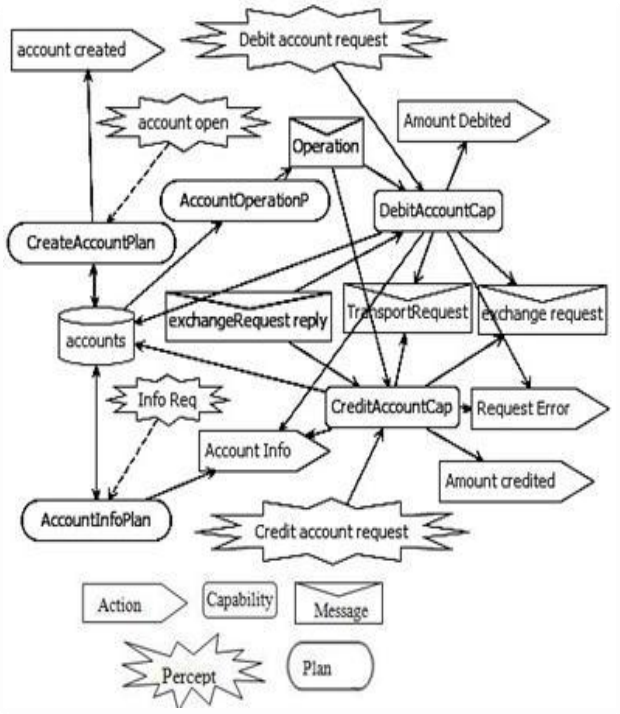

Figure 4. BankAccount agent overview diagram of MAS.

The BankAccount agent overview diagram is shown in Figure 4; it has two capabilities, i.e., Credit Account Cap and Debit Account Cap and three plans, i.e., Create AccountP, Account Info P, Account Operation P which have some goals to achieve. Percepts and messages are triggering events for the plans and capabilities as shown in diagram. Arrow shows the flow of information from one entity to other. Each capability is further elaborated in capability overview diagram as depicted in Figure 5. Each capability has three plans for alternative three goals as depicted in Goal Overview diagram.

In case of exchange request Communicator agent get the message of Transport Request; which generates Exchange Request message for Currency Exchange agent which execute its relevant plans and capability i.e., Perform Exchange P, Set Exchange Rate Plan and one capability Compute Rate.

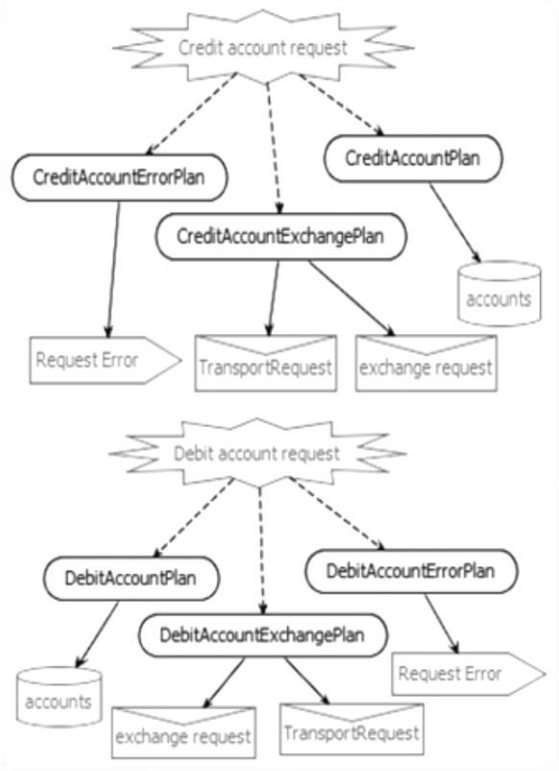

Figure 5. Debit account cap and credit Account cap capability overview diagram of MAS.

Currency Exchange agent have two plans i.e., Perform Exchange P, Set Exchnage Rate Plan and one capability Compute Rate. While considering the protocol diagram loop can be on creation of accounts and debit or credit the account more than one time.

4.1.1. GOAL-PLAN GRAPH

We use the details contained in design artifacts for our case study discussed in section 4.1 and generate a Goal-Plan Graph (GPG) i.e., Test Model. We have used Prometheus design artifacts, e.g., scenario overview diagram, goal overview diagram, protocol diagram (only for loops), agent and capability overview diagrams. These design artifacts contain rich information from goals identification to assignment of plans for that goal. Protocol diagram is used as interaction between agents and also contains different loops in it. We have used only loops information from protocol diagram and added loops in our test model.

