

An Automated Approach to Model Based Testing of Multi-agent Systems

By

**Shafiq Ur Rehman
PC103005**

A Thesis submitted to the
Department of Computer Sciences
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE



**Faculty of Computing
Capital University of Science and Technology
Islamabad
March, 2017**

An Automated Approach to Model Based Testing of Multi-agent Systems

By

Shafiq Ur Rehman
PC103005

Dr. Jin Song Dong
School of Computing
National University of Singapore, Singapore
Foreign Evaluator

Dr. Jianjun Zhao
Kyushu University
Fukuoka, Japan
Foreign Evaluator

Dr. Aamer Nadeem
Associate Professor, HOD BI & BS
Faculty of Computing
Capital University of Science and Technology, Islamabad
Thesis Supervisor

Dr. Nayyer Masood
PROFESSOR / HOD Computer Science

Dr. Abdul Qadir
(Professor / Dean Faculty of Computing)

Faculty of Computing
Capital University of Science and Technology
Islamabad
March, 2017

Copyright© 2017 by Mr. Shafiq Ur Rehman

All rights are reserved. No part of the material protected by this copy right notice may be reproduced or utilized in any form or any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the permission from the author.

To my Mother, Siblings, My Wife and Loving Memory of My Father



C.U.S.T.

**CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD**

CERTIFICATE OF APPROVAL

This is to certify that the research work presented in the thesis, entitled “**An Automated Approach to Model Based Testing of Multi-agent Systems**” was conducted under the supervision of **Dr. Aamer Nadeem**. No part of this thesis has been submitted anywhere else for any other degree. This thesis is submitted to the **Department of Computer Science** in partial fulfillment of the requirements for the degree of Doctor in Philosophy in the field of **Computer Science, Department of Computer Science, Capital University of Science and Technology**. The Open defence of the thesis was conducted on **10 March, 2017**.

The Examination Committee unanimously agrees to award PhD degree to Mr. Shafiq ur Rehman in the field of Computer Science.

Student Name : Mr. Shafiq ur Rehman

Examination Committee :

(a) External Examiner 1: Dr. Muhammad Zohaib Zafar Iqbal,
Associate Professor
FAST -NU Islamabad

(b) External Examiner 2: Dr. Onaiza Maqbool,
Assistant Professor
QAU, Islamabad

(c) Internal Examiner : Dr. Nayyer Masood
Professor
CUST, Islamabad

Supervisor Name : Dr. Aamer Nadeem
Associate Professor
CUST, Islamabad

Name of HoD : Dr. Nayyer Masood
Professor
CUST, Islamabad

Name of Dean : Dr. Muhammad Abdul Qadir
Professor
CUST, Islamabad



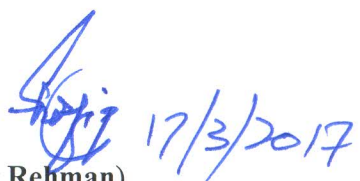
C.U.S.T.

**CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD**

AUTHOR'S DECLARATION

I, **Mr. Shafiq ur Rehman (Registration No. PC103005)**, hereby state that my PhD thesis titled, '**An Automated Approach to Model Based Testing of Multi-agent Systems**' is my own work and has not been submitted previously by me for taking any degree from Capital University of Science and Technology, Islamabad or anywhere else in the country/ world.

At any time, if my statement is found to be incorrect even after my graduation, the University has the right to withdraw my PhD Degree.


(Mr. Shafiq ur Rehman)

Dated:

March, 2017

Registration No : PC103005



C.U.S.T.

**CAPITAL UNIVERSITY OF SCIENCE & TECHNOLOGY
ISLAMABAD**

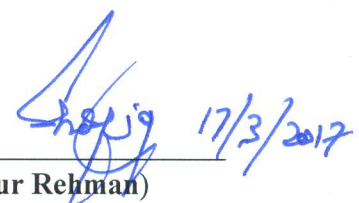
PLAGIARISM UNDERTAKING

I solemnly declare that research work presented in the thesis titled “**An Automated Approach to Model Based Testing of Multi-agent Systems**” is solely my research work with no significant contribution from any other person. Small contribution/ help wherever taken has been duly acknowledged and that complete thesis has been written by me.

I understand the zero tolerance policy of the HEC and Capital University of Science and Technology towards plagiarism. Therefore, I as an author of the above titled thesis declare that no portion of my thesis has been plagiarized and any material used as reference is properly referred/ cited.

I undertake that if I am found guilty of any formal plagiarism in the above titled thesis even after award of PhD Degree, the University reserves the right to withdraw/ revoke my PhD degree and that HEC and the University have the right to publish my name on the HEC/ University Website on which names of students are placed who submitted plagiarized thesis.

Dated: March, 2017



(Mr. Shafiq ur Rehman)
Registration No. PC103005

Acknowledgements

First of all praise be to Allah, The most gracious, Who blessed me with the opportunity, capability and resources to pursue the doctoral program, and it's all due to His grace that I saw it through.

One of the most notable of Allah's blessings upon me was in the form of my supervisor. I would like to thank my supervisors, Associate Professor/HOD BI & BS, Head of Center for Software Dependability research group Dr. Aamer Nadeem for the support and guidance he have given me throughout my candidature. I don't have words to thank you, Dr. Aamer Nadeem, for the motivation, guidance, support and encouragement that you provided to me on constant basis. I am also grateful to Dr. Muddassar Azam Sindhu, Assistant Professor QAU, who had reviewed my thesis write up. His reviews help me a lot to improve thesis write up. I feel very lucky to be part of Center for Software Dependability research group members whose discussion and constructive criticism maintained an environment that was conducive for research. More over without recreational activities of our CSD research group I may have gone insane over the last few years.

I would like to thank my family specially my wife Tehreem Jahangir, who always played a very supportive role, boost me up whenever I feel down. I have profound gratitude for her who had to face my educational stress. May Allah reward you for your sacrifice and selflessness. I am thankful to my brothers, who were always there for me since my childhood. There is one person who had set the base of my educational career that is my uncle Ch. Mehmood Akhtar, without his motivation I would not be able to achieve such an excellence in education.

I am very thankful to my friends, especially Bilal Bashir, who was always there to take me out of stress and motivated me many times. He had more confidence in me than I had in myself. There are so many other well wishers including friends, colleagues and relations who remembered me in their prayers. Allah bless you all. In the end I must mention that the being who cares more about me than I do for myself, my mother. I always felt her prayers by my side in the time of despair and frustration, and this feeling gave me energy to put myself together.

Credits

Research work appeared in the following publications:

- [1]. **Shafiq Ur Rehman** and Aamer Nadeem, “Interaction Testing of Multi-agent Systems Using Design Models”, Under Revision.
- [2]. **Shafiq Ur Rehman**, Aamer Nadeem, M. A. Sindhu, “Towards Automated Testing of Multi-agent Systems using Prometheus Design Models”, The International Arab Journal of Information Technology, Accepted in September 2016.
- [3]. **Shafiq Ur Rehman** and Aamer Nadeem, “An Approach to Model Based Testing of Multiagent Systems”, Special issue on Research and Development of Advanced Computing Technologies, 925206, 2014, The Scientific World Journal, Hindawi Publishing Corporation.
- [4]. **Rehman, S.U.**; Nadeem, A, “Testing of Autonomous Agents: A Critical Analysis”, IEEE Electronics, Communications and Photonics Conference (SIECPC), 2013 Saudi International , vol. 1, no. 5, pp. 27-30 April 2013, doi:10.1109/SIECPC.2013.6550990.
- [5]. **SU Rehman**, A Nadeem, “AgentSpeak (L) bases testing of autonomous agents”, T.-h. Kim et al. (Eds.): ASEA/DRBC/EL 2011, CCIS 257, pp. 11–20, 2011. © Springer-Verlag Berlin Heidelberg 2011.

Table of Contents

CHAPTER 1: INTRODUCTION.....	1
1.1. Prometheus Methodology.....	3
1.2. Research Aims and Objectives.....	4
1.3. Research Questions.....	4
1.4. Problem Statement.....	5
1.5. Research Contributions.....	6
1.6. Thesis Outline.....	8
CHAPTER 2: BACKGROUND	9
2.1 Agents	9
2.2 Belief Desire Intension (BDI) Architecture	10
2.3 Agent Unified Modeling Languages (AUML)	11
2.4 Agent Oriented Software Engineering Methodologies.....	11
2.4.1 GAIA Methodology.....	12
2.4.2 MaSE Methodology.....	12
2.4.3 MESSAGE Methodology	12
2.4.4 CoMoMAS and SODA Methodologies.....	12
2.4.5 DESIRE Methodology	13
2.4.6 Tropos Methodology.....	13
2.4.7 Prometheus Methodology	13
2.5 Software Testing.....	15
2.5.1 Test Automation	16
2.6 Model Based Testing	17
2.6.1 Derivation of Test Cases from a Test Model.....	18
2.7 Levels of MAS Testing.....	18
CHAPTER 3: LITERATURE SURVEY AND ANALYSIS.....	20
3.1 Multi-agent System Testing.....	20
3.2 Analysis.....	29
3.2.1. Evaluation Criteria.....	29
3.3 Summary	32

CHAPTER 4: A FAULT MODEL FOR MULTI-AGENT SYSTEMS	34
4.1 Fault Model.....	34
4.1.1 Fault Types: Interaction Faults.....	35
4.1.2 Faults Types: Goals, Sub-goals and Plans Faults	38
CHAPTER 5: MULTI-AGENT SYSTEM TESTING FRAMEWORK	43
5.1 Multi-agent System Testing Framework	43
5.2 MAS Integration Testing: Interaction Testing.....	44
5.2.1 PROTOCOL GRAPH GENERATION	46
5.2.2 TEST COVERAGE CRITERIA	50
5.2.3 TEST PATHS GENERATION	53
5.2.4 TEST CASE GENERATION	54
5.2.5 TEST RESULT EVALUATION: INTEGRATION TESTING.....	55
5.3 System Level MAS Testing Framework and Process: Goal and Plans Faults Identification & Coverage:.....	55
5.3.1 TEST MODEL GENERATION.....	57
5.3.2 COVERAGE CRITERIA	62
5.3.3 TEST PATHS GENERATION	64
5.3.4 TEST CASE GENERATION AND EXECUTION	66
5.3.5 TEST RESULT EVALUATION: GOAL-PLAN COVERAGE	68
CHAPTER 6: MAS TESTING FRAMEWORK IMPLEMENTATION.....	70
6.1 Interaction Testing	70
6.1.1 ATL Transformation Rules PD meta-model to PG meta-model	71
6.1.2 Protocol Graph Test Paths Generator.....	72
6.1.3 Interaction Test Paths Generator Tool.....	73
6.2 System Testing: Goal-Plan Graph Generator	76
6.3 Goal-Plan Graph Test Paths Generator.....	77
6.3.1 Goal-Plan Graph Test Paths Generator Tool.....	78
6.4 Test Case Execution.....	83
CHAPTER 7: EVALUATION: RESULTS AND DISCUSSION.....	86
7.1 Evaluation: Case Study.....	86
7.1.1 DESIGN ARTIFACTS.....	86

7.2 Faults Seeded in MAS implementation	93
7.2.1 FAULT TYPE SEEDED: INTERACTION FAULTS.....	93
7.2.2 FAULT TYPE SEEDED: GOAL, SUB-GOAL AND PLAN FAULTS	94
7.3 Test Model Generation from design artifacts	95
7.4 Test Cases Generation And Execution (Coverage Criteria Based)	97
7.4.1 INTERACTION TEST PATHS GENERATION	97
7.4.2 GOALS AND PLANS TEST PATHS GENERATION.....	98
7.4.3 TEST CASE GENERATION AND EXECUTION	101
7.5 Fault Identification By Test Cases	105
CHAPTER 8: CONCLUSIONS AND FUTURE WORK.....	112
APPENDIX -A.....	114
A-1: Code for Tests Path Generation Tool: Goal-Plan Paths	114
APPENDIX-B	118
Test Cases Execution Log.....	118
Sample Interaction Testing Log:.....	118
Sample Goal-Plan Coverage Log:	120
REFERENCES	123

LIST OF FIGURES

2.1	Phases and artifacts of The Prometheus Methodology.....	14
2.2	Fault, Error and Failure Relation.....	16
2.3	Model Based Testing Application Fields	17
4.1	MAS Working and Flow of Information	35
4.2	Protocol Diagram in PDT with Interactions	36
4.3	Warnings Protocol of Meteorological Alerting System	37
4.4	Goal Plan Relationship in PDT	39
4.5	A Plan Descriptor in PDT	40
5.1	MAS Testing Framework	44
5.2	Testing Framework for Interaction Testing	45
5.3	Data Retrieval Protocol Diagram	47
5.4	Meta-model of Protocol Diagram	48
5.5	Meta-model of Protocol Graph	49
5.6	Protocol Graph for Data Retrieval Protocol Diagram	50
5.7	Test Coverage Criteria Hierarchy.	53
5.8	Overview of Testing Process for Goal and Plans Coverage	56
5.9	Testing Framework for Goal, Sub-Goals and Plans Coverage	57
5.10	Example of Scenario and Goal overview diagram (Obtain Data Scenario).....	58
5.11	Goal Overview Diagram of Meteorological Alerting System	59
5.12	MAS Notations Used in Prometheus Design Diagrams	59
5.13	Agent overview diagram (AirPort Agent).....	60
5.14	Goal-Plan Graph (Test Model) of MAS	61
5.15	Test Case Generation Process for MAS	67
6.1	Protocol Graph and Test Path Generator Tool Architecture	70
6.2	Code of Finding Paths from Protocol Graph	72
6.3	Test Path Generator Tool Process	73
6.4	Test Path Generator Tool Input File	73

6.5	Test path Generation Tool (Coverage Criteria)	74
6.6	Test path Generation Tool (Action Coverage)	74
6.7	Test path Generation Tool (Percept Coverage)	75
6.8	Test path Generation Tool (Message Action Coverage)	75
6.9	Test path Generation Tool (Percept Message Coverage)	75
6.10	GPG (Test Model) Test Paths Generation Tool Input	79
6.11	Automatic Test Path Generation Architecture using GPG	79
6.12	GPG Test Paths Generator (Scenario Coverage Criteria)	80
6.13	GPG Test Paths Generator (Plan Coverage Criteria)	80
6.14	GPG Test Paths Generator (Loop Coverage Criteria)	81
6.15	GPG Test Paths Generator (Goal-Plan Coverage)	81
6.16	GPG Test Paths Generator (All Goals Coverage)	82
6.17	GPG Test Paths Generator (Capability Coverage)	82
6.18	GPG Test Paths Generator (Agent Coverage)	83
7.1	Scenario Diagram of Multi Currency Banking MAS.	87
7.2	Goal Overview Diagram of MAS	88
7.3	System Overview Diagram of Multi-agent System	88
7.4	AUML Description of Account Operation Protocol	89
7.5	Account Operation Protocol Diagram	90
7.6	BankAccount Agent overview diagram of MAS	91
7.7	DebitAccountCap and CreditAccountCap Capability overview diagram of MAS ...	91
7.8	CurrencyExchange Agent, Communicator Agent and ComputeRate Capability Overview diagrams	92
7.9	Protocol Graph (Test Model) for Account Operation Protocol Diagram	95
7.10	Goal-Plan Graph (Test Model) for Multi Currency MAS system	96
7.11	Test Path Generator Tool Input File	95
7.12	Chart with Test Cases and Coverage Criteria Detecting Types of Faults	109
7.13	Chart of Test Cases and Coverage Criteria Detecting Goal-Plan Faults	111

LIST OF TABLES

Table-1	Comparison of Technique Based on Parameters.....	31
Table-2	Test Paths for Data Retrieval Protocol Graph	53
Table-3	Test Paths for Each Coverage Criteria Applied on GPG	65
Table-4	Node Description Table for Paths Nodes	66
Table-5	Injected Faults for Interaction Testing.....	93
Table-6	Injected Faults in Multi-Currency Banking MAS.....	94
Table-7	Test Paths for Account Operation Protocol Diagram	97
Table-8	Structures of the Goal-Plan Graphs used as Input to Tool	98
Table-9	Test Paths for Goal-Plan Graph	99
Table-10	Node Description Table Interaction Test Paths Nodes	101
Table-11	Node Description Table for Goal-Plan Test Paths Nodes	102
Table-12	Test Cases for MAS testing	107
Table-13	Interaction Fault Types Vs Coverage Criteria	108
Table-14	Goal-Plan Fault Types Vs Coverage Criteria	108
Table-15	Detected Interaction Faults by Coverage Criteria and Minimum Test Cases ..	109
Table-16	Detected Faults by Coverage Criteria and Minimum Test Cases Required ...	110

LIST OF ABBREVIATIONS

AOSE	Agent Oriented Software Engineering
PDT	Prometheus Design Tool
AUML	Agent Unified Modeling Language
MBT	Model Based Testing
SUT	System under Test
MAS	Multi-agent Systems

ABSTRACT

Multi-agent systems (MAS) have been used progressively more for complicated and dynamic environments. Complex environments require MAS applications to work efficiently. MAS are used in dynamic and complex environments like e-commerce, banking, air traffic control, information management due to agents' unique features like autonomy to make their own decisions, reactivity upon environmental changes, social ability and pro-activeness in goal-directed behaviors to select feasible plans based on current situation.

Software testing plays a vital role in ensuring multi-agent systems' quality and acts as a major phase in their development life cycle. There is a need to have a good testing technique in order to ensure MAS quality. Testing based on extracting test requirements from system models is useful for revealing faults in MAS testing because testing can start early, we do not need to wait for complete system to develop. In literature, some work has been done on testing only a few features of MAS using model based testing (MBT), i.e., testing MAS units using system models and testing only one type of interaction between agents in MAS. There is a lack of a comprehensive testing technique which assures aspects ranging from system specification level to detailed plans execution covering integration and system level testing. Existing model based testing techniques for multi-agent systems do not cover all aspects of MAS, e.g., dependencies between interactions and goal plan coverage. Goals and plans related existing testing techniques only cover part of plans and goals in a specific design artifact. Greater coverage of design artifacts ensures higher fault detection capability.

Prometheus is a well-developed MAS designing methodology based on Agent Unified Modeling Language (AUML) notation. Interactions between agents in Prometheus methodology have actions, percepts and message interactions between agents; but only message interactions have been covered in existing techniques which is not enough for fault free MAS because action and percept interactions are also equally important. Dependency fault occurs in case of missing percept as percepts are required for events. Actions are used to update output of agent to environment. Messages usually depend on the correct sequential execution of related actions and percepts involved in an interaction.

Goals and plans are the key premise to achieve MAS targets. Different types of faults can occur if certain plans, goals, sub-goals or their order of execution is incorrect. Literature covers faults like incorrect belief and incorrect context etc., but there are certain aspects of MAS that are still

missing and can cause MAS to behave unexpectedly, like inaccurate goal achievement, plan failure, internal agent fault, missing functionality and scenario related faults. Such faults can be minimized by ensuring maximum coverage of goals and plan using design artifacts.

We have developed an approach using Prometheus design artifacts for integration and system level testing of MAS. AUML interaction protocol is used for interaction between agents and environment which is further elaborated in process diagrams corresponding to each agent. Fault models for interaction coverage and goal-plan coverage have been presented in which different integration and system level fault types are discussed. In this novel approach different interactions are considered like percept, action and message between the agents which can be modeled in a test model, i.e., protocol graph. Different coverage criteria for interaction coverage have been devised and applied to generate test paths for interactions between agents. For system level testing, we have also created test model, i.e., Goal-Plan Graph (GPG) for goals, sub-goals and plans using Prometheus design artifacts, i.e., goal overview diagram, scenario overview, agent and capability overview diagrams. We have defined coverage criteria for system level testing and applied on Goal-Plan Graph for test paths generation. Test cases have been generated using test model paths and its relevant implementation in agent development environment. Test paths are generated automatically with the help of tool from protocol graph and Goal-Plan Graph. We executed test cases on MAS implementation and compared expected results with the actual results to evaluate test cases. Failed test cases are further investigated to identify which type of fault was detected. We seeded faults in MAS and applied interaction and system level test cases. Expected results were gathered manually for evaluation purpose.

CHAPTER 1: INTRODUCTION

Multi Agent Systems (MAS) have been adopted widely in complex systems due to agents' unique features like reactivity, proactivity, autonomy and social ability (Padgham & Winikoff, 2004). MAS are composed of software agents which are software components that have autonomous actions to achieve assigned goals in a specified environment (Wooldridge, 2002).

Agent refers to a computer program that is situated in an environment where it can perceive input, do some processing and perform actions according to the assigned goals. Agents are able to interact with each other in order to perform certain tasks (Padgham & Winikoff, 2004). Autonomy is the agent's ability to operate independently as they make their own decisions, without the need for human guidance or intervention (Winikoff & Cranefield, 2010). Agents are robust and have the capability to operate even if the environment changes. Agents are programmed to perform their steps automatically in order to achieve certain goals. All of their activities converge towards achieving their defined goals in any possible way (Wooldridge, 2002). Agents have the potential to deal with complex and dynamic environments. All these features of agents and MAS pose challenges that must be handled and tested before MAS goes into operation.

Applications of multi-agent systems are in many domains like e-commerce, banking, air traffic control, information management. There are certain commercial agents' applications presented in (Munroe et al., 2006) which show the sensitivity of agent applications as they are meant to solve real life problems in almost every domain. Multi agent systems should have confidence of performing and achieving assigned goals. To gain confidence to rely on a multi-agent system, it must be properly tested. Testing of MAS is an important and complex task as agents possess dynamic, goal directed and proactive behavior by choosing plans based on current situation (Padgham & Winikoff, 2004, Wooldridge, 2002). Padgham and Winikoff show that MAS provide great flexibility, with over a million ways to achieve a given goal using only a relatively small hierarchy of goals and plans (2004). Because agents are autonomous and flexible, MAS can be difficult to test. Real-time response and dynamism make testing of such applications very difficult. Performance and accuracy of results must be checked and this can be achieved with the effective testing of MAS applications.

Software testing is an important and critical part of software development life cycle (SDLC). It consumes a major portion of development life cycle (Taipale et al., 2005) i.e., time, money and

effort etc. Testing is aimed at finding inconsistencies between the system's expected output and actual output (IEEE, 1998). We need to have a good testing technique in order to ensure MAS quality. Software testing aims to identify bugs in SUT to raise its quality and to ensure its correctness and usefulness. MAS testing can be performed at unit, integration and system level. In MAS testing, unit test includes testing of block of code or testing of individual agents. Unit testing of individual agents has been performed by (Zhang, Thangarajah & Padgham, 2007 & 2011). Integration and system levels are important as agents have to interact with each other and achieve MAS assigned goals. Integration testing of MAS includes testing the agent interaction and communication described in protocol diagram. We use term interaction testing to test MAS interactions defined in protocol diagram. There is some work done on integration testing by (Miller, Padgham & Thangarajah, 2010) but there are certain aspects, e.g., percept and action interactions were not covered or tested. Percepts environmental information required for events, used to trigger plans. Actions are used to update output of agent to environment. Messages are communication medium between agents. System testing includes testing the system as a whole. Testing the goals specified in system analysis phase and their relevant plans defined in detailed design phase.