Algorithm 1 is designed that takes these design artifacts as input, extract and process goals and plans information, and generates a test model which is the Goal-Plan Graph. It extract sub-goals from the body of plan using process diagrams, i.e., agent and capability diagrams; and add sub-goal to Goal-Plan Graph as they are listed in goal overview diagram. We build GPG for each scenario and link different GPG of the system by looking their working in agent overview diagram. Algorithm 1 generates a list of all goal and plan from design diagrams. Applicable plans list contains applicable plans for each goal along with related scenario, agent and capability. Sub-goals list is

prepared for each plan containing its sub-goals. Step by step GPG will be generated by the listed steps in Algorithm 1. GPG in Figure 6 consists of nodes and edges where nodes are of two type i.e., goal node and plan node. Each node has relevant scenario, agent and capability associated which are also annotated with the node.

Algorithm 1: Goal-Plan graph generation Algorithm Using Prometheus Design Artifact

Input: Goal-Overview Diagram (GD), Scenario Diagram (SD), Protocol Diagram (PD), Agent Diagram (AD) and Capability Overview Diagram (CD).
Output: Goal-Plan Graph (GPG) with plans and goals as nodes.
Declare: GPG=empty, SG is the sub-goal, AS=Applicable Scenario, AA=Applicable Agent, AC=Applicable Capability, AP=Applicable Plan, Each capability will be treated as a plan as well.
Step 1: Extract goals list from GD: GL← GD.goals
Step 2: Extract plans from AD and CD: PL ← AD.plans ∪ CD.plans
Step 3: For each Goal and Plan
Step 4: Add Plan P(G)← List of Applicable Plans (G, AP)
Step 5: Add SG (P)← List of sub-goals for Plans (P, SG).
Step 6: Add Scenario S(G)/S(P)←Scenario for Goal/Plan (G/P,AS)
Step 7: Add Agent A(G)/A(P)← Agent for Goal/Plan (G/P, AA)
Step 8: Add Capability C(G)/C(P)← Capability containing Goal/Plan (G/P, AC)
Step 9: For each Goal-Diagram against each Scenario
Step 10: Set Root (GPG)← GD.root
Step 11: Set Current Goal (CG)← Root
Step 12: Add S (G), A (G) and/or C(G)
Step 13: Add Children (CG)← AP
Step 14: Add Constraint(G-Node)←AND or OR
Step 15: Add S(P), A(P) and/or C(P)
Step 16: For Each Plan (P, CG)
Step 17: Add Children (P)← (SG, P)
Step 18: Add Constraint(P-Node)←AND or OR
Step 19: Set CG←SG
Step 20: While CG ≠ {}
Step 21: Repeat step 11-18
Step 22: End While
Step 23: If Goal-Diagrams > 1 and n = Number of Scenario
Step 24: Add link GPG (Scenario-I) to GPG (Scenario-n) Using Detail Design
Step 25: Extract Loops from PD
Step 26: Add Loop link goal← Plan
Step 27: Return GPG

GPG in Figure 6 shows the complete flow of system from high level goal to detail sub-goals and plans execution. Each goal can have more than one applicable plans where all applicable plans can have 'AND' or 'OR' relationships annotated with arcs. Every plan has exactly one triggering goal and multiple sub-goals (steps) in the plan. These sub-goals can also have 'AND' or 'OR' relationships. Loop edges are always starts from an arrow from plan to goal somewhere earlier. Each node contains metadata which includes scenario, agent and/or capability. Every plan and goal belongs to some scenario and performed by some agent within any capability belonging. Such detail of node type is also included in GPG nodes as metadata of a node.