Model based testing (MBT) uses system models to generate tests for system under test (SUT). Model represents an abstraction of the actual system. Design artifacts exhibit rich information of a multi agent system and its internal working. Testing based on extracting test requirements from system models are useful for revealing faults in MAS. Model based testing is a testing strategy in which model of the system are used to derive test cases for the SUT. Models are used to test multi agent systems by constructing test models of the actual working system. MBT has several advantages like test model can depict system functionality and it helps to show any deficiencies in MAS implementation from its design, test creation process can start early. No need to wait for complete system implementation. Test paths are identified from test model. One design diagram can be chosen to transform into a test model or a test model may be created from multiple diagrams. Coverage criteria forces the execution of certain path(s) based upon the criteria. Coverage criteria are often created on test model and model based testing can be applied to all levels from unit to system testing. Coverage criteria can be defined on the test model ensuring complete coverage of interactions, goals and plans for MAS. Coverage has a significant

relationship with fault detection. More coverage ensures more fault detection. Coverage criteria ensure certain types of faults detection and identification within a system (Tian, 2001).

1.1. Prometheus Methodology

There are many agent development methodologies in which agent based systems can be modeled, one of the detailed methodologies is Prometheus (Padgham & Winikoff, 2003) that is extensively in use since more than a decade. Prometheus agent oriented software engineering methodology has a well developed process from system specification to architectural design and then detailed design leading easily to code. Prometheus methodology has three views (1) dynamics (2) graphical overview and (3) structural forms (Padgham, Thangarajah & Winikoff, 2014). 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 associated with them. 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 (Padgham & Winikoff, 2003). Artifacts in all three phases are linked together and propagated to next level, i.e., system specification artifacts appear in architectural design and architectural phase artifacts propagates in detailed design phases where more details can be added as required. One can model the MAS using the Prometheus methodology starting from system specification to detailed design. Each agent has its own defined goals to achieve and together they work to achieve main goal. Accordingly different plans are associated with different goals (Padgham, Thangarajah & Winikoff, 2006). Prometheus also supports the design phase via tool named Prometheus Design Tool (PDT) (Thangarajah, Padgham & Winikoff, 2005) in which design activities can be modeled. Based on the richness of Prometheus methodology, we have used Prometheus design artifacts for test model generation. Agent UML (AUML) extends the UML for designing agents in MAS. AUML notations are used to design agents and their functionality in Prometheus methodology. PDT generates skeleton code for JACK (Winikoff, 2005), a fine reputable agent development environment that has been used for large commercial and complex applications development. There is a need to address quality assurance issues in multi-agent systems designed in Prometheus methodology.

1.2. Research Aims and Objectives

Our first aim is to perform integration testing of MAS covering all interactions like action, percept and messages. Interaction protocol in Prometheus methodology is used for interaction between the agents and actors. Interaction protocol in architectural design is captured by interaction pattern/sequence between the agents in a certain scenario. These interactions occur between agents and actors in the form of messages, actions and percepts. For a multi-agent system to perform correctly; these interactions must be tested and their occurrence in protocol must be verified with test cases. Leaving some interactions untested can cause many faults to occur like dependency, operational and synchronization between activities.

Another aim is to test goals, sub-goal and plans coverage to satisfy system testing of MAS. 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., 2014), (Thangarajah, Sardina & Padgham, 2012) and (Duff, Thangarajah & Harland, 2014) but goal-plan coverage and their coverage criteria definition work has not been performed by existing techniques. Although some work has been done in (Thangarajah, Jayatilleke & Padgham, 2011) covering only single scenario, but 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, contains features that must be covered for reliability. We can capture the relationship between goals and plans of an agent by a goal-plan diagram. Non coverage and non execution of any goal and plan can cause problems in MAS to achieve its objectives.

1.3. Research Questions

Based on MAS testing and aims, we have four research questions which we will cover in our testing framework.

- Which AUML design artifacts can be used to perform interactions and system testing of MAS?

This involves illustrating which design artifacts of Prometheus methodology are chosen to be used to test the different agent interactions like action, percept and messages? Each interaction is carried out to meet some defined objective. Which design artifact suits best to

cover all type of interactions and system level aspects in MAS? Whether the chosen design artifact is rich enough to be used to test MAS?

- How AUML design artifacts are used to perform interactions and system testing of MAS?
After choosing design artifacts whether we use the design artifact as it is or we need to convert it into test model? How model is used for testing MAS?
- How can the process of generating tests be guided by coverage criteria and detecting faults?
Whether it is possible to define coverage criteria on system model/test model? Can we define coverage criteria on system model/test model? How coverage of interaction and system aspects can enhance fault identification? What coverage criteria will be best for MAS testing as system possesses dynamic behavior?
- What Types of interaction and system level 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? How fault model ensures possible occurrence of faults in MAS? What types of faults remain in MAS if certain interactions, goals and plans are missed?
- 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?

1.4. Problem Statement

Based on research questions and MAS testing challenges, there is a need to ensure integration and system level testing of MAS. In MAS design artifacts, protocol diagram uses actions, percepts and messages between agents and actors for interactions. There is a need to test all interactions occurring between the agent and actor in a protocol covering actions, percepts and messages. In the available literature, only message coverage has been performed so far which is not enough to guarantee fault free MAS because several problem can occur when actions and percepts are missed in test coverage. Dependency faults can be present in the case of missing percept, as a percept is required to trigger the event that will cause a plan to start in order to achieve its goal. Missing action can cause operational faults as output of an agent will not be

conveyed to its external environment. A message is always dependent on some actions and percepts for execution before its content is passed between agents. Problem can occur when any of an action or a percept is not executed, possibly conveying wrong message content to other agent(s). There is no technique which covers all MAS interactions occurring in an interaction protocol in the form of action and percept neither any coverage metrics have been defined for them.

Coverage of goals and sub-goals identified at specification level, used in scenario overview diagram and their relation with plans, seems missing in literature. To fulfill any goals there are interactions in MAS followed by capability overview diagram of an agent in Prometheus methodology. Capability overview diagram contains different plans that are executed to achieve any goal. Coverage of goal, sub-goals and plans is important for MAS to achieve its system level objectives. Non coverage of any of goals, sub-goals and plans can cause many faults to prevail in MAS like in-accurate goal achievement, plan failure, missing functionality and scenario faults etc. No fault model currently exist which covers goal and plans related faults for MAS that should be revealed in testing. Test case generation from test paths and test case execution to identify interaction and goal-plan related faults are also missing in literature for MAS testing.

1.5. Research Contributions

Our research contribution for model based testing of MAS focusing on interaction and system level is summarized as follows:

- We have conducted comprehensive literature survey of existing approaches for interactions and system testing of MAS. We have published our survey in (Rehman & Nadeem, 2013) and details have been presented in chapter 3.
- We have defined fault models for both interaction and system testing. System testing included goal, sub-goals and plans coverage and testing. Possible faults that could occur in MAS have been presented in fault models.
- We have identified design artifacts for model based testing of MAS. We have converted design artifacts into test models via algorithms defined in chapter 6. We have two test models, i.e., protocol graph and Goal-Plan Graph for interaction and goal plan coverage.
- We have devised coverage criteria on both test models and identify possible test paths with respect to coverage criteria. Coverage criteria for both types of test models have

been devised and test path are calculated automatically with the help of tool. Details are presented in chapter 5 and 6. We have derived test cases from test path generated from test models. Test cases structure and details of test case for MAS testing have been presented in chapter 7.

- We have performed the validation of our testing framework with the case study. We have seeded faults in MAS implementation in JACK and run our test cases derived from test model paths. Test result evaluation shows the effectiveness of test case to identify seeded faults. Failed test cases are further investigated to identify reason of failure, e.g., certain interaction has not been covered or certain plan has not been triggered causing a fault to occur in MAS.

Our purpose in this research is to perform interaction testing and system testing of MAS. Interaction testing uses interaction protocol to perform message, actions and percepts interactions coverage. Goals and sub goals coverage is vital to MAS system testing. Plans contain the steps to fulfill goal completeness. Goals are defined and plans are triggered to achieve desired goals. There are many ways to achieve a certain goal. Goal 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 almost maximum aspect of MAS coverage. Coverage criteria can ensure maximum testing adequacy, which in turn can identify faults which can remain undetected if certain goals and plans are not executed. JACK development environment (Busetta, et al, 1998) has been used for MAS implementation, which is then instrumented to evaluate goal and plans coverage approach to find injected faults.

Faults model have been defined to cover possible faults in MAS relating to interaction occurring in protocol diagram. Goal, sub-goals, plans execution and interactions related faults have also been presented in fault model. Fault model describes the faults which remain if some coverage has not been achieved. We have developed a tool which generates test paths based on specified coverage criterion that will test the interactions between the agents via some protocol as well as goal-plan coverage. In order to achieve a goal there can be interactions between the agents and between agents to actors as well. An agent achieves its goals with the help of plans specified. A main goal may have some sub-goals contributing their part in achieving the objective. Correct and ordered execution and achievement of goals and plans in MAS can assure its correctness. System models or design artifacts possess all the details about the functionality of interactions,

goals and plans in MAS and their working. We assume that design models are complete and specified requirements are properly propagated from specification to design. Since design faults are detected and handled by (Abushark et al. 2015). Therefore, model based testing (MBT) of MAS uses system models to generate tests for system under test (SUT). Protocol graph is used as the test model to test interaction testing of MAS while Goal-Plan Graph is used as a test model to test behavior of the MAS modeled in scenario, agent and capability diagrams of Prometheus methodology. Goals are executed by respective plan(s) in-order to achieve assigned goals. Based on the discussion an approach is necessary that can test an agent system effectively and efficiently. Subsequent chapters of this research thesis focus on defining testing framework, its processes and validation of our testing framework for MAS testing.

1.6. Thesis Outline

In the rest of thesis, Chapter 2 describes background and relevant information to this research including agent oriented software engineering (AOSE) methodology, agents and their role in multi-agent system, particularly Prometheus methodology (Padgham & Winikoff , 2003) in details with its benefits. Chapter 3 includes the related work done in testing of MAS. Each of the techniques for MAS interaction and system testing have been discussed and analyzed. Chapter 4 describes a fault model based on identified problem and short comings in existing work discussed in the previous chapter 3. Chapter 5 describes testing framework for model based testing of MAS for integration and system testing. Chapter 6 describes the implementation details of the testing framework. Chapter 7 describes results and discussion on the testing framework and validation. Chapter 8 concludes the thesis and provides possible future directions of research.

CHAPTER 2: BACKGROUND

This chapter describes the background and relevant information to this research including agents and their role in multi-agent system, Agent Unified Modeling Language (AUML), Agent Oriented Software Engineering (AOSE) methodologies, particularly focusing on Prometheus Methodology (Padgham & Winikoff, 2003) in detail with its potential benefits. This chapter also describes software testing and model based testing. Section 2.1 starts with a brief introduction to agents, their characteristics, behaviors and AOSE. Section 2.2 elaborates Belief Desire Intension (BDI) Architecture. Section 2.3 provides a brief introduction to AUML and how it is used? Section 2.4 describes AOSE methodologies, along with other relevant methodologies with potential benefits of choosing Prometheus methodology. Section 2.5 provides a brief introduction to software testing, test paths and test cases. Finally, model based testing and how a design model can be used for testing is discussed in Section 2.6.

2.1 Agents

An agent refers to a computer program that is situated in an environment where it can perceive input, do some processing and can perform actions according to the assigned goals (Padgham & Winikoff, 2004). Autonomy is the agent's ability to operate independently, without the need for human guidance or intervention (Alonso et al., 2009). Wooldridge and other researchers further distinguish autonomous agent's characteristics (2002). They describe that an autonomous agent must be:

- Pro-active: agents have goal-directed behavior, is able to pursue its own goals even when the environment has changed.
- Autonomous: agents are independent and make their own decisions, without any direct intervention.
- Reactive: Agents have the ability to act upon environmental changes where they are situated.
- Social: agents interact with other agents and actors via some kind of communication protocol like protocol diagram in Prometheus methodology.

Rao and Georgeff (1995) further describe the agent's abilities as robust, flexible and situated. Agents should recover from failure due to environmental changes. Agents are situated in a certain environment to perform their designed goals.

2.2 Belief Desire Intension (BDI) Architecture

Agent architecture models the behavior of agents, one of which is Belief Desire Intension (BDI) architecture (Bratman 1987; Rao and Georgeff 1991). BDI agents have certain goals to achieve. Belief-Desire-Intention properties are used to program intelligent agents. BDI agents have been widely used since last the two decades and various researchers have explored their behavior. We discuss and use BDI agents in our research. We consider multi-agent systems developed by using Prometheus methodology. Padgham and Winikoff (2003) present Prometheus as an agent oriented methodology based on BDI agents. Two concepts that make the relationship between environment and agent are percepts and action. Percepts are information or state of environment an agent can receive and actions are performed by the agent to affect the environment.

BDI agents are elaborated in (Rao & Georgeff, 1995) in which agents have three states; beliefs, desires and intentions. Belief is the information the agent has about the environment and itself. Desires are the states or goals which an agent wants to achieve. Intentions are steps taken to achieve the desires or goals. We can say that intentions are plans which describe the steps of how to achieve the goals. There are many agent implementation platforms based on the BDI model, such as PRS (Ingrand, Georgeff & Rao, 1992), dMARS (D'Inverno et al., 1998), JAM (Huber, 1999), JACK (Winikoff 2005; Jack intelligent agents 2014), JADDEX (Pokahr, Braubach & Lamersdorf, 2003) and JASON (Bordini, Wooldridge & Hubner, 2007). Agent oriented software engineering (AOSE) is the term used for systems developed by using autonomous agents; such systems are called multi-agent systems.

Intentions are represented as plans while goals are presented as events which are used to trigger the relevant plan to achieve certain goal (Rao & Georgeff, 1995). A plan depicts the steps to achieve a certain goal. Agent invokes the relevant plan upon a triggering event from a set of available plans.

2.3 Agent Unified Modeling Languages (AUML)

Agent's designs are represented in graphical form by using Agent Unified Modeling Language (AUML) (Winikoff, 2005). AUML is based on UML 2.0 (OMG, 2003) and it is rich enough to provide an elegant graphical representation of agents using modeling notations for this purpose. UML 2.0 is extended to represent agent models. AUML is used to represent agent, their behavior and interaction among them. UML class diagrams can be used to represent the static view of agents. AUML is used to express Prometheus methodology design diagrams or artifacts. In AUML there is an interaction protocol; Interaction protocols can be specified in more detail (i.e., leveled or nested) using a combination of diagrams. Level I represents the overall protocol; level II represents interaction among agents; level III represents internal agent processing (Odell, Parunak & Bauer, 2000). Interaction protocol which is commonly presented in protocol diagram is used for interaction between agents and between agent and actor. Environment and actor both terms are used interchangeably. Protocol diagram is similar to UML sequence diagram. Means of communication between agents is message passing. Different notations are used to show different behaviors like alt, opt, loop, parallel etc. Lifeline shows the presence of participating agent in a certain activity like message passing etc. An extension of activity diagram is used for detail elaboration of protocol diagram. Each protocol diagram can have multiple process diagram represented by extended activity diagram. A Collaboration diagram is also used to depict interactions between agents responsible for different roles. State charts can be used to model individual agent behavior. Textual representation of AUML can also be made which shows all the details depicted in graphical representation (Winikoff, 2005).

2.4 Agent Oriented Software Engineering Methodologies

A methodology is a collection of activities used to develop the system. Additionally, a methodology can also be supported by a tool. There are several agent agent-oriented software engineering methodologies, e.g., Gaia (Generic Architecture for Information Availability) (Wooldridge, Jennings & Kinny, 2000), Multi-agent Systems Engineering (MaSE) (DeLoach, Wood & Sparkman 2001; DeLoach 1999), MESSAGE (Caire et al. 2001), Prometheus (Padgham & Winikoff, 2003), Tropos (Bresciani et al. 2002), CoMoMAS (Glaser, 1996), SODA (Societies in Open and Distributed Agent spaces) (Omicini, 2001), DESIRE (Brazier et al., 1997) and MAS-CommonKADS (Iglesias et al., 1998).

2.4.1 GAIA Methodology

Requirements are assumed to be known in Gaia (Wooldridge, Jennings & Kinny, 2000) methodology which forms the basis of analysis and design phases. GAIA is a methodology which distinguishes between analysis and design phases. It has Role Model and Interaction Model in analysis phase, and Agent Model, Services Model, and Acquaintance Model in design phase. Gaia does not address implementation of MAS using its models and hence is no tool support for it.

2.4.2 MaSE Methodology

MaSE (DeLoach 1999) is an extension of the object-oriented approach and has two phases' analysis & design. MaSe does not have the view that agents should be autonomous instead it assumes agents only as software which interacts with other software i.e. agents. Analysis contains three steps i.e. Capturing Goals, Applying Use Cases, and Refining Roles and design contains four steps i.e. Creating Agent Classes, Constructing Conversations, Assembling Agent Classes, and System Design (DeLoach, Wood & Sparkman, 2001). MaSE has limitation in terms of agent characteristic modeling criteria (Juneidi and Vouros, 2010).

2.4.3 MESSAGE Methodology

MESSAGE (Caire et al. 2001) adopts the life-cycle model of the Rational Unified Process (RUP) and is limited to analysis and design activities only. It uses UML as modeling language. It has five different views e.g. Organization view, Goal/Task view, Agent/Role view, Interaction view and Domain view (Caire et al. 2001). MESSAGE is weak in expressiveness and completeness (Dam, 2003).

2.4.4 CoMoMAS and SODA Methodologies

CoMoMAS (Glaser, 1996) focuses on knowledge engineering problem that arises in multi-agent systems and provides extension in Cooperation Modeling Language for agents. SODA (Omicini 2001) focuses on the social inter-agent aspects of agent systems and employs the concept of coordination models (Juneid & Vouros 201). Both do not have comprehensive tool support available.

2.4.5 DESIRE Methodology

DESIRE contains expertise model and agents (Brazier et al., 1997). Once analysis phase has been done, DESIRE could be used for specifying the design and implementation of MAS.

2.4.6 Tropos Methodology

Tropos (Bresciani et al. 2002) is an AOSE methodology whose main distinction is the early requirements analysis (Dam 2003). Agent related concepts like goals, plans, and tasks are included in all phases. No detailed information is available for the last process defining agent types and mapping them to capabilities (Juneidi and Vouros, 2010). The methodology does not appear to provide heuristics for any phase (Dam, 2003).

2.4.7 Prometheus Methodology

MESSAGE, GAIA, Tropos are not easy to use and have some deficiencies (Dam 2003, Juneidi & Vouros 2010). MaSE is better than the ones mentioned earlier but it has limitation of agents modeling characteristic (Juneidi & Vouros, 2010).

The Prometheus methodology is a detailed AOSE methodology, which aims to cover all of the major activities required in developing multi-agent systems from specification to architectural design and detailed design (Padgham & Winikoff, 2003). Each of its phases is rich enough to capture design details of the MAS that lead to code generation. Details of each of its phase along with artifact details are explained below.

System Specification

As depicted in Figure 2.1, design begins with system specification in which actors are needed to be defined. Actors are entities external to MAS like environment etc. System specification contains three diagrams, i.e., analysis Overview, scenario and goal Overview. Analysis overview presents the interface of the system with its environment in which actors, scenario, percept, message and actions interaction show an overview of the MAS. Goal are defined and sub goals along with ‘AND’ and ‘OR’ relationships between goals are defined in goal overview diagram. Scenario contains actions, percepts and goals used in normal working like use case scenario of the system. Scenario can have sub scenarios as well. In Prometheus, each scenario has

corresponding goal overview diagram. A MAS has more than one functionalities or operations to perform. Such operations are called roles; each role has associated actions, percepts and goals. These roles are used in architectural design to form agent roles grouping.

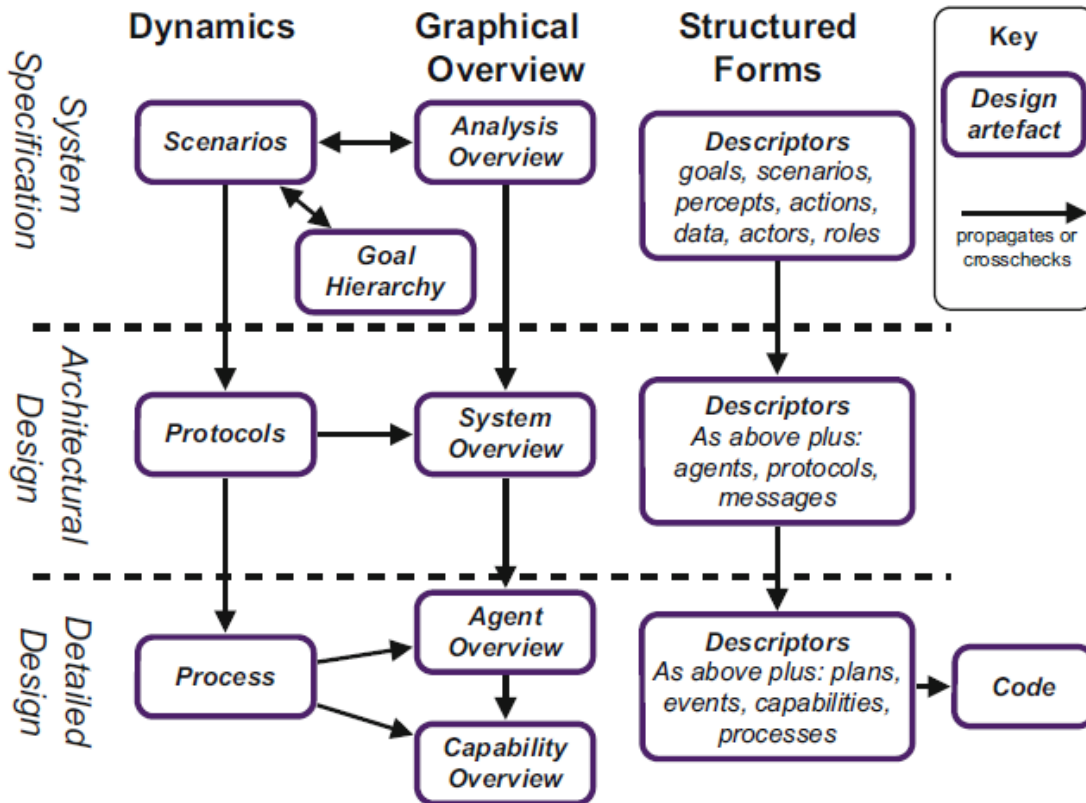


Figure 2.1: Phases and Artifacts of The Prometheus Methodology (Padgham, Thangarajah & Winikoff, 2014)

Architectural Design

Architectural design captures the agents' details, their dependencies and associated roles to perform. Each role/functionality needs to read or write into database and this behavior captures in data coupling. Dynamic behavior of MAS is modeled in System Overview diagram. System Overview diagram contains agents communicating through Interaction Protocol(s). Different message, action and percept interactions in MAS are presented through a protocol diagram. For communication between agents, messages are used. Action can be performed by agent to affect environment and environment state can be received through percepts. Messages are used between the agents.