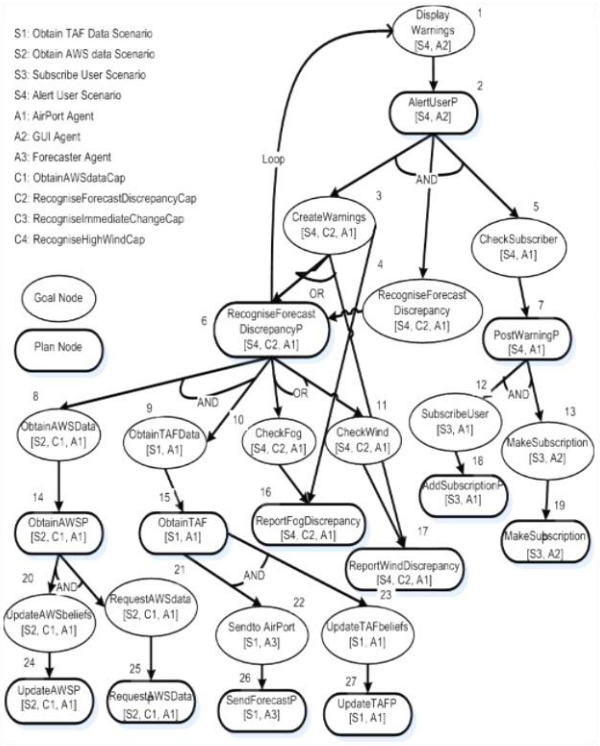

Figure 6. Goal-Plan graph (Test Model) for multi currency MAS.

4.2. Coverage Criteria

It is possible that some parts of system under test remains untested that can cause problem in MAS operation. Maximum possible coverage is essential. Once Goal-Plan Graph (test model) is generated for goals, sub-goals and plans; there is a need to measure the coverage of executed/traversed nodes in the test model. MAS posses different characteristics so new coverage criteria have been defined that are different from the literature. To ensure maximum coverage of all goals and plans in MAS we have defined following coverage criteria for Goal-Plan Graph.

1. *All goals Coverage*: A set of Test Paths (TP) is said to satisfy all goals coverage criterion for Goal-Plan Graph G if each goal node g of graph G is included in at least one path P ϵ TP.

Test path(s) in which all goals from goal-diagram have been covered at least once. Only all AND condition branches will be covered. As shown in Figure 7, if OR is the constraint then only one path coverage is enough which traverse only all goals.

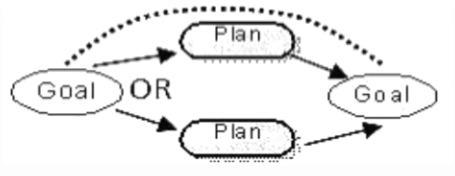

Figure 7. All Goal Coverage (OR constraint).

2. *Scenario Coverage*: A set of Test Paths (TP) is said to satisfy Scenario coverage criterion for Goal-Plan Graph G if each Scenario S of graph G (nodes metadata) is included in at least one path P ϵ TP.

Test path(s) in which every scenario have been covered at least once**.**

3. *Agent Coverage*: A set of Test Paths (TP) is said to satisfy agent coverage criterion for Goal-Plan Graph G if each agent A of graph G (nodes metadata) is included in at least one path P ϵ TP.

Test path(s) in which every agent has been traversed at least once.

4. *Capability Coverage*: A set of Test Paths (TP) is said to satisfy Capability coverage criterion for Goal-Plan Graph G if each Capability C of graph G (nodes metadata) is included in at least one path P ϵ TP.

Test path(s) in which every capability(s) have been covered at least once.

5. *Plan Coverage*: A set of Test Paths (TP) is said to satisfy Plan coverage criterion for Goal-Plan Graph G if each Plan node p of graph G is included in at least one path P ϵ TP.

Test path(s) in which every Plan has been covered at least once. Only all AND condition branches will be covered. As shown in Figure 8, if OR is the constraint then only one path coverage is enough which traverses only all plans.

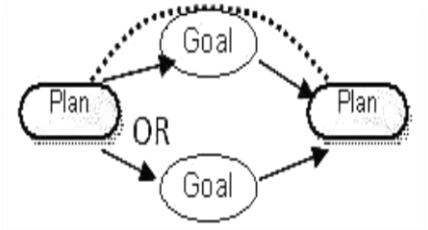

Figure 8. Plan Coverage model (OR constraint).

6. *Goal Plan Coverage*: A set of Test Paths (TP) is said to satisfy Goal Plan coverage criterion for Goal-Plan Graph G if each Arc of graph G is included in at least one path P ϵ TP.

Test path(s) in which every goal and its all applicable Plans (Arcs) must be covered at least once. It will cover OR conditions branches as well.