Detailed design

Detailed design defines the internals of agents, capability overview, event, data and plan descriptors. Each Plan has its triggering event, context (precondition), associated incoming/outgoing messages, data used, associated goals relevant to a plan and its steps. Steps of plans trigger sub-goals. Process diagram is also defined in detailed design which is derived from protocol diagrams (Padgham & Winikoff, 2003). Process diagrams are actually agent overview and capability overview diagrams. Agent overview diagram captures complete working of an agent, plans used and data produced/used. Capability overview diagram presents module of an agent in which more than one plans can be used.

Each phase has its own supported symbols which are used to represent agent's behavior. Agents receive percept from the environment and communicate with each other through messages and actions are the changes that an agent can perform to the environment. All these phases have rich notation support which is available under Prometheus Design Tool (PDT) (Thangarajah, Padgham & Winikoff, 2005). One of the main advantages of Prometheus methodology is the availability of tool support, where consistency checking between various diagrams can be done. This methodology also has support of JACK (Winikoff 2005; Jack intelligent agents 2014) development environment where design of multi-agent system can be implemented. JACK is a Java based development environment used to implement the system.

2.5 Software Testing

Development of system contains several phases like requirements definition, design, implementation, testing and deployment. Software testing is one of the important phases of SDLC. We need to have a good testing technique in-order to ensure software quality. Testing can be very helpful in failure detection and validation of System under Test (SUT). According to a 1998 IEEE definition, software testing can be defined as: "Testing is the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software item".

We need to distinguish between a few terms with respect to testing e.g. fault, error and failure, test path, test case, test bed, test data. Fault, error and failure described by (Joe 1983, Joe 1990, Paul & Offutt , 2008) can be depicted in figure 2.2.

- Fault is static defect in a system like code error. It effects the system once it is executed otherwise it is of no harm to the system
- Error is wrong state of the system exposed during running program.
- If there is an error then failure can occur in the SUT.

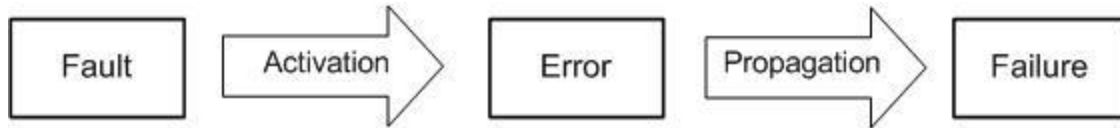


Figure 2.2: Fault, Error and Failure Relation

In testing a system, fault linking must be identified and reached with test data. Testing is referred to as validation and systematic observation of system behavior with given inputs (Paul, Offutt, 2008). Actual and expected behavior is checked in testing for which we need to define test cases. A test case is the sequence of steps to follow which executes SUT and records the actual output of the SUT. Test suite is the collection of test cases. Important activities in software test plan are to define testing strategy, choose system model in case of model based testing, identify expected behavior (identity test paths), generate test data to execute test paths and run test cases. Test data generation, test case execution and comparing expected and actual behavior continues until we achieve our desired goal. In general, we have White Box testing and Black Box testing techniques. Black Box testing focuses only on input and monitors expected behavior while White Box testing monitors internal program's control and data flow. Usually graph of program or design is generated on which flow of data and control can be monitored. We can define a fault model, in which possible faults that could occur in SUT are defined. For validation a fault is injected in the SUT and behavior is observed with test cases whether injected fault can be reached by our test case or not (Myers et al., 2009).

2.5.1 Test Automation

In test automation all or part of testing activity is done automatically through some program. Automated software testing is the need of time as complexity of software is growing day by day and it is becoming difficult to test it with manual approaches. Test automation focuses on automatic generation of test cases with test data to test the SUT.

2.6 Model Based Testing

Model based testing is a testing strategy in which model of the system can be used to define test cases for the SUT. A model can be of any form that could be used for testing like design document, AUML diagram specifically protocol diagram for multi-agent system (Zhang, Thangarajah & Padgham, 2009). Test model is chosen from SUT to generate test cases from it. Test model is simpler and smaller than the actual system but rich enough to have necessary details useful for testing of SUT (Zhang, Thangarajah & Padgham, 2009). During the design of system a series of diagrams are generated, same is the case with Prometheus methodology. One diagram can be chosen to act like test model or test model can be constructed from more than one design models. Coverage criteria are often created on test model and model based testing can be applied to all levels from unit to system testing, Figure 2.3 shows these details in graphical form (Jan 2004, Utting , Legeard, 2006).

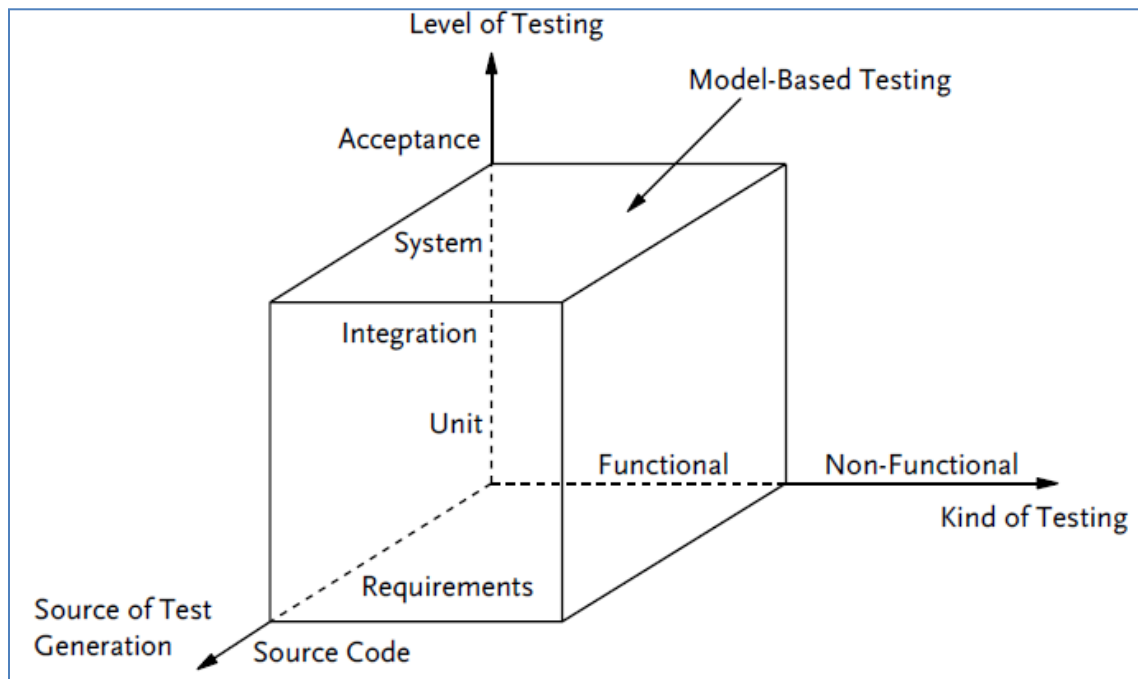


Figure 2.3: Model Based Testing Application Fields (Jan 2004, Utting & Legeard, 2006)

Model based testing plays an important role in such software which uses models in their analysis and design phases. Model based testing has several advantages to software testing like:

- Test model created from design diagram(s) is simple and rich enough to contain the required details. Test model is easy to understand for tester and it is also easy to maintain and trace functionality.
- In model based testing, one can start testing process early. No need to wait for complete system implementation. It will reduce cost as anomalies can be identified early. Requirements related faults can be identified and corrected early.
- Once test cases are created from test model then each test case has its relevant requirement traceability by using test model.
- Coverage criteria can be defined on test model ensuring complete coverage of all functionalities. Furthermore test cases can be created automatically from the test model with reference to coverage analysis.

This thesis focuses on identifying design diagrams which are used to create test model(s). Coverage criteria have been defined and applied on these test model(s). Test paths are generated from test model (s) and test cases are created from test model paths. Comparison of test paths and actual paths after test case execution has also been performed. Additionally, a fault model has been defined for MAS testing and seeded faults are successfully identified by test cases.

2.6.1 Derivation of Test Cases from a Test Model

In model based testing approaches test cases are derived automatically or semi-automatically from test model. Test cases can be generated from test model paths depending upon the details of information available in the test model.

2.7 Levels of MAS Testing

In this section, we discuss different testing levels of MAS and our focus on levels of testing considered in this thesis. There are three levels of MAS, i.e., Unit level, Integration level and System level. Unit testing includes testing of a block of code or individual units for an agent etc. Unit testing of individual agents has been performed by (Zhang, Thangarajah & Padgham, 2007 & 2011). Integration testing of MAS includes testing the agent interaction and communication described in the protocol diagram. There is some work done on integration level as well by (Miller, Padgham & Thangarajah, 2010) but there are certain aspects that were not covered e.g.

percept and action interactions. System level testing includes testing the system as a whole, by testing the expected outcome of the system against the given input. Our focus in this research thesis is on Integration and system level testing. In Integration testing Prometheus design artifacts, i.e., a protocol diagram will be used for interaction testing, while analysis phase and detailed design phase diagrams will be used for system level testing to test goal, sub-goal and plan coverage. In our testing framework we have generated a test model from design artifacts and test paths are generated automatically on our test models. Test cases are generated and executed to find possible faults in MAS.

We have elaborated all related concepts regarding testing of MAS. Literature work done so far in testing the MAS is elaborated and analyzed in next chapter.

CHAPTER 3: LITERATURE SURVEY AND ANALYSIS

The purpose of writing this chapter is to give a brief overview of the existing research carried out for testing the quality of MAS. Section 3.1 describes literature survey that will focus on the research being done in the areas of MAS. Our main focus is on Prometheus methodology. In the previous chapter we have discussed different methodologies and the reason of choosing the Prometheus methodology design to be used for test model construction. Section 3.2 will provide the drawbacks of existing techniques mainly focusing to test MAS designed using Prometheus methodology. We have conducted a survey on existing approaches that address the domain of testing the multi-agent system and model based testing approaches to multi-agent systems.

3.1 Multi-agent System Testing

To gain confidence in the working of MAS, it must be properly tested. Testing of a software agent is an important and critical task as agents possess dynamic behavior. Agents have a run time response and adaptability. Basic agent-oriented concepts, e.g., autonomy, social ability, proactiveness etc have been covered in previous chapter but there are several exceptions. Tropos was not perceived as being easy to use whilst MESSAGE and GAIA were both ranked weakly on adequacy and expressiveness. MaSE does not provide detailed design. Prometheus methodology is rich enough to provide detailed design and tool support as well for developers (Dam, 2003). There is a need to address quality assurance issues in multi-agent systems designed with Prometheus methodology. Proper coverage and testing of design artifacts can enhance quality. Coverage criteria for testing can be applied to both code and model (Spillner, 1995). Code base testing technique test that all code are covered in term of statements etc. while model based coverage requires the different interaction from different states of the system, represented in specific model (Utting & Legeard, 2007).

C.K. Low, TY. Chen, R. Ronnquist (1999)

Low et al. (1999) consider test coverage criteria for BDI agents (Low, Chen & Ronnquist, 1999). They derive two types of control-flow graphs: one with nodes, nodes representing plans for BDI agent and arcs representing messages or other events which initiate a certain plan. Several coverage criteria are defined, based on node, arc, and path coverage, as well as some based on the success or failure of executing statements and plans (Low, Chen & Ronnquist, 1999).

Different interactions between the modeling artifacts are not presented. Instead this approach is not considering interactions between agents.

J. Thangarajah, L. Padgham, and M. Winikoff (2003)

Thangarajah et al. (2003) presented the phenomenon of goals interactions between agents, specifically negative goal interaction in which one goal can cause another goal's unsuccessful execution. They have maintained summary information about requirements for goal to execute and scheduling which protects unordered execution of goals and plan steps (Thangarajah, Padgham & Winikoff, 2003). Their work does not target faults identification and complete coverage of goals and plans. No design model is considered for tracking the dependency between goals and plan, neither agent interaction is considered.

M. Nunez, I. Rodriguez, F. Rubio (2005)

Autonomous agents are specified and tested by using generic formal framework. Individual agents are tested and their behavior in multi agent system is also observed (Nunez, Rodriguez & Rubio, 2005). Main theme is to observe the behavior of the agent against the given input and validate its conformance with the specification. The authors present the idea of Utility state Machine (USM) in order to specify and test the autonomous agents (Nunez, Rodriguez & Rubio, 2005). The idea of USM is inspired by finite state machines representing states with predicate at each state that must be met, and set of transitions, start state, amortization time and maximal investment that the machine can afford (Nunez, Rodriguez & Rubio, 2005). User has different preferences represented by its utility functions at each state. Each USM has its configuration containing the pair of profit owned (either -ve or +ve) and its expiry time. Clear profit is the amount in hand after compensating all negative transactions or loss. Evolution may be a change of state, time passing or a transaction, where a valid trace refers to the traces that a USM may perform from a start state to its target or final state (Nunez, Rodriguez & Rubio, 2005).

Each test will test set of states of USM (Nunez, Rodriguez & Rubio, 2005). Each test has the resources to reach the IUT to final state and perform testing. Resources can be allocated, de-allocated and modify their amount respectively. USM also maintains the reserve of resources to be used to reach the final state (Nunez, Rodriguez & Rubio, 2005).

J. Thangarajah, J. Harland, and N. Yorke-Smith (2007)

Thangarajah et al. (2007) defined several criteria for agent consideration or discussion e.g. time varying utilization, deadline, resource requirement, dependencies, communication with other goals etc. Main consideration in their work is goal deliberation but faults that may occur in MAS are not identified. How AND, OR constrains between goals and plans coverage are not discussed.

Z. Zhang, J. Thangarajah, L. Padgham (2007)

Zhang et al. (2007) have presented an approach to perform unit testing of autonomous agents. Plan has been tested containing different items. Different possibilities of plans have been exercised. Different plan cycles have been tested by (Zhang, Thangarajah & Padgham, 2007). Main theme is to test the small units of autonomous agents. Plans are tested in an ordered way mostly using bottom up approach in plans hierarchy. Each variable is checked against its equivalent class which has five fields against each variable (Zhang, Thangarajah & Padgham, 2007). Plans are tested considering their cycle to achieve the goals. Technique presented does not possess the evolution capability. Code based testing approach has been used and individual plans are covered. No tool support is provided with this approach.

P. Shaw, B. Farwer, and R. H. Bordini (2008)

Shaw et al. (2008) presented theoretical and practical results of agent consideration by using goal plan tree. Main focus is to reason about both reusable and consumable resources using Petri-nets which also contains summary details of resources. Best plan is selected based on demanded resources (Shaw, Farwer & Bordini, 2008). Faults model along with coverage metrics for goal and plans is still missing.

Y. Zhou, L. van der Torre, and Y. Zhang (2008)

Zhou et al. (2008) presented an approach to study partial goal achievement. They have used propositional logic e.g. disjunctive proposition to show whether a goal is achieved or not. They introduced strong and weak partial implications and studied their semantics. Authors have considered the case where goal has been modified due to change in belief set (Zhou, Torre &

Zhang, 2008). They have not discussed the fault model. Action and impact of goal modification have not been analyzed.

D. Nguyen, A. Perini and P. Tonella (2008)

Nguyen et al. (2008) presented testing framework for the Tropos testing methodology. Goal oriented testing methodology represents internal and external level of testing. Authors mainly focus on internal level of testing including unit, integration and system testing. Test cases are derived from the requirements goal analysis. Goals are of different types including own goals and delegated goals (Nguyen, Perini & Tonella, 2008). Goals have elementary and intermediate relationships between themselves which enable towards achieving a particular goal. Sub-goals must be achieved first before going to achieve the root goal, which is composed into different sub goals (Nguyen, Perini & Tonella, 2008). Test suites are derived by looking at relationships between the goals. Test suite is presented by BNF style notations (Nguyen, Perini & Tonella, 2008). Nguyen etc al also presented an example of BibFinder MAS system used to retrieve bibliographic information. An agent acts as the tester agent which tests the search function of system under test having multiple sequences (Nguyen, Perini & Tonella, 2008).

Only goals are considered in this approach of testing. Other important dependencies like resource and plan are not considered. Internal structure of the plan is not exploited. Bibfinder case study has been presented. This technique is automatable as test structure is presented in OCL. Different types of relationships with each goal are used to generate test cases.

Z. Zhang, J. Thangarajah, and L. Padgham (2008)

Zhang et al. (2008) presented an approach for Model based testing for agent system. Testing framework caters the different sequence of agent program execution. Fault directed testing approach is used by first Identifying appropriate units of the agent and test the unit with the defined mechanism. It considers the plan as a single unit then it is checked whether the plan is triggered by the appropriate event or not, checks its precondition, cycles in plan and plan completeness etc. Event testing is performed for numbers of applicable plans for the event. An electronic bookstore system has been used as the sample system; testing framework will execute test units in a sequence (Zhang, Thangarajah & Padgham, 2008). No coverage measures have

been taken while considering interactions between agent and external agent or stub. We are considering interactions between multi-agent systems through coverage measures.

M. B. van Riemsdijk and N. Yorke-Smith (2010)

Riemsdijk and York-Smith (2010) presented an approach about partially-complete goal in belief desire, intention like agents. A metric has been used to capture progress for partial goal achievement. They have used a minimum completed value to declare that goal is completely achieved or not. Authors discussed agent demonstration, but no thorough computational means are not discussed (Riemsdijk & Yorke-Smith, 2010). No such detail has been discussed to show whether goal will be satisfied or not instead only progress is considered with reference to goal achievement. No fault model is presented if certain goals are not achieved or plans for the goals are not triggered.

T. Miller, L. Padgham , J. Thangarajah (2010)

Miller, Padgham & Thangarajah (2010) state that the interaction between the agents possesses complex behavior and therefore testing of interactions is important. They defined two sets of test coverage criteria for multi-agent interaction testing. The first uses only the protocol specification, while the second considers the plans that generate and receive the messages in the protocol (Miller, Padgham & Thangarajah, 2010). Authors did not consider percepts and actions in capturing the interactions. Neither test data generation has been neither done nor automatic test case generation for path coverage.

M. Winikoff, S. Cranefield (2010)

Winikoff and Cranefield (2010) have analyzed the size of behavior space for BDI agent and found that failure handling has larger impact on size of behavior space than expected (Winikoff & Cranefield, 2010). They have identified different factors that influence the size of behavioral space. Goal plan tree has been discussed which has been modeled by goals and plans in the form of tree. Failure handling has been introduced in context of agent's behavioral space (Winikoff & Cranefield, 2010). Goal-plan tree has also number of successful and unsuccessful execution paths or traces in both failure handling and without failure handling cases. Probability of failure is checked in a particular trace.

Behavioral space is very large in which failure handling makes significant difference. Comparison with an industrial application has also been made to check the effectiveness of the technique. No tool support is discussed with the approach. The author generally discusses different testing aspects for the BDI agents instead of providing one concrete testing approach. Goal-plan tree has been used to present goals and technique does not have the evolution capability. Both techniques above do not consider interactions between agents neither any coverage measures have been taken even in unit testing.

J. Thangarajah and L. Padgham (2011)

Thangarajah and Padgham (2011) discussed both positive and negative goal interactions. Negative interactions are basically conflicts between goals. They have defined resource requirements of a goal by considering all of its plans. Authors have defined an algorithm for Goal Plan Tree construction, annotated with resource requirements of goal. At run time resource summary is updated in Goal Plan Tree. Focus of their work is on defining goal plan tree annotated with resources both at start and run time (Thangarajah & Padgham, 2011). Faults that may occur if a certain interaction or missing coverage; are not discussed. No coverage metrics has been defined neither design diagrams used for tree construction are specifically elaborated.

J. Thangarajah, G. Jayatilleke, and L. Padgham (2011)

Thangarajah et al. (2011) used scenarios of Prometheus methodology and add a structure in scenario which is then used at later stage of design for traceability, testing and analysis. Scenario is extended with three steps; first add sequence of percept and actions to be executed in scenario, second add test descriptor with the scenario and third modification is to add traceability link between different entities. Propagate the design after scenario modification to other design artifacts (Thangarajah, Jayatilleke & Padgham, 2011). Limitation is that only single scenario is tested in isolation, no system level traceability is performed. MAS' execution and faults are not considered in this approach.

P. Shaw and R. H. Bordini (2011)

Shaw and Bordini (2011) presented an alternative approach for goal plan tree analysis. For goal plan tree modeling 5 tuples prolog function node is used. This approach is then compared with another approach that used Petri-nets instead of goal plan tree for goals reasoning, negative and

positive interaction (Shaw & Bordini, 2011). Fault model has not been presented instead only Petri-net and goal plan tree models have been compared. No coverage metrics have been defined.

Z. Zhang, J. Thangarajah, and L. Padgham (2011)

Zhang et al. (2012) presented an approach for automated testing of multi agent system for units. The system under test is evaluated with respect to system design model. Authors have devised a framework for testing individual plan. Orders of events, plans have developed and test cases are executed with proper test data. Equivalence class partitioning is used for test data generation against variables e.g. simple, complex and belief variables (Zhang, Thangarajah & Padgham, 2011). Only unit testing is performed, no faults identification model and coverage metrics for faults with respect to goals and plans have been discussed.

J. Thangarajah, S. Sardina, L. Padgham (2012)

Thangarajah et al. (2012) presented a technique to measure plans coverage by using numeric measures and their overlap for agents. Agents have multiple plans and they are executed to achieve certain goal(s). Sub goals are actually intentions from a plan and agent has to choose which intention to execute. Coverage is measured by number of models or area a plan is applicable and exclusive coverage equation is used for coverage measure. Overlap means that two or more plans are applicable for a certain situation and exclusive overlap equation is used to distribute overlap (Thangarajah, Sardina & Padgham, 2012). No implementation and validation have been done neither any fault model along with coverage criteria is specifically defined for goals and plans.

L. Padgham, Z. Zhang, J. Thangarajah and T. Miller (2013)

Padgham et al. (2013) presented model based test oracle creation for unit testing of agents. Fault model has been created to cover individual units. Prometheus methodology detailed design artifacts are used. For each agent and unit within an agent, code augmentation is performed for test harness and test cases for the unit are executed and finally the test report for each unit is generated. Equivalence class partitioning and boundary value analysis testing techniques have been applied to test variables for test case generation (Padgham et al., 2013). Event, Plans and beliefs related faults are considered and results of test cases are evaluated with respect to fault types and number of occurrences (Padgham et al., 2013). Event plan tree has been developed but

goals and their link to plans and sub-goals seems 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 are also missing in the presented fault model.

J. Harland, D. N. Morley, J. Thangarajah, N. Yorke-Smith (2014)

Harland et al. (2014) discussed operational semantics of goal life cycle in BDI agent. Authors have included both achievement and maintenance goals along with detailed description of goal state; whether suspended or not. Difference goals operations i.e. drop, abort, suspend and resume; are practiced. Authors present these semantics in agent language CAN (Harland et al., 2014). Major contributions are focusing on rich and accomplishment goals, specification for abort and suspended in all goal types, and considering plans execution in case sub goals are triggered dynamically (Harland et al., 2014). Authors in this paper, worked in line with proactive behavior in goals presented by (Duff, Thangarajah & Harland, 2014). Goals semantics are presented using CAN agent based language, but fault model has not been addressed in case any mandatory or optional goal is not executed. Furthermore, only single agent's goals are considered by (Harland et al., 2014), integration and system level aspects of MAS are not considered.

S. Duff, J. Thangarajah, and J. Harland (2014)

Duff et al. (2014) presented a technique in which maintenance goals are experimented to work in proactive mode. Maintenance goals are type of goals in which a condition has to be satisfied. Proactive mode is to act before a condition is violated e.g. fuel level in mars rover agent (Duff, Thangarajah & Harland, 2014). Only the goal condition is discussed in their technique. Other factor like relevant plan/sub goals achievement and faults that could occur if certain coverage is missed is not discussed in by Duff et al.

J. Thangarajah, J. Harland, D. N. Morley, and N. Yorke-Smith (2014)

Thangarajah et al. (2014) devised a mechanism for level of completeness of goal in BDI style agents. They used resource consumption and effects after completing a goal to measure goals completeness (Thangarajah et al. 2014). This approach has the overhead that an agent needs to track number of resources consumed so far for goals. Effects are used to quantify completeness

while achieving the goals. Possible faults that may occur while achieving desired goals can cause MAS to produce undesirable outcomes. Faults identification with respect to coverage measures are not addressed by this approach.

J. Thangarajah, J. Harland, D. N. Morley and N. Yorke-Smith (2014)

Thangarajah et al. (2014) presented an approach to quantify the goals completeness in BDI 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. Efforts, accomplishments, no of actions performed by agent and time taken for the action; are factors that have been considered to quantify goals completeness (Thangarajah et al. 2014). Authors have taken the idea of a goal-plan tree, in which goals with relevant plans have been annotated to form a tree. Authors have implemented the approach using prolog based agent language using Mars rover agent case study (Thangarajah et al. 2014). Goals and plans coverage criteria were not defined neither relationship of goal-plan tree with respect to scenarios and protocol diagram was done. Faults identification was not covered.

Y. Abushark, J. Thangarajah, T. Miller, J. Harland (2014)

Abushark et al. (2014) has presented an approach for early phase detection of design faults by comparing traces from detailed design. Authors have transformed agent design like interaction diagram into Petri-Nets. A plan graph for all plans along with events related to the plans has been constructed and possible traces from plan graph have been extracted. Finally they compare the traces with Petri-Nets representation of protocol and violation of message interactions were recorded and presented in the form of report. They verified that messages occurred in detail design in same order as described in protocol. Authors have built a tool for automating the said process (Abushark et al., 2014). Multi-agents systems interactions have not been covered neither percepts and actions are recorded.

Y. Abushark, J. Thangarajah, T. Miller, J. Harland, M. Winikoff (2015)

Abushark et al. (2015) devised an approach to detect design faults in agent designs. Plan structure was checked against requirements specifications. Abushark et al., used Prometheus design files e.g. goal overview and scenario diagram and converted these design models into

Petri-net. Authors constructed a plan graph using agent role grouping and agent design. Plans traces were extracted from plan graph and checked traces with Petri-net. A comparison report was also generated. They generated 21 plan graphs for validation. The approach however, is applicable only at design time and identifies only design faults. Implementations or working of MAS has not been tested with respect to MAS design. Defects that were injected at time of implementation were completely missed in this approach.

3.2 Analysis

In this section we present an analysis in a table, our analysis is done based on some parameters. Following are parameters on which above described techniques are analyzed. These parameters are selected based on the MAS testing requirements and literature, i.e., (Rehman and Nadeem, 2013).

3.2.1. Evaluation Criteria

This section presents the evaluation criteria for the analysis and evaluation of the covered techniques. The analysis is based on parameters whether a given technique supports the selected parameter or not. Following are the some selected parameters along with possible values:

a) Modeling Methodology

There are different agents modeling methodologies used for designing a multi-agent system. This parameter is used to identify which modeling methodology is used in a testing approach. Approaches using Prometheus methodology design are of more interest here in this research as Prometheus methodology is very rich to capture all design phases information and has its design tool as well.

b) Testing Basis (Code based/Model Based)

This parameter is used to identify a testing technique on the basis of specification, model or code as its input artifact for performing the testing. We are looking for model based testing approaches; while some code based technique targeting plans in the implementation are also important for our research in this thesis.

c) Level of Testing

This parameter is used to check whether a technique is testing the individual agent behavior or it is testing the MAS by combination of multiple agents working in an environment. Whether test cases are generated according to the testing requirements? Integration or system level testing techniques are of consideration in our research. Based on the level of testing, further details like agents interaction and flow of information between different design artifacts is considered.

d) Input Artifact

This parameter is used to present input artifacts used in approach. Input artifact coverage shows the behavior covered in testing approach. Different modeling methodologies have different design artifacts. Prometheus methodology has different design artifacts that contain different type of MAS information. The richer artifacts a technique uses, the more effective test of MAS is performed.

e) Test Data Generation

Test data is used to run test cases. Test cases derived from test model have to execute on MAS implementation. A testing approach can use generate and use test data. This parameter is used to identify whether testing technique is generating automated test data or not.

f) Artifact Coverage

This parameter is used to identify level of coverage achieved by the testing approach with respect to a certain input artifact or design diagram. Artifact coverage can be numbers showing the scale of coverage achieved for different type of interactions or activities involved in a certain artifact. For example 3 for high, 2 for medium and 1 for low.

g) Tool Support

This parameter is used to check whether the testing approach is supported with some tool or not. Tools are a measure to validate the testing approach results that depict whether the testing processes working or not.

Table1: Comparison of Technique Based on Parameters

Parameters	Modeling Methodology	Testing Basis (Code based/Model Based)	Level of testing	Input Artifact	Test Data Generation	Artifact Coverage	Tool support
Low et al., 1999	No specific methodology (all following BDI architecture)	Code Based	Unit Testing	Plans and Nodes as statements	Yes	2	Yes
Thangarajah et al. 2003	No specific methodology (BDI architecture)	Model based	Unit Testing	Goals	No	2	No
Nunez et al., 2005	Not specified	Model based	Unit Testing	Utility State Machine	No	2	No
Zhang et al., 2007	Prometheus	Model based	Unit Testing	Plans	Yes Manually	2, as only plans are covered.	No
Shaw et al. 2008	No specific methodology (BDI architecture)	Model based	Integration Testing	Goals and Plans	No	2	No
Zhang et al., 2008	Prometheus	Model based	Unit Testing	Plans	Yes Manually	3	No
Miller et al., 2010	Prometheus	Model based	Integration Testing	Protocol Diagram	No	1	No
Shaw & Bordini, 2011	No specific methodology (BDI architecture)	Model based	Integration Testing	Goals and Plans	No	2	No
Thangarajah et al. 2011	Prometheus Methodology	Model based	Unit Testing	Scenario Diagram	Yes	3	No
Zhang et al., 2011	Prometheus Methodology	Model based	Unit Testing	Goals and Plans	Yes	2	Yes
Thangarajah et al. 2012	No specific methodology (BDI architecture)	Code based	Integration Testing	Goals and Plans	No	2	No
Padgham et al., 2013	Prometheus Methodology	Model based	Unit Testing	Agent and Capability Diagrams	Yes	2	No, Automated Framework Presented
Harland et al., 2014	No specific methodology (BDI architecture)	Model & Code based	Unit Testing	Goals	No	2	No

Duff et al, 2014	No specific methodology (BDI architecture)	Code based	Unit Testing	--	No	1	No
Thangarajah et al. 2014 <i>(level of completeness of goal)</i>	No specific methodology (BDI architecture)	Code based	Unit Testing	Goals	No	2	No
Thangarajah, et al., 2014 <i>(Quantifying completeness of goals)</i>	No specific methodology (BDI architecture)	Model & Code based	Unit Testing	Goals	No	2	No
Abushark et al., 2014	Prometheus Methodology	Model based	Integration Testing	Protocol Diagram	No	1	Yes
Abushark et al. 2015	Prometheus Methodology	Model based	Integration testing	Scenario, Role, Agent and goal diagrams	No	2	Yes

Based on the comparison performed in Table 1, existing model based testing techniques for multi-agent systems do not cover every aspect of multi-agent systems, i.e., goal-plan coverage, dependencies and interactions. Goals coverage with respect to plan using design artifacts has not been done. We propose and implement automated testing approach for multi-agent systems using design artifacts like goal overview diagram, scenario overview, protocol diagrams and process diagrams, i.e., agent and capability overview diagrams. We will include goals coverage derived from goal overview diagram and Scenario diagram to capability overview diagrams by using Goal-Plan Graph.

3.3 Summary

Dynamic view of Prometheus methodology contains scenarios along with a goal hierarchy. Scenarios are converted to relevant protocols which are further elaborated in process overview. A process can be elaborated with the help of agent and capability overview diagrams in detailed design. Goals are defined at specification level in Prometheus methodology while a scenario captures the sequence of action, percepts and goals in scenario overview diagram. Scenarios are more specifically refined and elaborated in interaction diagram and protocols; each scenario has its own interaction diagram which follows the same sequence as described in scenario. Each interaction protocol can have multiple agents communicating with each other. Communication

between agents and environment in a multi-agent system is designed using Prometheus methodology through protocol diagram. Protocol diagram is used as the design model to test multi-agent system. Miller, Padgham & Thangarajah (2010) state that the interaction between the agents possesses complex behavior and therefore testing of interactions is important. They proposed testing strategy which uses protocol diagram and considered only message interaction between agents involved in a communication protocol. In Prometheus methodology, actions and percepts are also part of protocol diagram with messages, which are not covered in (Miller, Padgham & Thangarajah, 2010). However only message interactions are covered in existing techniques which are not enough to guarantee fault free MAS. Dependency fault occurs in case of missing percept as percepts are required for events. Actions are used to update output of agent to environment. Messages usually dependent on the correct sequential execution of related actions and percepts involved in an interaction. In a protocol, percepts and actions interactions have their importance and therefore their coverage is necessary for effective testing.

Goals and plans are key elements of any multi-agent system. Proper and ordered execution of goals and plans is necessary for fault free MAS. Process diagrams include agent and capability overview diagrams that contain the plan(s) for goals defined in system analysis phase of Prometheus methodology. In the literature no fault model is presented if certain goals are not achieved or plans for the goal are not triggered. The idea of goal-plan diagram has been previously presented in (Rehman & Nadeem 2011; Thangarajah & Padgham 2011; Shaw, Farwer & Bordini 2008) for BDI agents; but for Prometheus this work has not been done so far. We are not considering unit testing, but only focus on integration and system testing of MAS. Current work in the literature does not target faults identification and complete coverage of goals and plans.

CHAPTER 4: A FAULT MODEL FOR MULTI-AGENT SYSTEMS

Fault model for MAS is presented in this chapter. Based on the literature and our research questions, our research concentrates on two aspects of multi-agent system testing. One aspect is interaction testing using protocol diagrams which targets interaction faults and another aspect is goal and plans faults identification & coverage which are related to goal and plans within a MAS. Our aim is to test the interactions of agents using their model specified in terms of interaction protocol and goal, sub goals and plans execution from system specification to detailed design. Protocol diagram is used as a mean of interaction in architecture design of Prometheus Methodology. Goals Specified at system specification level are achieved by one or more plans in detailed design phase. An agent executes plans for the goals that may trigger sub-goals or plans to execute. A main goal may have some sub-goals contributing their part in achieving the objective. A relationship of goals and plans along with all of relevant details like agent, scenario and capability will be used to test goals and sub goals achievement along with proper plans execution.

Based on the details of MAS and Prometheus methodology design artifacts, we use the key design artifacts that play their role in model based interaction and goal/plan testing of MAS. Such artifact includes protocol diagram, goal overview diagram, scenario diagram and process diagrams.

4.1 Fault Model

In this section we will elaborate our identified fault model for MAS testing. To identify faults in MAS there is a need to define a fault model that defines possible faults that could occur in MAS. In order to successfully find faults, understanding of fault is helpful for its identification and understanding. Fault model elaborates details of possible faults in MAS and to describe faults which remain if some coverage has not been achieved. Fault directed approach to test MAS will effectively contribute towards fault free MAS. Actual output of the implementation is compared with expected output to check seeded faults identification. We are assuming that design of MAS is fault free because we utilize design artifacts for MAS testing. Faults that could occur in SUT's implementations will be successfully revealed and reported. Based on the literature and gap in

existing work we focus two branches of MAS testing in this research, i.e., interaction testing and goal plan coverage based testing, i.e., system testing of MAS. For each branch we have defined separate fault model. For interaction testing no fault model is available in literature. For system testing, i.e., goal, sub-goal and plan related faults some faults like incorrect belief, incorrect context etc are presented by (Padgham et al, 2013), but there are certain aspects of MAS that are still missing and can cause MAS to behave unexpectedly. Figure 4.1 shows MAS working in general along with percept, messages and actions interactions. Figure also shows how plans and goals have a relationship in MAS execution. Percepts are the information received from environment that creates an event. Event can trigger plan of an agent. Plans are executed to achieve a certain goal. Messages are the flow of information between agents. Actions are the output of the agent working in MAS which could change state of the environment by updating its beliefs.

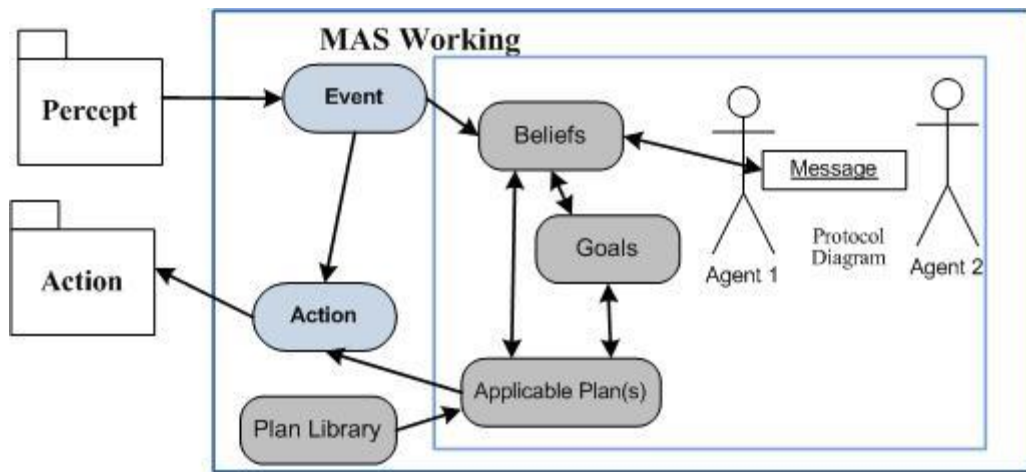


Figure-4.1: MAS Working and Flow of Information

By looking at MAS working and literature analysis we propose two types of faults for our testing approach in this research thesis. Following is the details of two types of our fault model.

4.1.1 Fault Types: Interaction Faults

We intend to identify possible faults that can occur if coverage of any interaction has not been done. In multi-agent system different faults can occur during interaction between agents and environment in a protocol diagram.

In analysis overview diagram, necessary percepts and actions are presented that are propagated further its sub-sequential levels of design like architectural and detailed design phase. Figure 4.2 shows propagation of percepts and actions along with messages from analysis overview to system overview. Details of protocol diagram is used to model the graphical representation of protocol, in which sequence of actions, percepts and messages along with required AUMN notation e.g. loop, option etc are listed.

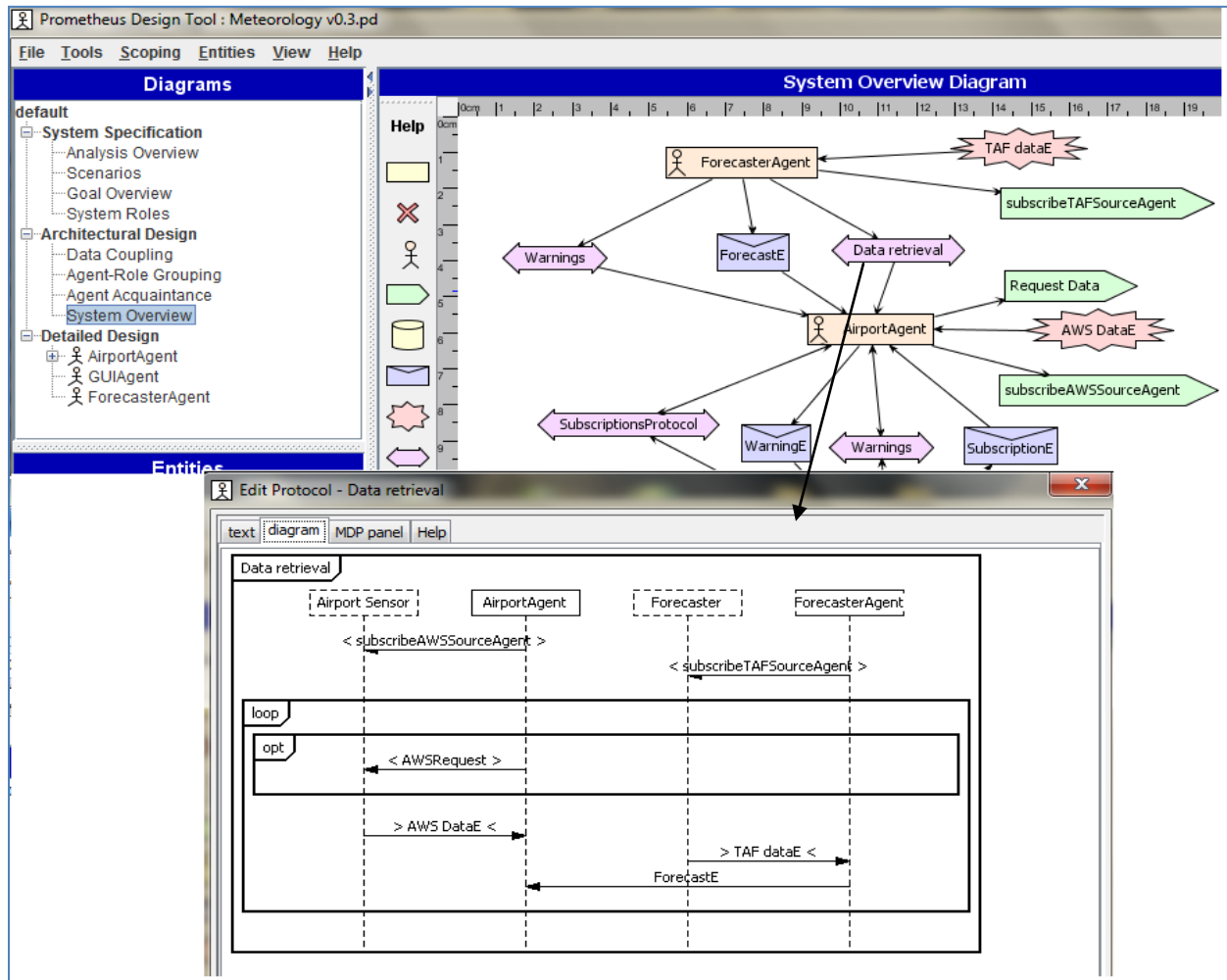


Figure-4.2: Protocol Diagram in PDT with Interactions

To detect faults, coverage of possible interactions should be achieved. Figure 4.3 describes Warnings Protocol of Meteorological Alerting System (Mathieson et al. 2004). Warnings protocol contains four actions namely *subscribeAWSSourceAgent*, *subscribeTAFSourceAgent*, *Request Data* & *Show Warning*; three percepts *Subscription from User*, *AWS DataE* & *TAF dataE* and three messages *SubscriptionE*, *ForecastE* & *WarningE*. Protocol involves three agents

GUIAgent, *AirportAgent*, & *ForecasterAgent* and three actors *User*, *Airport sensor* and *Forecaster*. Actors can be environment including people, external system etc which interact with multi-agent system and send percepts to system. System output will be in the form of actions.

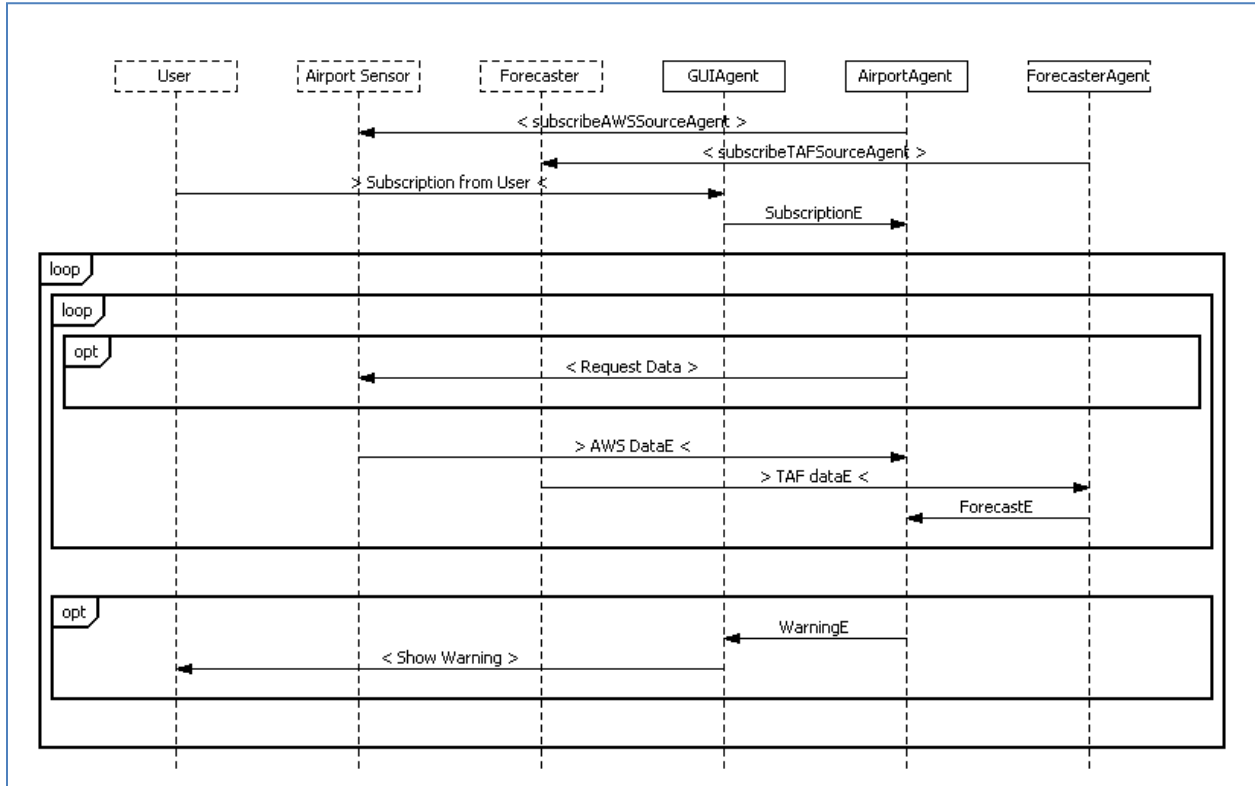


Figure-4.3: Warnings Protocol of Meteorological Alerting System

Currently only message interaction has been done while testing multi-agent system by (Miller, Padgham & Thangarajah, 2010). Percepts and actions play a vital role in any interaction if they are involved, so missing their coverage can contain following faults in MAS. A plan is triggered by an event which is invoked by percept. Event can be message or percept or internal event. Internal events are subtask within an agent in Prometheus methodology. Following are the possible faults that could occur in MAS if complete coverage of all possible interactions has not been achieved.