7. *Loop Coverage*: A set of Test Paths (TP) is said to satisfy Loop coverage criterion for a protocol graph G if it traverses each loop 0, 1 or more than one time in graph G and loop path(s) included in at least one test path P ϵ TP.

A set of test paths which by-passes every loop and a set of test paths which traverse each loop exactly once and a set of test paths which traverse each loop more than once.

Loop coverage is necessary to test functionalities in which a goal/plan is called more than once and in literature prime path and loop coverage 0, 1 or more than once is suggested. In MAS loop coverage 0 time, 1 time and more than one i.e., 2 is useful to check stability in multiple calls to certain goal.

4.3. Test Paths Generation

Test paths are generated from test model. We have automated test paths generation process with the help of a tool that takes a test model as input, apply different coverage criteria and generate test path against each coverage criteria. Based on our GPG test model in Figure 6, we have categorized goals and plans as basic nodes types. Based on coverage criteria; agent, scenario and capability coverage are considered as meta-data coverage as depicted in GPG. Algorithm 2 is used for automated test paths generation for each coverage criteria. Figure 9 shows the basic architecture of automatic test paths generation by following Algorithm 2. For loop coverage we created a list of nodes containing the loop edges and check its one, two or more than two occurrences.

Algorithm 2: Test Path generation Algorithm Using Test Model (Goal-Plan Graph)

Input: Goal-Plan Graph and Coverage Criteria
Output: Test Path for each Coverage Criteria
Let GPG be the Goal-Plan Graph with node type i.e. goal or plan,
meta-data (Capability, Agent, Scenario) and AND or OR constraints with edges.

Step 1: Insert metadata (Nodes) in data array
Step 2: Insert Nodes types (goal/Plan) in Array
Step 3: Make list of AND/OR edges
Step 4: If criteria = All Goals coverage/Plans coverage
Step 5: Call findpathsbytype ()
Step 6: End If
Step 7: If criteria = Capability/Agent/Scenario coverage
Step 8: Call findpathsbymetadata ()
Step 9: End If
Step 10: If criteria = Goal Plan coverage
Step 11: Call findall ()
Step 12: End If
Step 13: If criteria = Loop coverage
Step 14: Call findloop ()
Step 15: End If

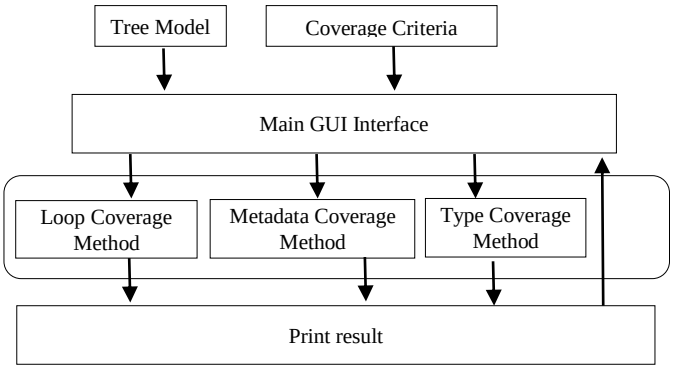

Figure 9. Automatic test path generation architecture using GPG.

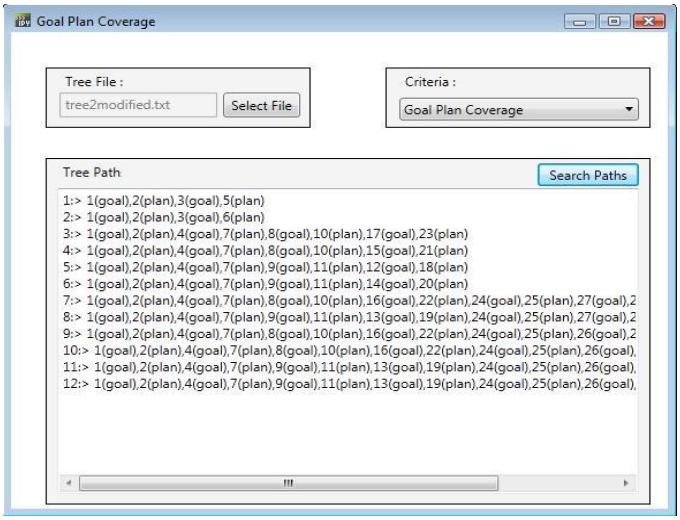

Figure 10. Test Paths generation tool GUI.