Problem 1:

Dependency fault: Missing environment information or percept; MAS cannot get updated information about environment or expected inputs. This may cause wrong plan to trigger or even no plan triggered if no event has been generated.

Problem 2:

Operational Fault: Unable to update actors/environment in case of action not executed; MAS may not update/inform user or external environment if action event has not been tested. Specific outgoing event may not ever be posted.

Problem 3:

Synchronization fault: Wrong message content can be conveyed to other agent if all necessary information is not available. This could only be possible if expected percepts or action do not take place on which message content is dependent. Improper sequence of execution is also problematic. Both action and percept coverage with defined sequence is required.

For example, if *Subscribe AWS source* action is not covered then proceeding *AWSDataE* percept may not occur and consequently wrong *WarningE* message will be sent.

We will use multi-agent system design to test it against its implementation focusing on identified problems that lies in existing work. Our approach will uncover the interaction faults described above that would lie between agents and actors. Our testing approach for multi-agent system will test interaction faults using design artifacts like protocol diagram. We convert protocol diagram into Protocol graph in our testing framework and perform all possible interactions testing.

4.1.2 Faults Types: Goals, Sub-goals and Plans Faults

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. Goals and plans in MAS are used to achieve desired functionality so their correct and ordered execution is necessary for MAS reliability. Different types of faults can occur in MAS some of them are discussed by (Padgham et al, 2013) like incorrect belief, incorrect context etc, but there are certain aspects of MAS that are still missing and can cause MAS to behave unexpectedly. As depicted in Figure 4.1, goal and plans do have a relationship between them in order to achieve MAS functionality. Goals and plans have a relationship as plans need to execute for a specific goal to achieve, plans can have sub-goals as well. Some goals have more than one applicable plans, all of which must be

executed in order to consider the goal as achieved while some goals are achieved if any of applicable plans is executed. Same is the case with plans and its sub-goals. Such type of relationships are represented by ‘AND’ and ‘OR’ between goals in goal overview and Goal-Plan Diagram.

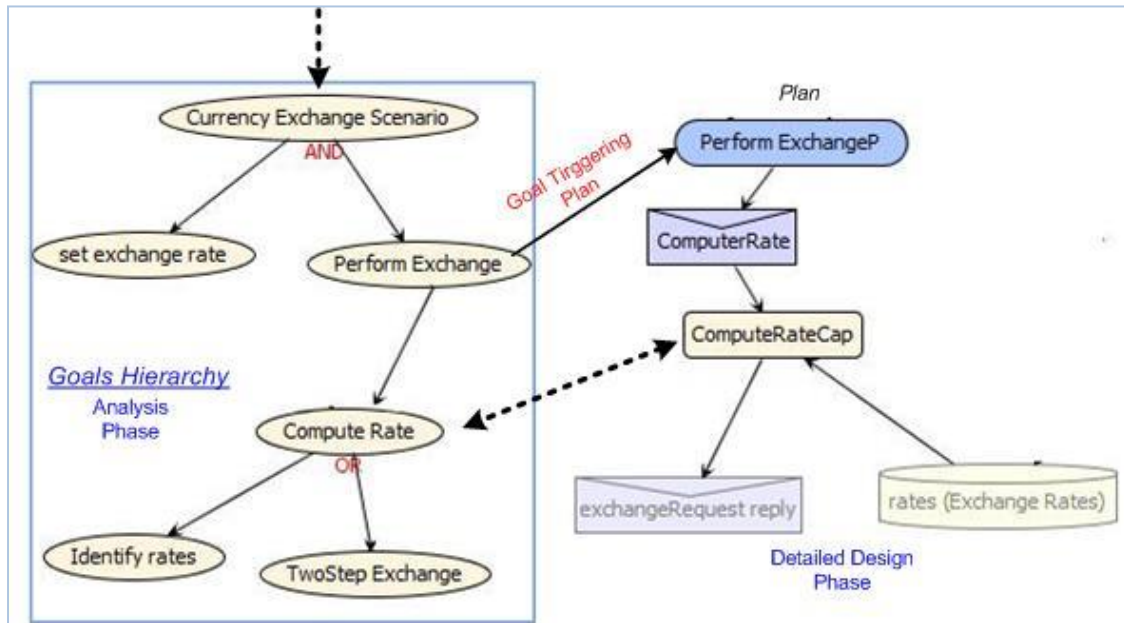


Figure 4.4: Goal Plan Relationship in PDT

In Prometheus Methodology, goals have been defined in goal overview diagram at system analysis phase. Each goal is part of system analysis and Scenario description of the MAS. Detailed design of Prometheus using PDT, plans do certainly have sub-goals. Goals defined in system analysis phase have assigned relevant plans in detailed designed phase. Detailed design contains applicable agents, applicable capability and plan(s) for specific goal and sub-goals of specific plans. Every plan has exactly one triggering goal and multiple sub-goals (steps) in the plan. Satisfaction of all sub-goals in a plan means the plan is satisfied, and therefore its triggering (sub) goal is achieved. Figure 4.4 captures such a relationship between goals and plan from system analysis phase to detailed design phase, where part of Multi-currency banking MAS design is shown.

Goal triggers a plan which generates some events or triggers other sub-goals. Each of which is executed to satisfy top level goal and plan. In Prometheus, plan descriptor shows goal(s) for which plan(s) will be triggered. Figure 4.5 shows plan descriptor providing details relevant to a

plan i.e. goal to satisfy, sub-goals, incoming and outgoing messages and triggers. In Figure 4.5 plan descriptor of *Perform ExchangeP* Plan is presented.

Perform ExchangeP - Descriptor

Name: Perform ExchangeP

Description: This Plan is used for currency exchange functionality

Triggers: exchangeRequest: Communicator agent --> CurrencyExchange agent (Message) [Edit]

Context: If account currency is different from operational currency

Incoming messages:

Outgoing messages: ComputerRate: Perform ExchangeP --> ComputeRateCap

Percepts:

Actions: Request Error

Used data:

Produced data:

Goal: Perform Exchange [Edit]

Figure 4.5: A Plan Descriptor in PDT

From Prometheus design artifacts we have following details for goal, sub-goal and plan relationships. For instance we consider account opening goal for multi-currency banking system.

Goal	Applicable Plan(s)	Applicable agent, Scenario
Account Open	Create AccountP, Account InfoP	Bank Account Agent, Obtain Information Scenario

These applicable plans can have ‘AND’ relationship with goal, both plans must be executed in order to declare account open goal as achieved.

Plan	Sub-Goals	Applicable Agent, Scenario
DebitAccountP	Debit Account, Debit Account Error, Debit Account Exchange	Bank Account Agent, Debit Account Scenario

Above plan have three sub-goal having OR relationship with them. Satisfaction or execution of any goal can satisfy the plan but some functionality may be left unchecked.

Goal plan relationship should also capture the agent which is executing a certain plan. The goal which is satisfied may also be part of certain scenario, plan may be included in any capability included under an agent. Testing artifacts for goals, sub-goals and plans related faults include goal overview, scenario overview, protocol diagram, agent and capability diagrams for Prometheus. Considering these testing artifacts and relationship between goals and plan we have defined goal, sub-goal and plan faults which cover possible faults in such interactions and coverage.

In our fault model we have also captured such relationship discussed above along with other possible faults that may occur when a MAS is running. Following are our defined fault types and their description covering possible faults occurrence in MAS:

- a) **Inaccurate goal achievement:** If more than one plan 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.
- b) **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 a plan not to produce desired output.
- c) **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.
- d) **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.

- e) **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.
- f) **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 is required.

We have defined a fault model with two types of faults for MAS interaction testing and goal, sub-goals and plans based testing. We have elaborated scope and details of design artifacts involved in our MAS testing research. In the subsequent chapter testing framework for testing of MAS is presented in detail.

CHAPTER 5: MULTI-AGENT SYSTEM TESTING FRAMEWORK

This chapter presents the proposed testing framework for testing of MAS. We provide testing framework for both interaction and system testing aspects of MAS. Testing artifacts are the Prometheus design artifacts of MAS that are used in testing framework. Fault model have also been described in previous chapter 4 depicting possible faults that could occur in multi-agent system. The Extent of level of MAS testing is then discussed according to the considered testing level; our proposed testing frameworks are presented in the following sections.

Our focus in this research thesis is on Integration and system level testing. In Integration testing, Prometheus design artifacts, i.e., protocol diagram is used for interaction testing, while analysis phase and detailed design phase diagrams are used for system level testing to test goal, sub-goal and plan coverage. In our testing framework, test model have been generated from design artifacts and test paths are generated automatically on our test models. Test cases are generated and executed to find possible faults in MAS.

5.1 Multi-agent System Testing Framework

In this section, we present our testing framework for MAS testing. Our focus is on integration and system level testing of MAS. Figure 5.1 shows testing framework for MAS in which Prometheus design artifacts are used as input to the testing framework. Test model is generated by using system design model and test path are then generated from test model for integration and system level testing. Integration testing is done by testing different MAS interactions. Prometheus methodology design diagrams like goal overview, scenario overview, protocol diagram, agent overview and capability overview diagram contains working knowledge of MAS. All these diagrams are required for testing MAS due to their information containment. For interaction testing the protocol diagram of Prometheus methodology is used and further test model is created from it. System testing requires protocol, goal, scenario and process diagrams for goal, sub-goals and plans coverage.

We generate test paths from test model. Integration and system test paths are used to generate test cases. For each testing level, test cases are generated and executed on actual implementation of MAS. Test result evaluation shows whether a test case is a fail or a pass after comparing with expected output. We will elaborate integration testing framework for interaction testing in section 5.2 and system level testing for goal, plan testing in section 5.3.

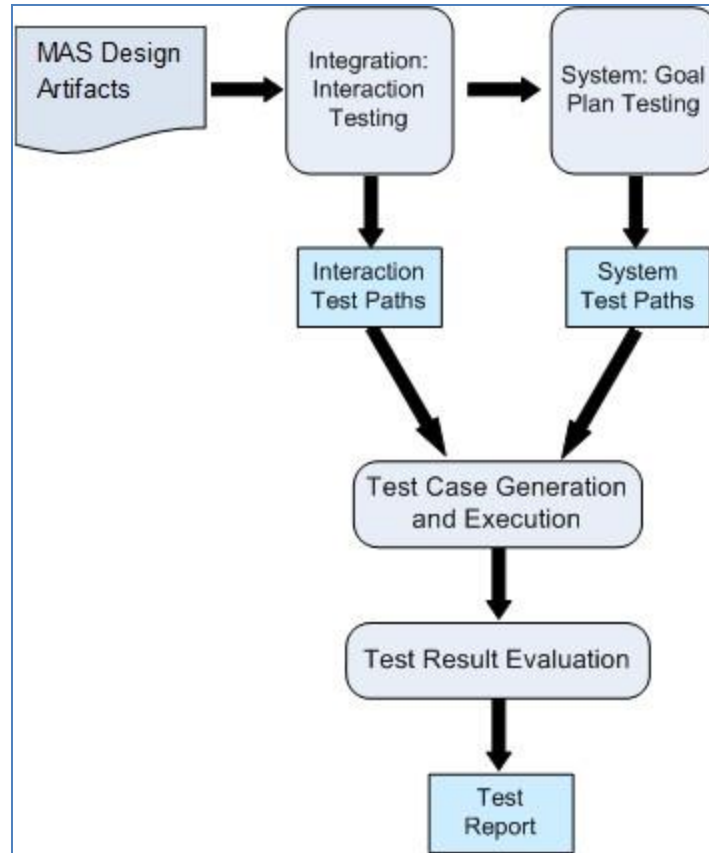


Figure 5.1: MAS Testing Framework

5.2 MAS Integration Testing: Interaction Testing

In this section, we discuss our testing framework for interaction testing of multi-agent systems using the Prometheus design artifact. Protocol diagram contains all possible interactions between agents working in MAS. Our testing framework for interaction testing is shown in figure 5.2. We take design diagram information as input and verify that system conforms to the design or not. Interaction protocols presented in protocol diagram will be used to build a test model which covers messages, actions and percepts in order to achieve certain goals. Coverage criteria have been defined on protocol graph, covering every possible interaction between agents. A tool has been developed which uses identified coverage criteria, keeping in mind the messages and percepts, interaction protocol and generates the test paths accordingly.

Figure 5.2 describes the testing framework for interaction testing approach. Testing framework technique has four main processes namely protocol graph (test model) generator and test path generator, test case generation and test result evaluation. Protocol graph generator uses

Prometheus interaction protocol presented in protocol diagram as input and produces protocol graph, i.e., test model for interaction testing as the output.

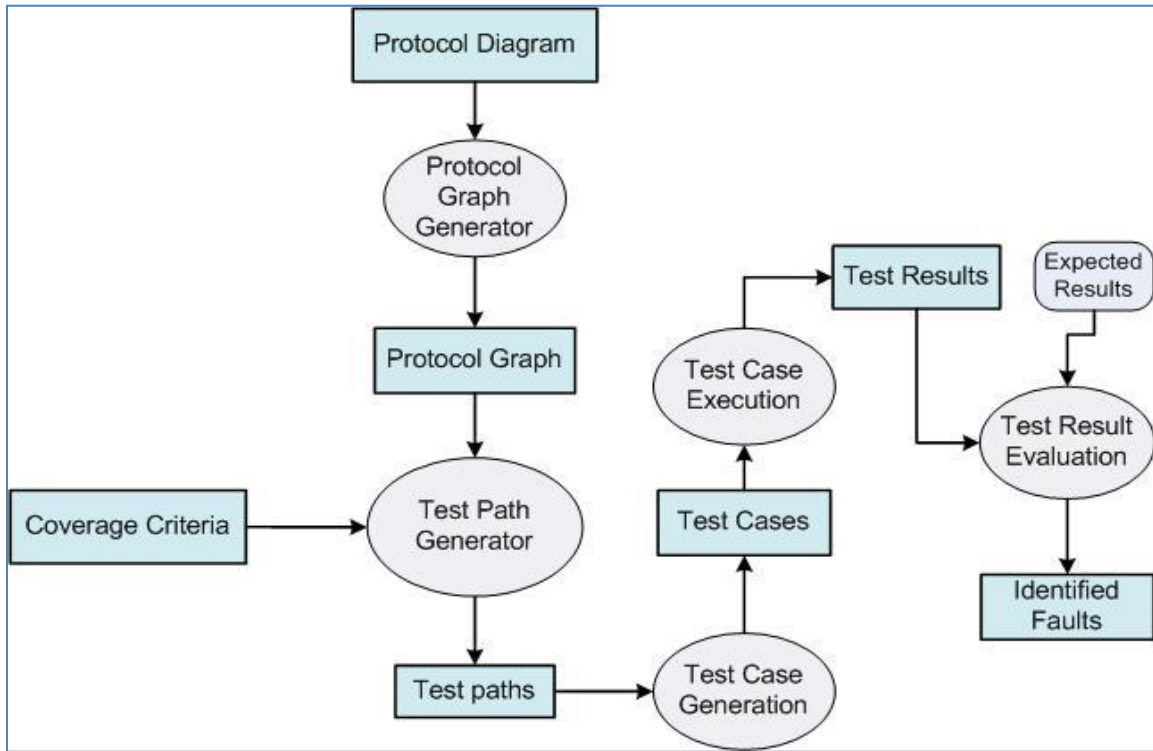


Figure 5.2: Testing Framework for Interaction Testing

Test model i.e. protocol graph will be used to generate test paths according to defined coverage criteria. Different coverage criteria are defined focusing on percepts and actions along with messages and are used as input to test path generator. Coverage criteria have been defined to cover all possible interactions occurring in a protocol graph. Test path generator uses our test model, i.e., protocol graph and applies different defined coverage criteria to generate test paths. These test paths cater for interactions like messages, actions and percepts in order to achieve certain goal. The test case generation process is used to generate test cases manually. We execute test cases to identify faults which reside in MAS. We have presented fault model for interaction testing in section 4.1.1. Test result evaluation compares expected results with observed results and identifies any faults presented in MAS. For verification we will seed faults in MAS. In subsections we will discuss details and working of all defined processes of our testing framework for interaction testing of MAS.

5.2.1 PROTOCOL GRAPH GENERATION

In interaction testing framework, interaction protocol or protocol diagram is used as the design artifact which is transformed into a protocol graph. Protocol diagram contains messages, actions and percepts interactions between agents and actors. Messages are passed only between the agents while actions and percepts interaction are performed between agents and actors. We convert the protocol diagram into protocol graph. Protocol graph has been introduced by (Miller, Padgham & Thangarajah, 2010). They defined two sets of test coverage criteria for multi-agent interaction testing. The first uses only the protocol specification, while the second considers also the plans that generate and receive the messages in the protocol. Miller, Padgham & Thangarajah do not cover the actions and percepts during the interaction (2010). We extend protocol graph because actions and percepts are also major interactions that occur in MAS. Many faults may remain in MAS if any of its interactions are left uncovered.

Figure 5.3 shows a Data Retrieval protocol diagram of Meteorological Alerting System. Prometheus interaction protocol presented in protocol diagram of a Meteorological Alerting System (Mathieson et al. 2004) is used as an example. Meteorological Alerting System is used to monitor real time terminals for forecasting in Australia. Data Retrieval protocol contains three actions namely *Subscribe AWS Source*, *Subscribe TAF source* & *Request Data*; two percepts *AWS Readings*, *TAF Data* and one message *ForecastE*. Protocol involves two agents *Airport Agent*, *Forecast Agent* and two actors *Airport sensor* and *Forecaster*. Actors can be environment including people and external systems which interact with multi-agent system and send percepts to system. System output is in the form of actions. A protocol diagram has been converted into protocol graph by considering AUML interactions occurring in it. Generated protocol graph is our test model for interaction testing of MAS. Our test model i.e. protocol graph contains complete representation of all messages, percepts and actions performed between the agents and actors in a specific protocol diagram.

AUML Protocol diagram can be represented in description as shown below.

start Data retrieval

actor A Airport Sensor

agent B AirportAgent

actor C Forecaster

agent D ForecasterAgent


```

action B A subscribeAWSSourceAgent
action D C subscribeTAFSourceAgent
box loop
box opt
action B A AWSRequest
end opt
percept A B AWS DataE
percept C D TAF dataE
message D B ForecastE
end loop
finish

```

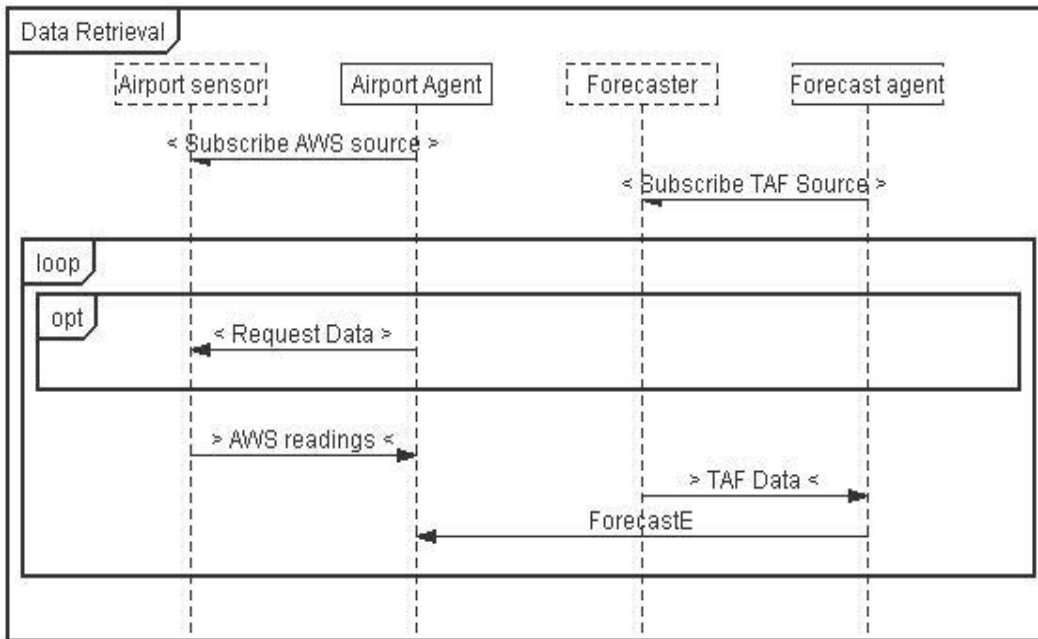


Figure-5.3: Data Retrieval Protocol Diagram (Mathieson et al. 2004)

Dam presented meta-model of protocol diagram which has two or more agents is shown in figure 5.4. (2008). Protocol can contain *Pelements* which can be a Message, a Go-to, a Label, a Box, a Region or even a Sub-Protocol. A *Pelements* is contained in a Protocol and a Region. A Box can be divided into a number of Regions, containing *Pelements*. There are different categories of Box such as Alternative, Option, Parallel, Loop, which are specified in the type attribute. Each region can contain *Pelements* and has a guard condition on that region being selected. Labels and Goto's represent incoming and outgoing continuations respectively. A Sub-Protocol represents a reference to another protocol (Dam, 2008).