Goal-Plan Graph structure used for automatic test paths generation is presented in Table 1. For understanding purpose only one node structure has been presented here. For each coverage criteria test paths are generated. Type coverage method covers all goals coverage, all plans coverage and goal-plan coverage criteria. Metadata coverage method covers scenario, agent and action coverage criteria. Generated test paths have relevant coverage criteria node name in it. e.g., 1(goal)→2→3 (goal)→6→10(goal)→16, one of the paths from all goals coverage criteria. The GUI of Test path generation tool is shown in Figure 10.

Table 1. Structures of the goal-plan graphs used paths generation.

Node Name	Node Metadata	Node type (G/P)	Node No	AND/OR constraint
debitaccountplan	[s3,c2,a1]	Plan	7	OR, (7,8), (7,9)
Example single node structure: debitaccountplan;[s3,c2,a1];plan;7				

4.4. Test Case Generation and Execution

Test case generation consists of two parts. First one is to identify variables used in test cases and second part is assigning test data to test case variables. Variables identification step is manual. Test cases are generated from test paths. Each test path consists of nodes and edges. Each node has some related information that will be used to generate a test case.

Each Node → Info (properties) →Extract variables associated at each node→Identify functions associated to the variables→Assign test data semi-automatically.

We construct a Node Description Table (NDT) manually for each path and use the variables or properties associated at each node for test case generation. Test cases then consist of value combinations of variables that make a certain path to follow. For example the test path: "1(goal) → 2 (Plan) → 4(goal) → 7(Plan) → 8(goal) → 10(Plan) →15(goal)→21(Plan)" has NDT presented in Table 2.

Table 2. Node description table for test paths nodes.

Node No.	Node Type	Associated Variables/functions
1	Goal (Obtain Information)	String = Account Title Function = Inquire
2	Plan (ObtainInfoP)	Triggering event = Yes String = Title
4	Goal (Account Operation)	String = Title Double = amount
7	Plan (Account opP)	Event = yes String = Title Double = amount String = Currency
8	Goal (Credit account)	Function = credit account
10	Plan (Credit AccountP)	String = Title Double = amount String = Currency

Figure 11 shows test case generation process for MAS under test, i.e., banking system. A MAS has many functions that are called or triggered. The number of test cases to execute depends on generated value combination for each variable, i.e., (Select number from array of values) and the number of generated patterns (number of test cases) for variables that make MAS to execute.

Test data generation and test case execution is semi-automatic. Test case execution process requires several set up variable values involved in test cases. Once values have been assigned then combination of execution is hard coded into MAS implementation. Randomly these values are called along with function name automatically once we run the implementation. Some details of the code have been shown at the end of this section.

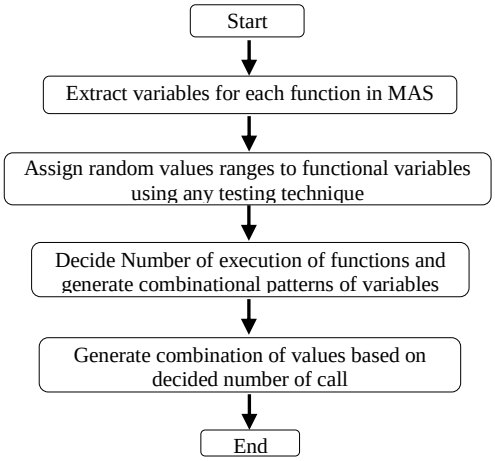

Figure 11. Test case generation process for MAS.

There are 8 plans for accounts operations handled by Bank Account agent, one plan for Communicator agent and 5 plans for Currency Exchange agent. Account name, currency and amount are three variables extracted and combinations of values are assigned for test case generation. JACK code has been instrumented for automatically assigning variable values for test case generation and test cases are executed by providing the total number test cases to execute. Patterns of variables for test case executions are then automatically formed. Instrumented code will generate output showing details of executed or traversed plans in test case execution. Only part of added code in JACK is shown below.

```
int length = ary.length,seq_num, looplen2 = 0;
seq_num = 0 + rand.nextInt(100);
// 100 test cases are generated
int[][]seq = new int[seq_num][];
callCommands(seq[i],communicator,nextName(),nextCurr(),
nextAmount());
```

Test case structure after extracting variables and functions from NDT and assigning test data for our MAS under test is as follows:

Test Case: createAccount (John,USD,100)|creditAccount (John,AUD,200) |debitAccount (John,AUD,50) etc.