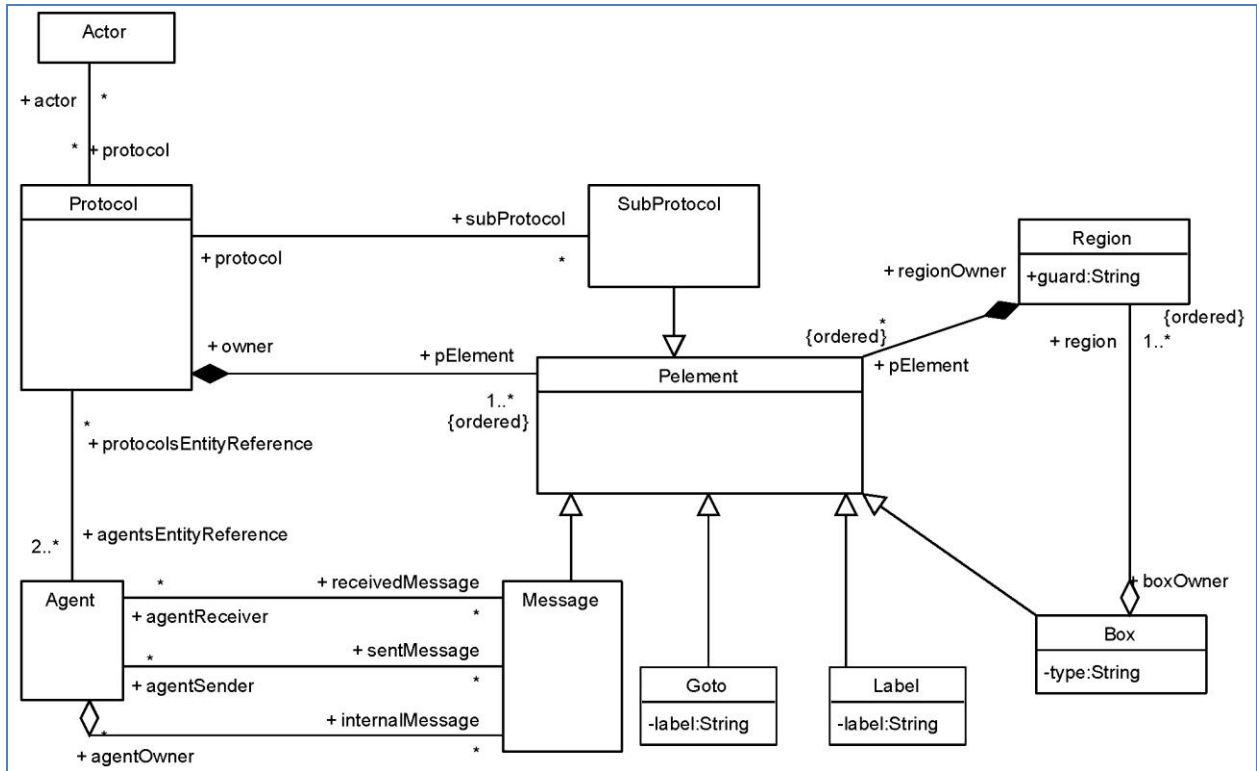


Figure-5.4: Meta-model of Protocol Diagram (Dam, 2008)

We have converted a protocol diagram meta-model into a protocol graph meta-model including messages, actions and percepts interactions as they are very important part of interaction protocol. There are following transformation rules used for protocol diagram meta-model into a protocol graph meta-model transformation.

Rule 1: Create a AUML Protocol instance from Meta-model of Protocol diagram with name as Protocol Graph

Rule II:

Create Graph Components class and Node class

Create an association of type composition between Component class and Protocol Graph

Create an association of type generalization between Node to Components

For each Pelement in Protocol Diagram

If Pelement is message

Create new Message bound element

Else If Pelement is GoTo

Create new Action bound element

Else Create new Percept bound element

End If

End If

Create associations of type generalization from bound element to Node class

End ForEach.

Rule III:

Create Graph Edge class

Create association of between Edge class and Node class

Atlas Transformation Language (ATL) rules for Protocol diagram Meta-model to Protocol Graph meta-model has been presented in implementation chapter 6 Section 6.1.1.

Figure 5.5 is the resultant meta-model of applying transformation rules on a protocol diagram meta-model.

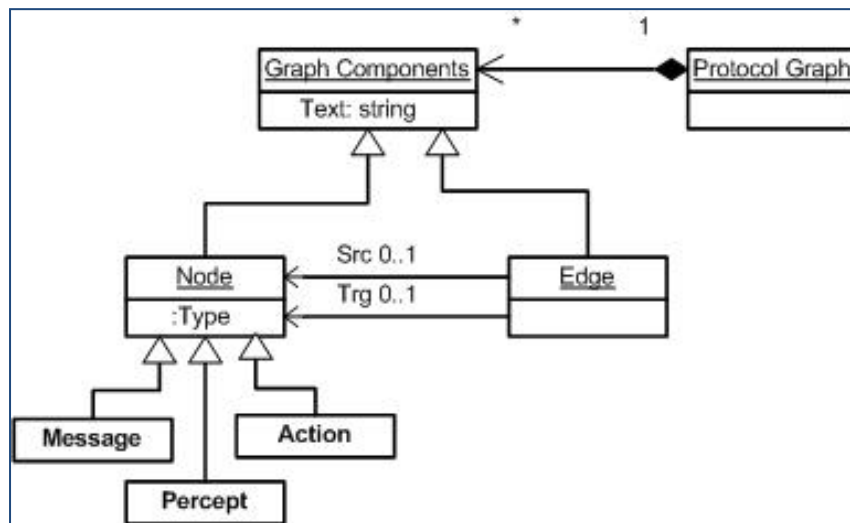


Figure-5.5: Meta-model of Protocol Graph

For each interaction, i.e., action, message and percept, relevant node is created in protocol graph. Protocol diagram meta-model, represents all AUML notations like loop, optional, parallel and alternative in BOX label. Transformation from meta-model to graph is automatic, details of implementations are discussed in chapter 6. Each node in the protocol graph has some data to receive, or send to other agents or environment. Figure 5.6 is the protocol graph of data retrieval protocol diagram. The protocol graph is our test model for interaction testing of MAS which is used to generate test paths that will lead to generation of test cases for MAS. Variables that cause this flow of information are extracted and their values are assigned as the test data. Protocol graph represents interaction protocol of MAS in terms of nodes and vertices form on which

different coverage criteria have been applied. Faults could occur in MAS if certain percept or action coverage or testing has not been done. For test paths generation different coverage criteria have been defined in the following section. We have test paths from model → add data with variables included in test path → test case is generated from test path.

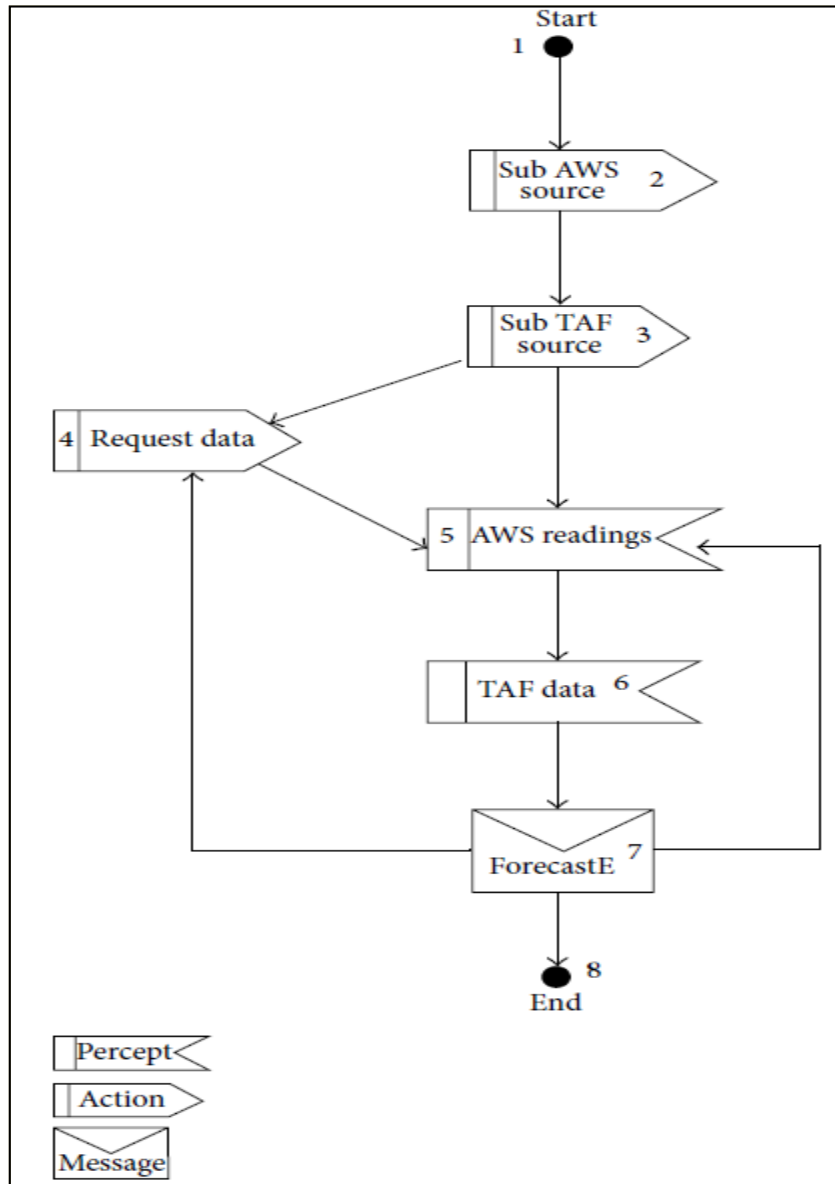


Figure-5.6: Protocol Graph for Data Retrieval Protocol Diagram

5.2.2 TEST COVERAGE CRITERIA

Miller, Padgham & Thangarajah have proposed some coverage criteria on protocol graph like message coverage and pair wise message coverage which are used in our proposed approach as well (2010). Additional coverage criteria for protocol graph including actions and percepts have

been defined in our testing technique. We have defined different coverage criteria that will cover all possible aspects of interactions between agents and actors in the form of message, action and percept. We have defined the following coverage criteria that will cover all possible aspects of interactions between agents and actors in the form of message, action, and percept.

Test Path: A test path is a complete path in a protocol graph G that starts at initial node i and end at final node f .

i. Message Coverage

A set of Test Paths (TP) is said to satisfy message coverage criterion for a protocol graph G if each message node m of graph G is included in at least one path $P \in TP$.

This coverage criterion ensures that every message node in the protocol has been traversed at least once. There exists a path from the start to traversing all messages in it.

ii. Action Coverage

A set of Test Paths (TP) is said to satisfy action coverage criterion for a protocol graph G if each action node 'a' of graph G is included in at least one path $P \in TP$.

In this coverage criterion every action node in the protocol graph must be covered by at least one test path for action coverage criterion.

iii. Percept Coverage

A set of Test Paths (TP) is said to satisfy percept coverage criterion for a protocol graph G if each percept node p of graph G is included in at least one path $P \in TP$.

In this coverage criterion every percept node in the protocol graph must be covered by at least one test path for percept coverage criterion.

iv. Message Action Coverage

A set of Test Paths (TP) is said to satisfy message-action coverage for protocol graph G if for each message $m \in M$ and each action $a \in A$; if edge (m, a) is in G , then (m, a) is a sub path of at least one path $P \in TP$.

Messages are passed between the agents and actions are passed between the agent and actor. Agent sends a message to an agent and agents send the action to actor, this sort of interaction must also be covered ensuring the message action coverage criterion.

v. Action Percept Coverage

A set of Test Paths (TP) is said to satisfy action-percept coverage for protocol graph G if for each action $a \in A$ and each percept $p \in P$; if edge (a, p) is in G, then (a, p) is a sub path of at least one path $P \in TP$.

Agents send an action to an actor in multi-agent system demanding some task to be completed, in return actor sends the percept containing the required information or data, and this sort of communication is covered in action percept coverage criterion.

vi. Percept Message Coverage

A set of Test Paths (TP) is said to satisfy percept-message coverage for protocol graph G if for each percept $p \in P$ and each message $m \in M$; if edge (p, m) is in G, then (p, m) is a sub path of at least one path $P \in TP$.

While receiving the percept from the actor, agents send a message to agent with necessary information, this sort of communication is covered in percept message coverage criterion.

vii. Pair wise Message Coverage

A set of Test Paths (TP) is said to satisfy pair wise-message coverage for protocol graph G if for each message $m \in M$ and each message n ; if edge (m, n) is in G, then (m, n) is a sub path of at least one path $P \in TP$.

In protocol graph, all cases in one message can be followed by another message are covered in pair wise message coverage. Addition of pair wise message coverage ensures arc coverage which is left uncovered in message coverage criterion.

viii. All Round Trip Paths

A set of Test Paths (TP) is said to satisfy all-round-trip-paths coverage criterion for a protocol graph G if it loop back on same state in graph G in at least one test path $P \in TP$.

Interaction protocol describes the protocol in AUML protocol diagram which contain loops as well depending upon the protocol requirements. All round trip paths coverage criterion in protocol diagram traverses all loops at least once and include those paths which loops back on same state in generated test paths. Figure 5.7 shows hierarchy of test coverage criteria used in interaction testing of multi-agent system.

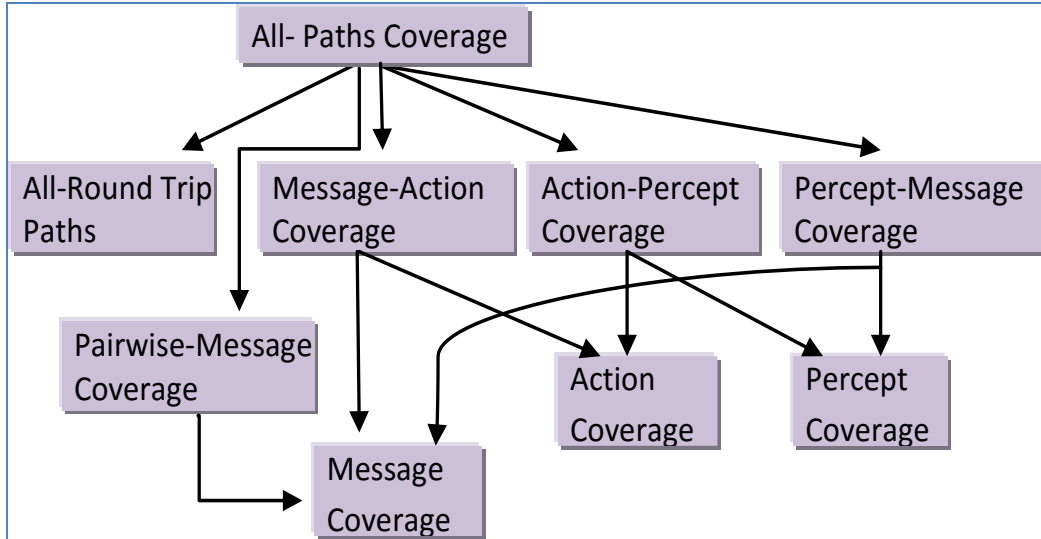


Figure-5.7: Test Coverage Criteria Hierarchy

5.2.3 TEST PATHS GENERATION

Once we have successfully converted protocol diagram into protocol graph, we need to generate test paths from protocol graph. In this subsection we describe second process of our testing approach named test path generation. Test path generation take protocol diagram and coverage criteria as input and generates test paths for a protocol. Algorithm-II presented in implementation chapter 6 is used to generate test paths from protocol graph. It takes Coverage Criteria (A set of defined coverage criteria) and protocol graph (Set of nodes and edges). Edge and node list is used to generate test paths according to defined coverage criteria. Details are explained in section 6.1.1. We have automated the test paths generation with the help of tool which is elaborated in 6.1.3. Table-2 shows generated paths by applying algorithm-II along with defined coverage criteria. Pair wise message criterion is not applicable in the example here; we will show this criterion path in our results and discussion chapter.

Table-2: Test Paths for Data Retrieval Protocol Graph

S. #	Coverage Criteria	Test Paths
1	Message Coverage	▪ 1→2→3→5→6→7(Message)→8
2	Action Coverage	▪ 1→2(Action)→3(Action)→4(Action)→5→6→7→8
3	Percept Coverage	▪ 1→2→3→5(Percept)→6(Percept)→7→8

4	Message Action Coverage	<ul style="list-style-type: none"> ▪ $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7(\text{Message}) \rightarrow 4(\text{Action}) \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$
5	Action Percept Coverage	<ul style="list-style-type: none"> ▪ $1 \rightarrow 2 \rightarrow 3(\text{Action}) \rightarrow 5(\text{Precept}) \rightarrow 6 \rightarrow 7 \rightarrow 8$ ▪ $1 \rightarrow 2 \rightarrow 3 \rightarrow 4(\text{Action}) \rightarrow 5(\text{Precept}) \rightarrow 6 \rightarrow 7 \rightarrow 8$
6	Percept Message Coverage	<ul style="list-style-type: none"> ▪ $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6(\text{Percept}) \rightarrow 7(\text{Message}) \rightarrow 8$
7	All round trip paths	<ul style="list-style-type: none"> ▪ $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$ ▪ $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$
9	All Paths coverage	<ul style="list-style-type: none"> ▪ Infinite # of Paths

5.2.4 TEST CASE GENERATION

Test case generation process of our testing framework for MAS interaction testing is manual. We use test paths generated from our test model, i.e., protocol graph. Test path contains nodes type, i.e., message, action and percept depending upon the applied coverage criterion. For each node we extract its triggering event and we provide data to trigger that event with the aim to traverse/test our target interaction. For example a test path against ‘action percept’ coverage criterion: $1 \rightarrow 2 \rightarrow 3(\text{Action}) \rightarrow 5(\text{Precept}) \rightarrow 6 \rightarrow 7 \rightarrow 8$ is used to test action interaction from an agent to environment and that should be proceeded by a percept from actor/environment to an agent. Our test case should target to generate test case that leads to the execution of MAS on action and then percept. But if there is some error in MAS implementation that could not traverse such an interaction in the desired order indicating a fault in MAS. We will generate test case for each coverage criteria so that all possible interaction have been tested.

Test path: $1 \rightarrow 2 \rightarrow 3(\text{Action}) \rightarrow 5(\text{Precept}) \rightarrow 6 \rightarrow 7 \rightarrow 8$

Action = Sub TAF Source

Percept = AWS Readings

Triggering event could be a calling function with some variable data values. Node Description Table (NDT) for each path contains all variable information or function required to traverse the path. Details of test case are presented with case study in results and discussion chapter 7.

5.2.5 TEST RESULT EVALUATION: INTEGRATION TESTING

Test Result evaluation process of MAS interaction testing consists of manual identification and calculation of expected results. For MAS to work correctly, all of its actions, percepts and messages should occur as defined in system design, i.e., protocol diagram. For each triggering event and function call, expected output is calculated manually and after executing test case observed output of MAS is compared with expected result. In case of same output there is no fault and test case is considered as a pass. A failed test case is further diagnosed to identify which type of fault has occurred. Type of fault identification depends on the type of node not traversed or not triggered. Chapter 7 presents expected results with actual result of test cases for evaluation purpose.

5.3 System Level MAS Testing Framework and Process: Goal and Plans Faults

Identification & Coverage:

This section describes our testing framework and testing process of system level testing of MAS. Our target is to ensure thorough coverage of plans and goals for MAS with reference to faults that may occur if certain coverage is missed. Goals are defined in System Analysis phase and their applicable plans are defined in process diagrams. Process diagram proceeded by Agent and Capability overview diagram will be used to link goals identified in certain scenario to their respective plans and sub-goals from a plan.

Figure 5.8 shows overall testing process of goals and plans fault identification and coverage, in which maximum coverage of Prometheus design artifacts for test model construction and faults identification has been 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. Identify coverage criteria and then apply on test model for test paths generation. Coverage criteria which are applied on test model are also defined in sub-sections to reveal possible faults. Test paths are generated against each coverage criteria. Test paths will lead to the generation of test cases and semi-automatic generation of test data.

Fault occurrence can cause the MAS to deliver an unexpected outcome which is evaluated with reference to goals and plans execution. Fault model was identified in Section 4.1.2, by considering possible faults that can appear in MAS with reference to goals and plans coverage.

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.

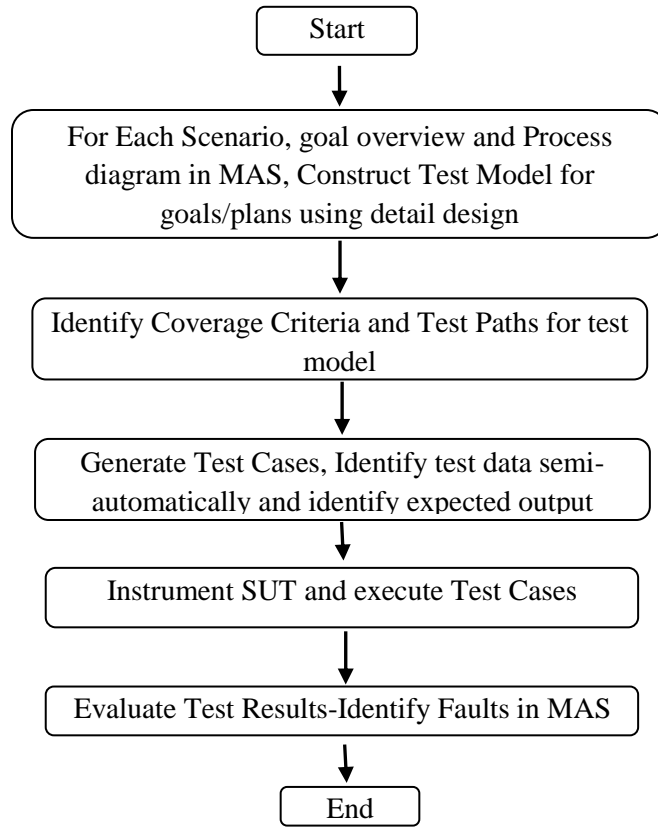


Figure-5.8: Overview of Testing Process for Goal and Plan Coverage

Figure 5.9 shows testing framework for goal, sub-goals and plans faults identification and coverage in MAS having five main processes, i.e. Goal-Plan Graph generation, test paths generation, test case generation, test case execution and test case evaluation. We have used design artifacts of Prometheus methodology as it is a rich methodology for MAS designing. Goal-Plan Graph (test model) will be annotated to show complete trace from scenario to interaction protocol then process diagram and ending in plans lies under an agent’s capability. We measure goal coverage against execution of plans for specific goal. Different design artifacts have been used in our fault identification approach. Sub-subsequent sections elaborate each process, i.e., test model generation, coverage criteria definition, test paths generation, test case generation, test case execution and test case evaluation for MAS testing with reference to goals and plans.

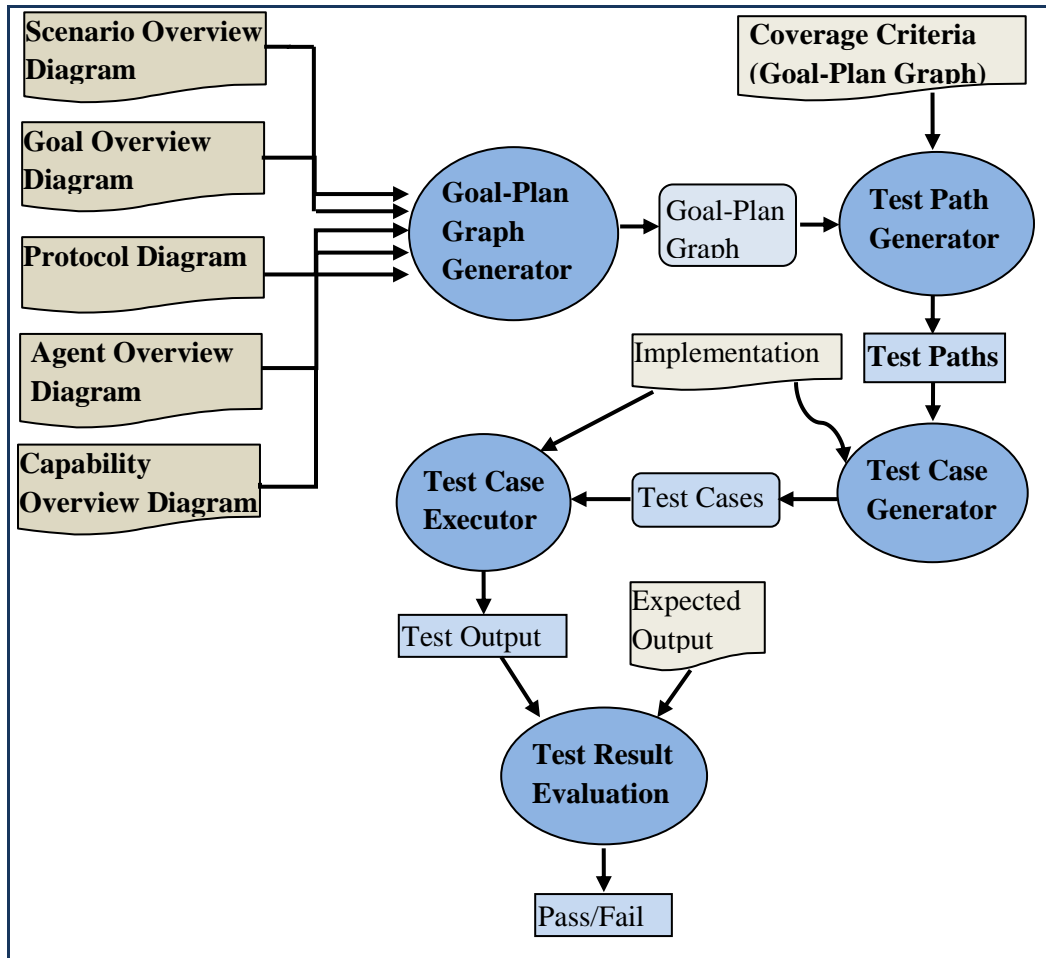


Figure-5.9: Testing Framework for Goal, Sub-Goals and Plans Coverage

Test paths are generated automatically from test model while test cases are generated for MAS implementation and they are executed to reveal injected faults. Test paths are used for generation of test cases, elaborated in subsequent sections. Test data are generated semi-automatically for test case execution. Expected output will be calculated manually for evaluation purpose. 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 test path has deviated and gave wrong output. Test result evaluation is performed manually.