4.5. Test Result Evaluation

This section discusses about manual calculation of expected output and test results evaluation. After executing test cases, we have our test case results which are used for test result evaluation. For example account debit request is made with 50 dollar then expected output shown that 50 dollars debited from given account etc. Output of MAS is compared with the expected output. If expected and actual output is same then we declare the test case as a pass otherwise a fail. A failed test case can be analyzed to trace the fault that caused the wrong output. We identify which node has caused fault in MAS. Faults are injected in MAS implementation. Test case output will reveal faults after executions of a test case's set. Even a single test case can identify an injected fault which is clearly compared with those of expected results.

Different coverage criteria paths have different test cases, while running these test cases reveals certain faults identified earlier in Section 3. Detailed test result evaluations with faults are presented in results and discussion section.

5. Results and Discussion

In this section, we discuss faults detected in relation to the fault model. For each fault type at least one fault is injected in the MAS implementation. We achieve effectiveness of our testing approach after finding injected faults in MAS. These faults are detected by applying different test cases that are generated from different coverage criteria paths. Coverage criteria ensure certain types of faults detection and identification with a system [19].

Table 3. Injected faults in multi-currency banking MAS.

Fault ID	Fault Type	Injected Faults details
F-1	Plan failure	CreditAccountPlan not covered- Making its context false
F-2	Inaccurate goal achievement	Debit Account goal not triggered – event for debit account not posted "#posted as" not working
F-3	Scenario fault, Internal agent fault	#posts event TransportRequest tev; not posted - Agent functionality missed, currency exchange scenario missed
F-4	Plan failure, Missing functionality	Compute rate event handler made false - Capability missed
F-5	Deliberate faults, Internal agent fault	#reads data Account "accounts" not allowed – database reading/writing not allowed - Deliberate faults
F-6	Missing functionality, Deliberate Fault	Obtain Information event not triggered after Node 6 etc - loop not executed

Table 3 provides the details of injected faults in multi-currency banking MAS, e.g., making context condition of plan to false, prevent plans not to trigger, changing the code so optional goal of a plan is not triggered, making certain scenario and capability not to execute etc. We have applied more than 100 test cases on implementation of multi-currency banking system case study. These test cases were selected after multiple executions of MAS. Test cases output is compared with the expected output. Execution trace is used to analyze failed test cases to identify which node created fault.

Table 4 shows detected faults by applying coverage criteria and minimum required test cases to cover test criterion. These minimum test cases are chosen after multiple execution and their result with respect to faults detected. It shows effectiveness coverage criteria in identifying injected faults, different coverage criteria reveals different faults in MAS. Test cases have been applied on the instrumented code and it is found that by applying our coverage criteria, which are defined in section 4.2, uncovers different faults discussed in section 3.

Figure 12 shows graphical representation of no of test cases executed for each coverage criterion and types of faults detected. At least 13 test cases are required to cover 'goal plan coverage' criterion. Our testing approach has been seen effective in identifying faults of different types when it comes to goal and plan coverage. Injected faults were successfully revealed by applying coverage criteria. For each coverage criteria we need certain test cases which assure its coverage.

Table 4. Detected faults by coverage criteria and minimum test cases required.

S. No	Coverage Criteria	Test cases (TC ID)	Faults Detected
1	All goals Coverage	5 Test cases	F-2, F-6
2	Scenario Coverage	6 Test Cases	F-3, F-6
3	Agent and capability Coverage	6 Test Cases	F-4, F-5
4	Plan Coverage	8 Test Cases	F-1, F-4
5	Goal Plan Coverage	13 Test Cases	F-1, F-2, F-3, F-4, F-5
6	Loop Coverage	8 Test Cases	F-2, F-6

In Table 4 goal-plan coverage identifies five faults by executing 13 test cases but F-6 is not detected by goal-plan coverage criterion. Because fault-6 is relevant to missing functionality or deliberate faults, only loop coverage criterion identifies such types of faults in MAS by executing test cases which test loop events in system execution.

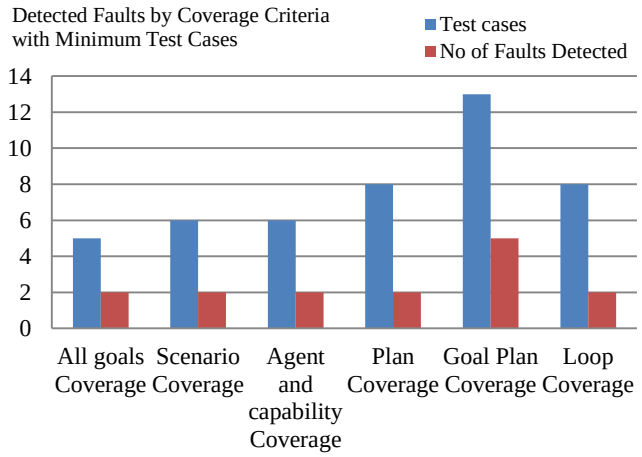

Figure 12. Chart with test cases and coverage criteria detecting types of faults.

A test case is failed either due to a plan or a goal which was not triggered thus not executing the relevant path and producing a wrong output.

Test Path = 1(goal)→2(Plan)→4(goal)→7(Plan) →9(goal) →11(Plan)→12(goal)→28(Plan).

Inject fault = Debit Account goal not triggered-event for debit account not posted "#posted as" not working.

Test case: createAccount(John,USD,100) | debitAccount (John,USD,40)

Actual output: 1(goal)→2(Plan)→4(goal)→7(Plan) →9(goal)→11(Plan)→{Not triggered}

Nodes of the path are not covered in case a fault occurs which restricts coverage/execution of certain goal and plans. Currently we are testing all possible calls in a single test case, therefore the numbers of test cases are minimum.

6. Conclusions

In this paper, we have defined a fault model for testing of MAS with respect to goals, plan and sub goals. We have used Prometheus methodology due to its rich artifacts and availability of its rich design tool i.e. PDT. We have used design artifacts of all three phases so no functionality is missed or remain uncovered.

This research paper uses scenario overview, goal overview, protocol diagram and process diagrams to generate test model for the SUT. An algorithm is defined for Goal-Plan Graph generation. New coverage criteria have been defined and automatic test paths generation has been done for each coverage criteria. JACK implementation of MAS has been instrumented to semi-automatically execute test cases. Faults are injected into MAS and test cases are executed to show identified faults. More than 100 test cases have been generated and executed on our case study for evaluation purpose. All fault categories which have been identified to find goal and plan related faults seems effective in building trust in MAS. The defined coverage criteria contribute to find possible root-cause up to node level of a detected fault.

In future work, faults that could occur in case of interaction between agents and environment will be identified in fault model. Test case generation and result evaluation process can be automated.

References

[1] Duff S., Thangarajah J., and Harland J., "Maintenance Goals in Intelligent Agents" *Computational Intelligence*, vol. 30, no. 1, pp. 71-114, 2014.

[2] Dam K., Evaluating and Comparing Agent-Oriented Software Engineering Methodologies, Ph.D. Thesis, RMIT University, 2003.

[3] Elammari M. and Issa Z., "Using Model Driven Architecture to Develop Multi-Agent Systems," *The International Arab Journal of Information Technology*, vol. 10, no. 4, pp. 349-355, 2013.

[4] IEEE. Standard for Software Test Documentation. IEEE STD 829, 1998. URL http://standards.ieee.org/findstds/standard/829-1998.html, Last Visited, 2016.

[5] Jack intelligent agents, http://aosgrp.com/products/jack/, Last Visited, 2015.

[6] Padgham L., Zhang Z., Thangarajah J., and Miller T., "Model-based Test Oracle Generation for Automated Unit Testing of Agent Systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1230-1244, 2013.

[7] Padgham L. and Winikoff M., *Developing Intelligent Agent Systems: A Practical Guide*, Wiley Series in Agent Technology. John Wiley and Sons, 2004.

[8] Padgham L. and Winikoff M., "Prometheus: A Methodology for Developing Intelligent Agents," *in Proceedings of International Workshop on Agent-Oriented Software Engineering*, Bologna, pp. 174-185, 2003.

[9] Riemsdijk M. and Yorke-Smith N., "Towards Reasoning with Partial Goal Satisfaction in Intelligent Agents," *in Proceedings of International Workshop on Programming Multi-Agent Systems*, Toronto, pp. 41-59, 2010.