5.3.1 TEST MODEL GENERATION

Goal and Plans are the factors used to measure the correctness of MAS working. In Prometheus methodology, goals are defined at system specification level in scenario overview diagram and further elaborated in goal overview diagram. Every MAS has a goal diagram for each scenario.

A scenario contains; goal, actions and percepts that can occur specific to scenario. In the remaining section we will discuss design diagrams that are created using Prometheus Design Tool (PDT). Our approach use PDT design artifacts in forming test model.

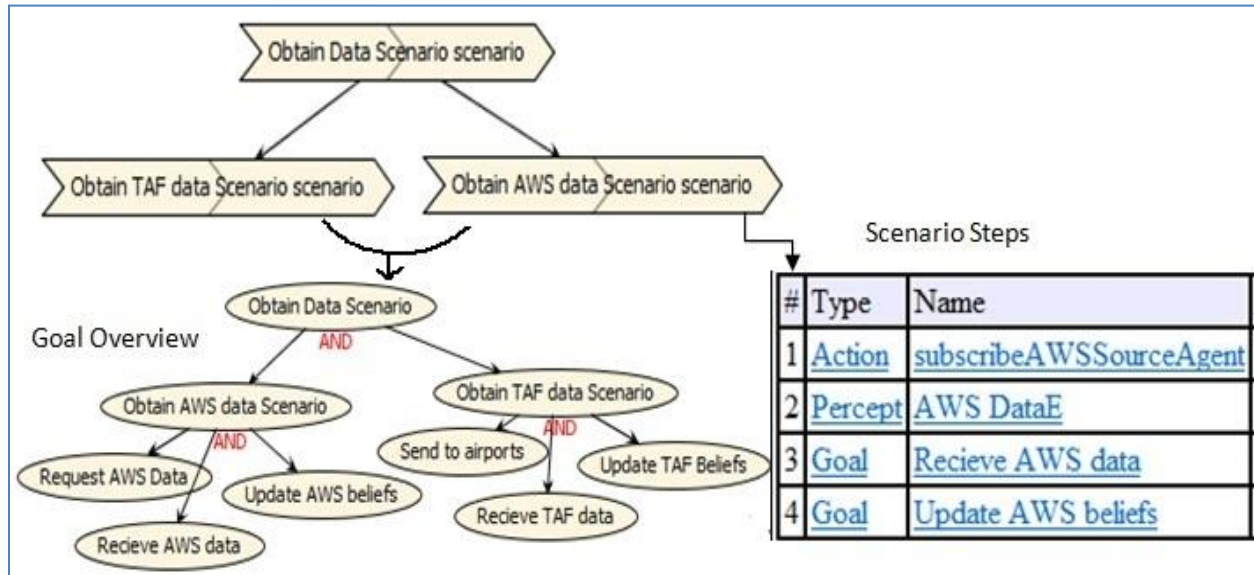


Figure 5.10: Example of Scenario and Goal overview diagram (Obtain Data Scenario)

Figure 5.10 is part of example scenario overview and goal overview diagrams for meteorological alerting system (Mathieson et al. 2004), a multi-agent system to present diverse form of weather warnings to users based on forecasts and actual sensor readings received from different airports. Multiple users can subscribe to one or more airports and receive warnings for specific situations on specific airports. As depicted in Figure 5.10, scenario can have sub scenarios as well. It shows how a scenario diagram's goals are represented in goal overview diagram along with compulsory and optional operators 'AND' and 'OR'. Figure 5.11 is the goal overview diagram of meteorological alerting system.

Goals and sub goals which are identified at specification level diagrams have their plans in detailed design in PDT. Detailed design of Prometheus methodology contains process diagrams i.e. capability and agent overview diagrams. 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. 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.

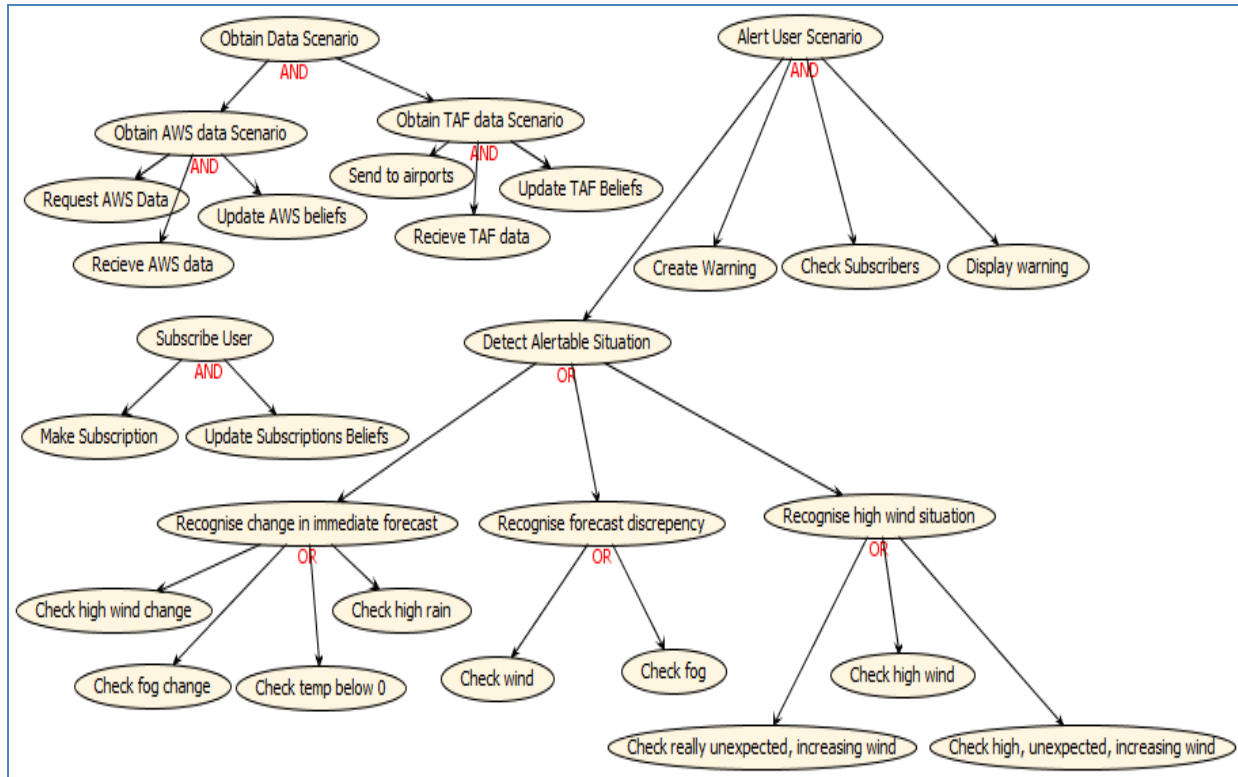


Figure 5.11: Goal Overview Diagram of Meteorological Alerting System (Mathieson et al. 2004) Notations used in diagram are also shown in figure 5.12 to have insight of each Prometheus design diagrams.

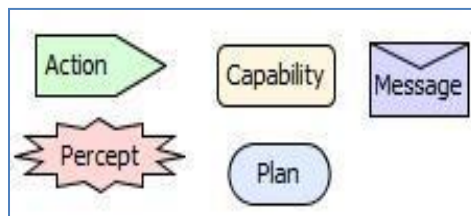


Figure- 5.12: MAS Notations Used in Prometheus Design Diagrams

Figure 5.13 shows an agent overview diagram of a meteorological alerting system that contains plans and capabilities. A capability can have multiple plans in it, satisfying a certain goal set. A percept or message event can be used to trigger a plan or capability. Plan or capability output is shown in the form of an action that affects the environment. Interaction between agents includes percepts, actions and messages all of which are modeled in a protocol diagram. Protocol diagram also contains different loops in it. Protocol diagram is used to get only loops information in order to add loops in our test model.

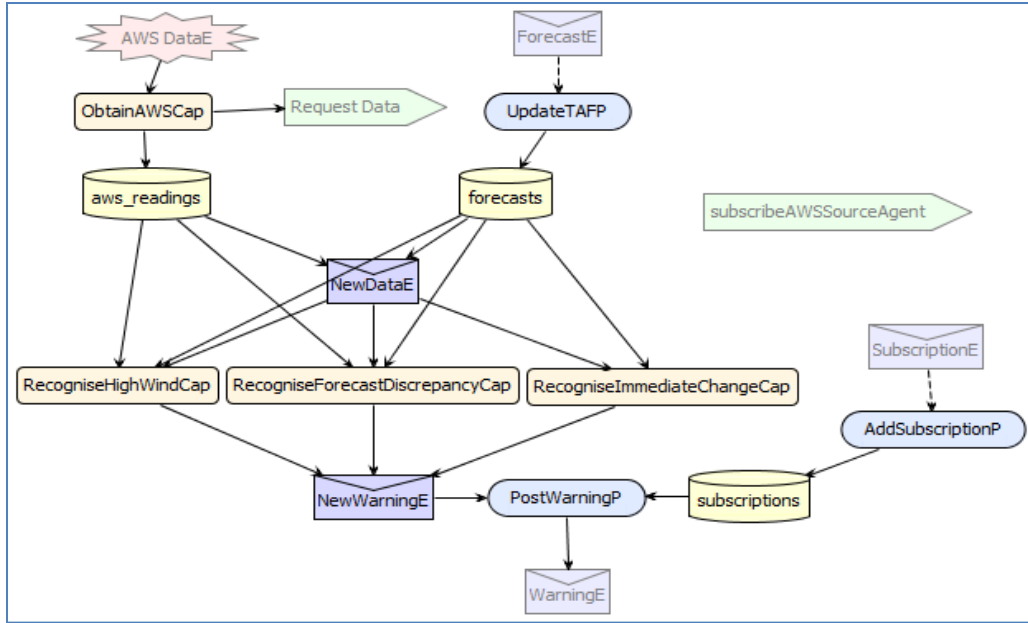


Figure- 5.13: Agent Overview Diagram (AirPort Agent) (Mathieson et al. 2004)

We have generated a test model which uses Prometheus design artifacts, e.g., scenario overview diagram, goal overview diagram, protocol diagram (only for loops), agent and capability overview diagrams. Goal-Plan Graph generator process of testing framework uses artifacts details to generate test model which is Goal-Plan Graph (GPG) for MAS. These design artifacts contain rich information from goals identification to assignment of plans for that goal. Algorithm II presented in implementation chapter 6 Section 6.2 is applied to generate GPG from design model. Algorithm II takes Prometheus design artifacts as input, extract and process goals and plans information, and generates a test model which is the GPG. It extracts sub-goals from the body of plan using process diagrams, i.e., agent and capability diagrams; and adds sub-goal to GPG as they are listed in goal overview diagram. It generates a list of all goals and plans 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.

Each scenario has its own goal overview diagram; we build GPG for each scenario and link different GPGs of the system by looking at their working in agent overview diagram. GPG consists of nodes and edges where nodes are of two type i.e., goal node and plan node. Each goal can have more than one applicable plans where all applicable plans can have ‘AND’ or ‘OR’ relationships depicted 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 always start from an arrow from a plan node to a goal node somewhere earlier in the graph. Each node contains metadata which includes scenario, agent and/or capability. Each node has a relevant scenario, agent and capability associated with it, all of which are also annotated with the node. GPG shows the complete flow of system from high level goal to detailed sub-goals and plans execution. Every plan and goal belongs to some scenario and is performed by some agent and capability belonging to the agent. Such detail of node type is also included in GPG nodes as metadata of a node.

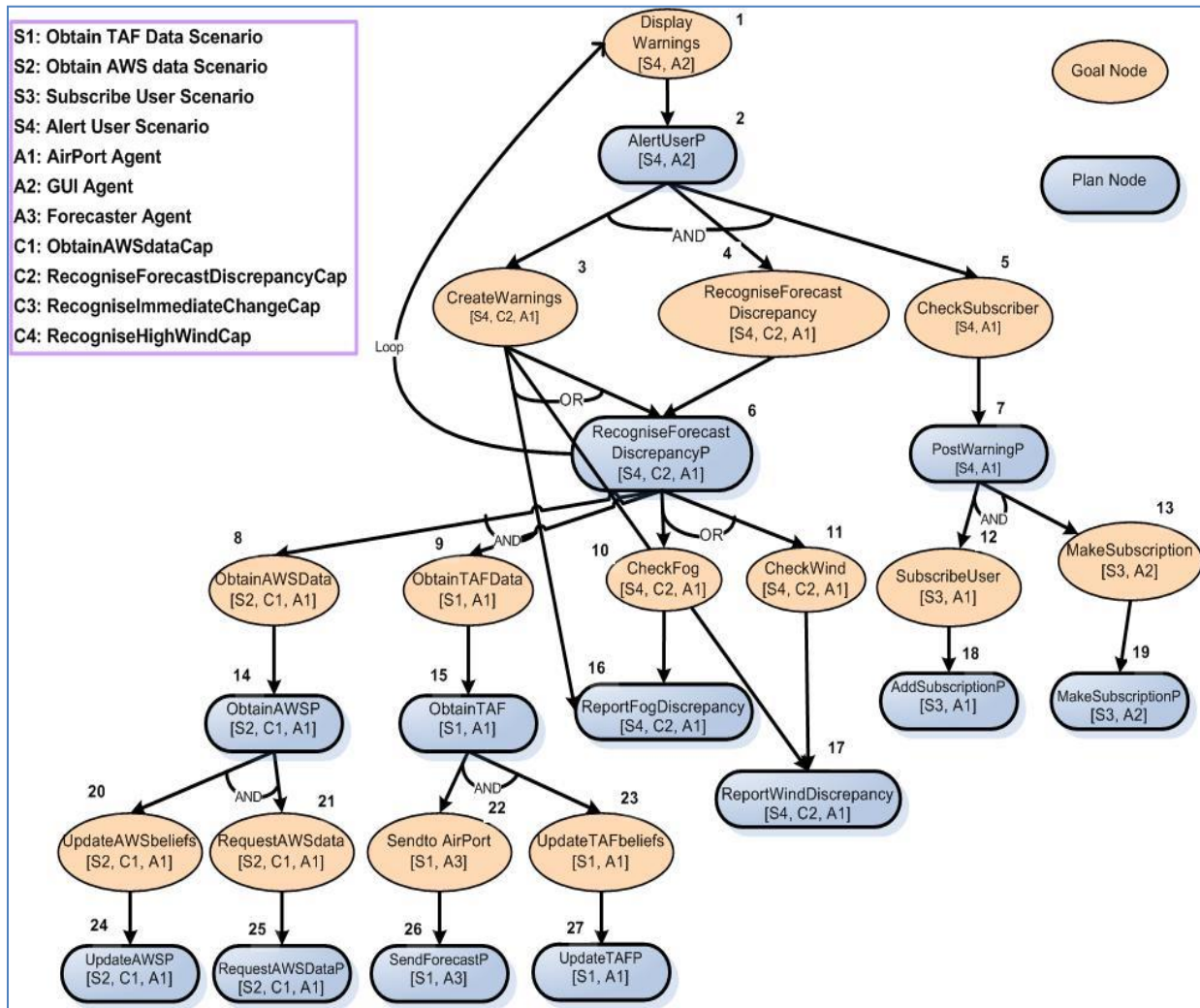


Figure- 5.14: Goal-Plan Graph (Test Model) of MAS

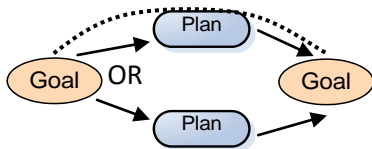
Figure 5.14 is the generated test model, i.e., Goal-Plan Graph that uses most of the design artifacts by following Algorithm II. Generated Goal-Plan Graph will be used as test model and

test paths will be generated. These generated paths will be used for test case generation. Injected faults will be identified by applying different coverage criteria discussed in the next section. Test cases are constructed from test model by extracting variables used in a certain test path node and assigning possible inputs as the test data that will be executed on the MAS implementation.

5.3.2 COVERAGE CRITERIA

It is possible that some parts of system under test (SUT) remains untested which may cause problems in the MAS operation. Coverage is of MAS goals and plans essential for reliability. 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. Coverage is measured by actual system execution and then traces are checked on test model accordingly. MAS possesses 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.

- i. **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 $P \in TP$.



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 above sketch if OR is the constraint then only one path coverage is enough, in this case only all goals are traversed.

- ii. **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 $P \in TP$.

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

- iii. **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 $P \in TP$.

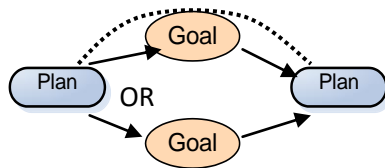
Test path(s) in which every agent has been traversed at least once.

- iv. **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 $P \in TP$.

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

- v. **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 $P \in TP$.

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



- vi. **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 $P \in TP$.

Test path(s) in which every goal and its all applicable plans (arcs of GPG) must be covered at least once. It will cover OR condition branches as well.

- vii. **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 $P \in TP$.

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. To test such functionality loop coverage is required 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.

Goal, sub-goal and plan related fault model is presented in Section 4.1.2 which can be identified, if present in MAS, by applying coverage criteria on test model. Each coverage criteria identify one or more than one fault types. Coverage criteria ensure certain types of faults detection and identification within a system (Tian, 2001). Following is the relationship of coverage criteria with fault types that depicts which coverage criteria can reveal which types of faults in MAS.

Coverage criteria → types of faults that could be identified by the coverage criteria.

All goals Coverage → Inaccurate goal achievement

Scenario Coverage → Scenario Fault

Agent Coverage → Missing functionality, Deliberate Fault

Capability Coverage → Internal Agent fault

Plan Coverage → Plan Failure, Deliberate Fault

Goal Plan Coverage → Inaccurate goal achievement, Plan Failure, Missing functionality

Loop Coverage → Missing functionality

In results and discussion chapter we will provide results of applying coverage criteria on a case study to validate their relationship with types of faults identified.

5.3.3 TEST PATHS GENERATION

Test paths are generated from test model, in our case it is Goal-Plan Graph constructed in Section 5.3.1. Algorithm III presented in implementation section 6.3 is used for automated test paths generation for each coverage criteria. 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 figure 5.14. Algorithm also IV makes a list of ‘AND’ and ‘OR’ constraints on edges. 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. Automated test paths generation tool is presented in section 6.3.1. Following is the structure used for test model input used in test path generation tool:

{Node Name, Node Metadata, Node type (G/P), Node No, AND/OR constraint}

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 of our test model. Table-3 shows paths generated by our test path generation tool by using GPG of Figure 5.14 as test model.

Table-3: Test Paths for Each Coverage Criteria Applied on GPG

S. No	Coverage Criteria	Test Paths
1	All goals Coverage	1(goal)→2→3 (goal)→6→10(goal)→16 1(goal)→2→3 (goal)→6→11(goal)→17 1(goal)→2→5(goal)→7→12(goal)→18 1(goal)→2→5(goal)→7→13(goal)→19 1(goal)→2→4(goal)→6→9(goal)→15→22(goal)→26 1(goal)→2→4(goal)→6→9(goal)→15→23(goal)→27 1(goal)→2→4(goal)→6→8(goal)→14→20(goal)→24 1(goal)→2→4(goal)→6→8(goal)→14→21(goal)→25
2	Scenario Coverage	1(S4)→2(S4)→5(S4)→7(S4)→12(S3)→18(S3) 1(S4)→2(S4)→3(S4)→6(S4)→9(S1)→15(S1)→23(S1)→27(S1) 1(S4)→2(S4)→3(S4)→6(S4)→8(S2)→14(S2)→21(S2)→25(S2)
3	Agent Coverage	1(A2)→2(A2)→3(A1)→6(A1)→9(A1)→15(A1)→22(A3)→26(A3)
4	Capability Coverage	1→2→4(C2)→6(C2)→8(C1)→14(C1)→21(C1)→25(C1)
5	Plan Coverage	1→2(Plan)→3→16(Plan) 1→2(Plan)→3→17(Plan) 1→2(Plan)→5→7(Plan)→12→18(Plan) 1→2(Plan)→5→7(Plan)→13→19(Plan) 1→2(Plan)→4→6(Plan)→9→15(Plan))→22→26(Plan) 1→2(Plan)→4→6(Plan)→9→15(Plan))→23→27(Plan) 1→2(Plan)→4→6(Plan)→8→14(Plan))→20→24(Plan) 1→2(Plan)→4→6(Plan)→8→14(Plan))→21→25(Plan)
6	Goal Plan Coverage	1(goal)→2(Plan)→5(goal)→7(Plan)→12(goal)→18(Plan) 1(goal)→2(Plan)→5(goal)→7(Plan)→13(goal)→19(Plan) 1(goal)→2(Plan)→3(goal)→16(Plan) 1(goal)→2(Plan)→3(goal)→17(Plan) 1(goal)→2(Plan)→3(Plan)→6(Plan)→10→16(Plan) 1(goal)→2(Plan)→3(goal)→6(Plan)→11(goal)→17(Plan) 1(goal)→2(Plan)→3(goal)→6(Plan)→9(goal)→15(Plan)→22(goal)→26(Plan) 1(goal)→2(Plan)→3(goal)→6(Plan)→9(goal)→15(Plan)→23(goal)→27(Plan)

		1(goal)→2(Plan)→3(goal)→6(Plan)→8(goal)→14(Plan)→20(goal)→24(Plan) 1(goal)→2(Plan)→3(goal)→6(Plan)→8(goal)→14(Plan)→21(goal)→25(Plan) 1(goal)→2(Plan)→4(Plan)→6(Plan)→10(goal)→16(Plan) 1(goal)→2(Plan)→4(goal)→6(Plan)→11(goal)→17(Plan) 1(goal)→2(Plan)→4(goal)→6(Plan)→9(goal)→15(Plan)→22(goal)→26(Plan) 1(goal)→2(Plan)→4(goal)→6(Plan)→9(goal)→15(Plan)→23(goal)→27(Plan) 1(goal)→2(Plan)→4(goal)→6(Plan)→8(goal)→14(Plan)→20(goal)→24(Plan) 1(goal)→2(Plan)→4(goal)→6(Plan)→8(goal)→14(Plan)→21(goal)→25(Plan)
7	Loop Coverage	1(goal)→2(Plan)→3(Plan)→6(Plan)→10→16(Plan) 1(goal)→2(Plan)→3(Plan)→6(Plan)→1(goal)→2(Plan)→3(goal)→6(Plan)→11(goal)→17(Plan) 1(goal)→2(Plan)→3(Plan)→6(Plan)→1(goal)→2(Plan)→3(goal)→6(Plan)→1(goal)→2(Plan)→4(goal)→6(Plan)→8(goal)→14(Plan)→21(goal)→25(Plan)

5.3.4 TEST CASE GENERATION AND EXECUTION

Test 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 node and use the variables or properties associated at each node for test case generation. Properties or triggering function/variables are used and in system implementation; test cases consist of value combinations of variables that make a certain path to follow. For example, for the test path 1(goal)→2(Plan)→3(goal)→ 6(Plan) →11(goal)→ 17 (Plan), table-4 is NDT for a test path.