[10] Shaw P. and Bordini R., "An Alternative Approach for Reasoning about the Goal-Plan Tree Problem," *in Proceedings of Languages, Methodologies, and Development Tools for Multi-Agent Systems, 3rd International Conference*, Lyon, pp. 115-135, 2010.

[11] Thangarajah J., Harland J., Morley D., and Yorke-Smith N., "Towards Quantifying the Completeness of BDI Goals," *in Proceedings of The International Conference on Autonomous Agents and Multi-Agent Systems*, Paris, pp. 1369-1370, 2014.

[12] Thangarajah J., Harland J., Morley D., and Yorke-Smith N., "Quantifying the Completeness of Goals in BDI Agent Systems," *in Proceedings of 21st European Conference on Artificial Intelligence*, Prague, pp. 879-884, 2014.

[13] Thangarajah J., Sardina S., and Padgham L., "Measuring Plan Coverage and Overlap for Agent Reasoning," *in Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*, Valencia, pp. 1049-1056, 2012.

[14] Thangarajah J. and Padgham L., "Computationally Effective Reasoning about Goal Interactions," *Journal of Automated Reasoning*, vol. 47, no. 1, pp. 17-56, 2011.

[15] Thangarajah J., Jayatilleke G., and Padgham L., "Scenarios for System Requirements Traceability and Testing," *in Proceedings of The 10th International Conference on Autonomous Agents and Multiagent Systems*, Taipei, pp. 285-292, 2011.

[16] Thangarajah J., Harland J., and Yorke-Smith N., "A Soft COP Model for Goal Deliberation in a BDI Agent," *in Proceedings of the 6th International Workshop on Constraint Modelling and Reformation*, Rhode Island, pp. 61-75, 2007.

[17] Thangarajah J., Padgham L., and Winikoff M., "Prometheus Design Tool," *in Proceedings of the 4th International Conference on Autonomous Agents and Multi Agent Systems*, Utrecht, pp. 127-128, 2005.

[18] Thangarajah J., Padgham L., and Winikoff M., "Detecting and Avoiding Interference between Goals in Intelligent Agents," *in Proceedings of the 18th International Joint Conference on Artificial Intelligence*, Acapulco, pp. 721-726, 2003.

[19] Tian J., "Quality Assurance Alternatives and Techniques: A Defect Based Survey and Analysis," *Software Quality Professional*, vol. 3, no. 3, pp. 6-18, 2001.

[20] Winikoff M., *in Multi-Agent Programming*, Springer, 2005.

[21] Wooldridge M., *an Introduction to Multi-Agent Systems*, John Wiley and Sons, 2002.

[22] Zhang Z., Thangarajah J., and Padgham L., "Automated Testing for Intelligent Agent Systems," *in Proceedings of the 10th International Conference on Agent-Oriented Software Engineering*, Budapest, pp. 66-79, 2011.

[23] Zhou Y., Torre L., and Zhang Y., "Partial Goal Satisfaction and Goal Change: Weak and Strong Partial Implication, Logical Properties, Complexity," *in Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems*, Estoril, pp. 413-420, 2008.

Shafiq Ur Rehman is PhD (CS) candidate at Capital University of Science and Technology, Islambad. He is member of Center for Software Dependability research group. His research focuses on software quality assurance and testing, spcifically model based testing of multi-agent systems. In this research area he has published journals and conferences papers as well. Besides his research activities he is working as a software test engineer as well.

Aamer Nadeem is an Associate Professor in the Department of Computer Science at Capital University of Science and Technology, Islamabad. He is also Head of the Center for Software Dependability - a research group working in the areas of software reliability, software fault tolerance, formal methods and safety-critical systems. He received his MSc in computer science from QAU, MS in software engineering from NUST, and PhD from Mohammad Ali Jinnah University, Islamabad. He did part of his PhD research work at the Chinese University of Hong Kong (CUHK) under a research collaboration. He is a professional member of the Association for Computing Machinery (ACM).

Muddassar Sindhu received his PhD from Royal Institute of Technology (KTH), Stockholm, Sweden. Currently, he is an Assistant Professor of Computer Science at Quaid-i-Azam University, Islamabad, Pakistan. His research interests include software testing, learning-based testing, formal methods and formalization of informal software requirements.