Table-4: Node Description Table for Test Paths Nodes

Node No.	Node Type	Associated Variables/functions
1	Goal (display Warnings)	String = warning name Int = Value
2	Plan (Alert UserP)	Triggering event = Yes String = level (Severe/Normal)

3	Goal (Create warnings)	Int = Value > threshold
6	Plan (CreateForecast DiscrepanceP)	String = warning name Int = value Fuct. = CreateForecast
11	Goal (Check Wind)	String = warning (high wind)
17	Plan (Report Wind Discrepancy)	String = Wind Int = 60 Mph

GPG nodes associated properties are extracted from its implementation. In result and discussion chapter we construct test cases and then execute them to validate the technique. Figure 5.15 shows test case generation process for MAS under test.

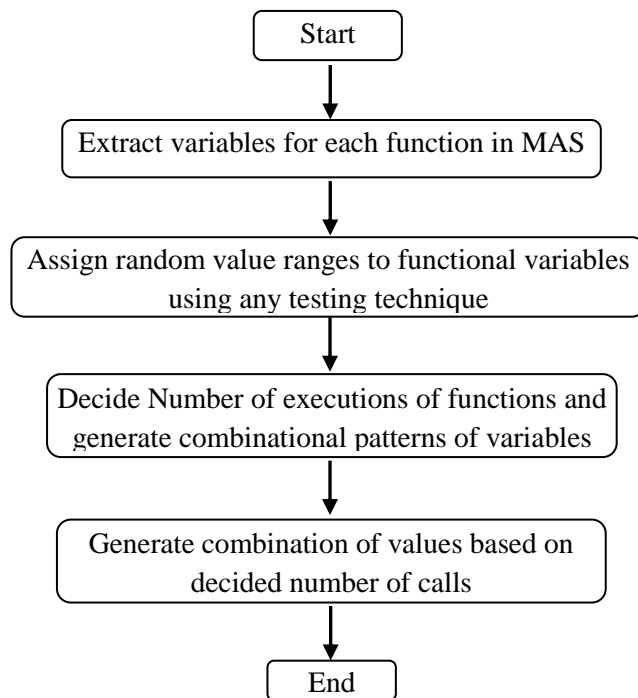


Figure - 5.15: Test Case Generation Process for MAS

A MAS has many functions that are called or triggered to get a desired output. Test cases are used to execute the MAS to get the desired output and to check what types of faults are identified. To assign values to function variables there is a need to first identify function variables and then generate proper values for the variables. The number of test cases to execute depends on generated value combinations for each variable and number of generated patterns for

variables in the MAS to execute. For each coverage criteria different test cases can be generated with different value combinations.

The process of test case execution is semi automatic. Once the test cases have been generated after extracting variables and value ranges for the variables from Node description table, we jump to the test case execution process. Assigning values to test cases extracted variables is semi-automatic. Test case execution process requires several variable set up values involved in a test cases. Our testing framework allows test engineers to access the implementation and assign range of possible values to be used in test case. Once values have been assigned then combination of execution is hard coded into MAS implementation, execution trace is the output of the test case execution. Random numeric values for integer variables are stored in an array and that are assigned to variables at the time of execution automatically. Result and discussion chapter elaborates test case execution with real time values assigned to variables.

We further evaluate this approach on a case study in the result and discussion chapter to validate the correctness of fault identification approach. Types of faults discovered by a coverage criteria are also discussed in chapter 7.

5.3.5 TEST RESULT EVALUATION: GOAL-PLAN COVERAGE

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. We have the expected output against a certain test case. Expected output will also be calculated manually. Against a certain input, MAS have to produce some output after executing its plans. Specific path is followed against certain input. Based on the event and plan execution of the system we calculate expected output for each test case. For example in Metrological Alerting System presented earlier in this chapter, if wind pressure is normal and fog discrepancy level is also normal then expected output is normal situation and no alert situation is generated. Expected outputs are also calculated for our case study in result and discussion chapter 7.

Test result evaluation process is manual. We have calculated expected output of MAS and run identified test cases. 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 the fault in MAS. We have identified fault types in our fault model and these faults were injected in MAS implementation. Test case output will reveal faults identified 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 reveal certain faults identified earlier in Chapter 4. Detailed test result evaluations with faults are presented in the results and discussion chapter 7.

CHAPTER 6: MAS TESTING FRAMEWORK IMPLEMENTATION

In this chapter, we will explain details of implementation that have been done to automate MAS testing framework presented in previous chapter. Different algorithms have been designed for test model generation and test paths generation, we will present algorithms and corresponding tools architecture in subsequent sections of the chapter. As described in our testing framework, we have separate algorithms and tools for test protocol graph generation, goal-plan graph generation and test paths generation for both test models.

6.1 Interaction Testing

Interactions of MAS are presented in protocol diagram. We use protocol diagram AUMML description as input and convert it into protocol graph that is used as test model for interaction testing of MAS. Figure 6.1 shows the architecture to convert protocol graph into protocol graph.

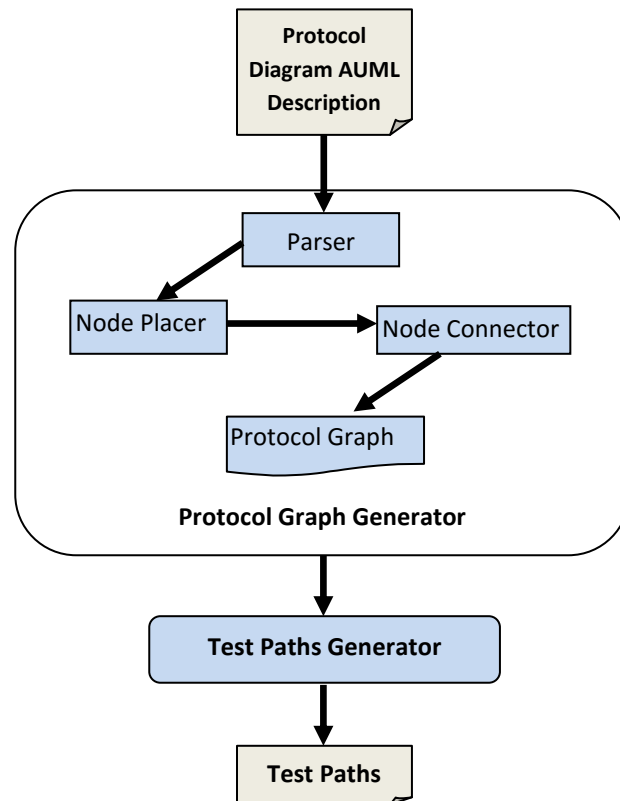


Figure - 6.1: Protocol Graph and Test Path Generator Tool Architecture

After parsing the AUML description of protocol diagram, node placer place identified node of each type and node connector links the node with each other. ATL transformation rules presented in Section 6.1.1 is followed for PD meta-model to PG meta-model generation. Test paths generator process in explained in detail in section 6.1.2.

6.1.1 ATL Transformation Rules PD meta-model to PG meta-model

To automate the transformation process, we have used Atlas Transformation Language (ATL) (ATL 2008). ATL is a model transformation language that allows specification of rules to transform source models, to target models.

Module PD2PG;

Create OUT:PGMM **from** IN:PDMM;

helper def: getprimitiveTypeMapping : Map(**String**, **String**) = Map
{ ('**String**', '**String**'), };

rule Create AUMLProtocolInstance {
from inputModel : ProtocolDiagram!MM
to PGMM : GraphMM!ProtocolGraph (name <-inputModel.name,
Protocol <-inputModel.modelElements)}

rule PelementToNode {
from inputModel : ProtocolDiagram!MM
to PGMMPelement : GraphMM!Nodes
do

(
NodeName <- Pelement.name;
type <-clAttribute.type.name,
)

helper def: getprimitiveTypeMapping : Map(**String**, **String**) = Map

('**Message**', '**Message**'); ('**Label**', '**Percept**'); ('**GoTo**', '**Action**');

}

rule CreateAssociation {
from inputModel : ProtocolDiagram!MM
to PGMMassociation : GraphMM!Associations
do (
name <- 'Association_'+PGMM+'_'+PGMM,
ownedEnd <- Set { new_ownedEnd, new_ownedEnd1 }
)

new_ownedEnd : PGMM!Property(
name <- PDMM.toLower(),
type <- thisModule.classNameBy.get(PDMM)

```
)
aggregation<- #composite
)
do{ association; }}
```

6.1.2 Protocol Graph Test Paths Generator

Test path generation takes protocol diagram and coverage criteria as input and generates test paths for protocol. Algorithm-II is designed for test paths generation.

Algorithm I: Test Path generation from Protocol Graph

Input: Coverage Criteria (A set of defined coverage criteria), Graph (Set of nodes and edges)

Output: Test Paths

- Step 1: Build an **edge list** and **node list** of graph
 - Step 2: Categorize node with respect to type
 - Step 3: if all paths from graph = empty
 - Step 4: **find_paths from graph**
 - Step 5: **End if**
 - Step 6: Sort the paths in ascending order of the path length ending
 - Step 7: **if** current path = selected coverage criteria
 - Step 8: append (current path) in result
 - Step 9: **End if**
 - Step 10: Print Result
-

```
def find_paths(names, graph, start, end, pathof=
    ['start', 'end', 'message', 'action', 'precept'],
    path=[]):
    path = path + [start]
    if start == end:
        return [path]
    if not graph.has_key(start):
        return []
    paths = []
    for node in graph[start]:
        if names[node][1] in pathof:
            if path.count(node) < 2:
                newpaths = find_all_paths(names, graph,
                    node, end, pathof, path)
                for newpath in newpaths:
                    paths.append(newpath)
    return paths
```

Figure - 6.2: Code of Finding Paths from Protocol Graph

Protocol graph is used as the input and according to coverage criteria test paths according are generated. We have a test path generation tool for automated test paths generation, tool architecture is presented in the next sub section. Figure 6.2 shows code of finding paths from protocol graph used in our tool implementation.

6.1.3 Interaction Test Paths Generator Tool

Our test paths generation tool takes protocol diagram as input and generates test paths. Test Path Generator tool has two main classes namely Graph Regeneration and Graph Parser.

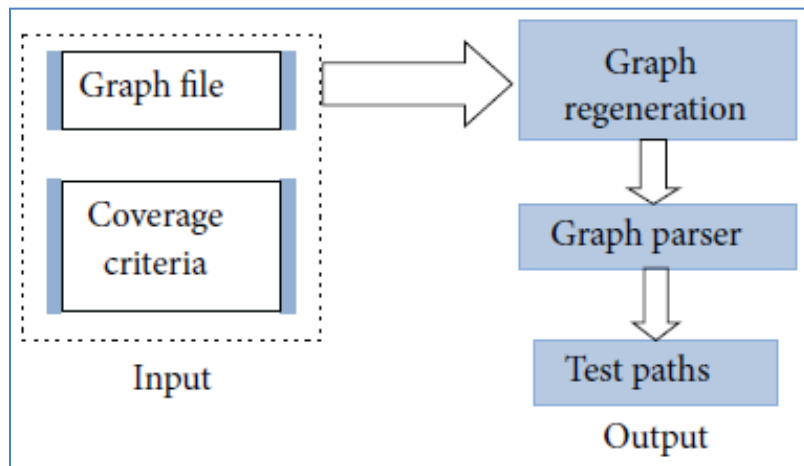


Figure-6.3: Test Path Generator Tool Process

14	1,2	8,9
start, start	2,3	9,10
account open, precept	3,2	9,11
account created, action	3,4	9,12
debit account request, precept	3,5	10,13
credit account request, precept	4,6	11,13
transport request, message	4,10	12,14
exchange request, message	4,11	12,4
exchange rates, precept	4,12	12,5
exchange request reply, message	5,6	13,14
amount credited, action	5,10	13,4
amount debited, action	5,11	13,5
request error, action	5,12	
account info, action	6,7	
end, end	7,8	
27		

Figure-6.4: Test Path Generator Tool Input File

Graph Regeneration reads the input file and makes a graph object according to the file. This object is used in the program to produce the paths. Graph Parser searches all the possible paths according to the coverage criteria given to it. Figure 6.3 shows the process of test paths generation tool. Figure 6.4 shows input file (graph file of protocol graph presented in figure 7.9) for test path generator tool. Input contains protocol graph details in textual form.

Some screen shots of test path generator tool based on the coverage criteria is shown in Figures 6.5-6.9.

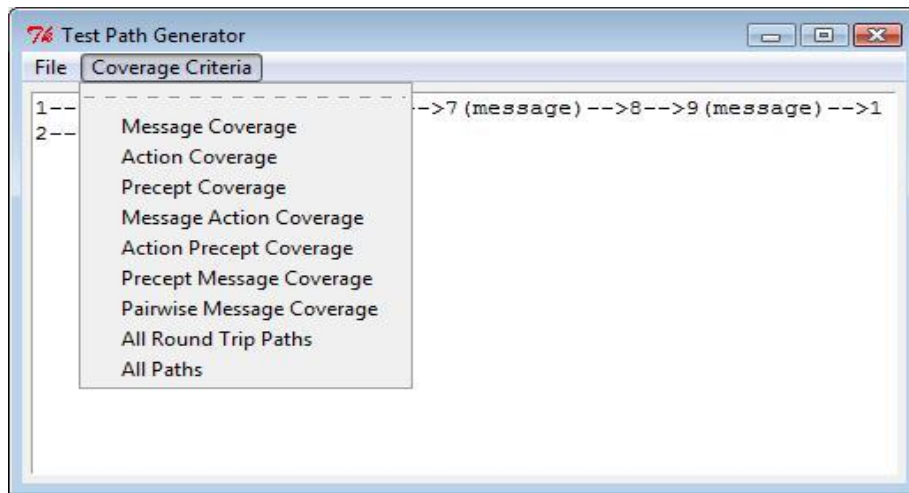


Figure- 6.5: Test Path Generation Tool (Coverage Criteria)

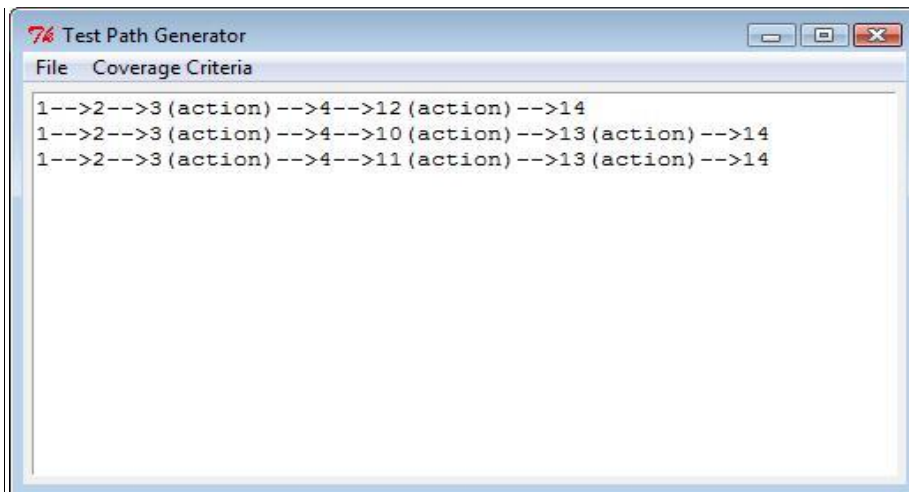


Figure- 6.6: Test Path Generation Tool (Action Coverage)

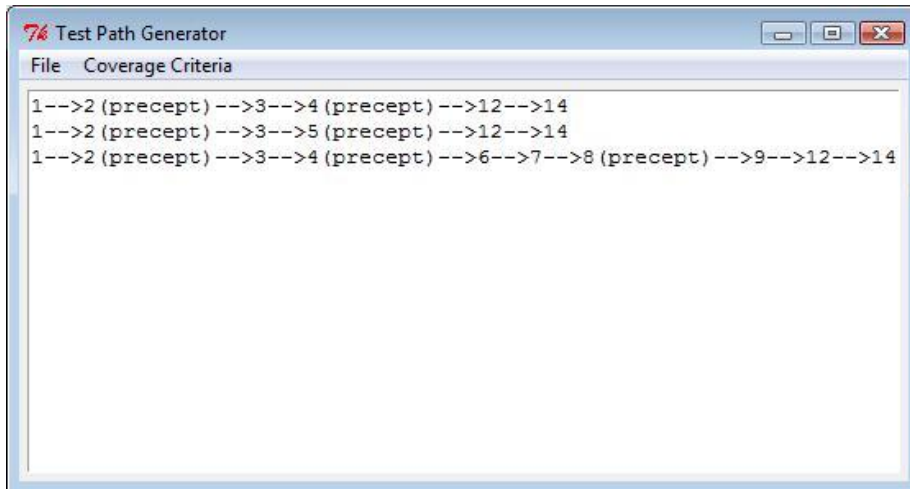


Figure- 6.7: Test Path Generation Tool (Percept Coverage)

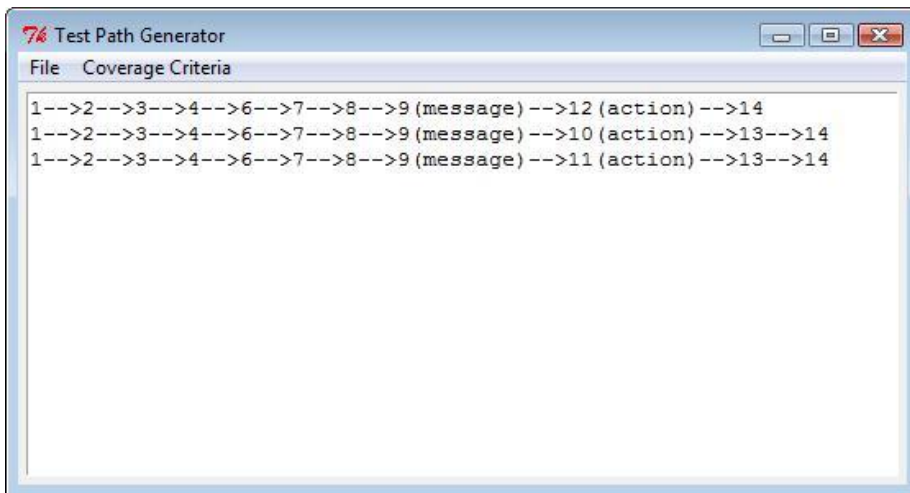


Figure- 6.8: Test Path Generation Tool (Message Action Coverage)

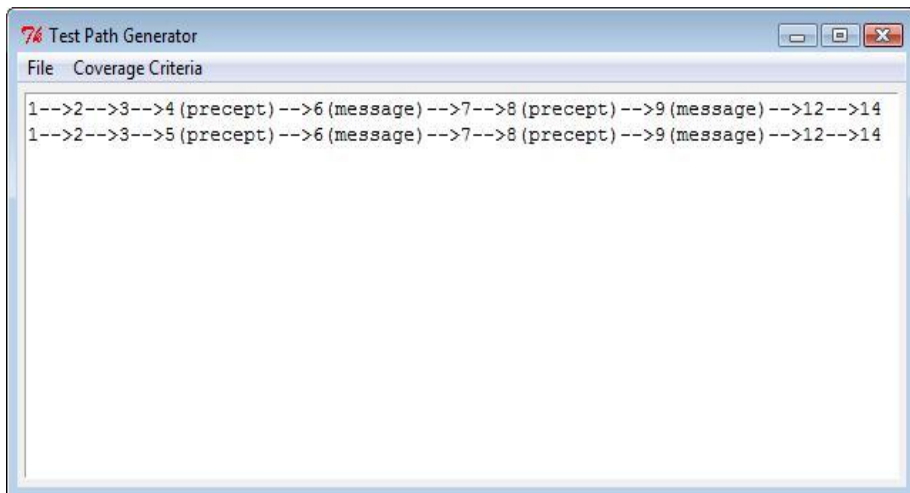


Figure- 6.9: Test Path Generation Tool (Percept Message Coverage)

6.2 System Testing: Goal-Plan Graph Generator

Goal and Plans are the key artifacts of MAS working. In Prometheus methodology, goals are defined at system specification level and their relevant plans are defined in process diagrams. We have used design diagrams to generate test model for goal and plans coverage. Algorithm II is designed that takes design artifacts defined in testing framework as input, extracts and processes goals and plans information, and generates a test model which is the GPG. It extracts sub-goals from the body of plan using process diagrams, i.e., agent and capability diagrams; and adds sub-goal to GPG as they are listed in goal overview diagram. Step by step GPG will be generated by following the listed steps in Algorithm II.

Algorithm II: 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: $GL \leftarrow GD.goals$

Step 2: Extract plans from AD and CD: $PL \leftarrow AD.plans \cup CD.plans$

Step 3: For each Goal and Plan

Step 4: Add Plan $P(G) \leftarrow$ List of Applicable Plans (G, AP)

Step 5: Add SG (P) \leftarrow List of sub-goals for Plans (P, SG).

Step 6: Add Scenario $S(G)/S(P) \leftarrow$ Scenario for Goal/Plan (G/P, AS)

Step 7: Add Agent $A(G)/A(P) \leftarrow$ Agent for Goal/Plan (G/P, AA)

Step 8: Add Capability $C(G)/C(P) \leftarrow$ Capability containing Goal/Plan (G/P, AC)

Step 9: For each Goal-Diagram against each Scenario

Step 10: Set Root (GPG) $\leftarrow GD.root$

Step 11: Set Current Goal (CG) \leftarrow Root

Step 12: Add S (G), A (G) and/or C(G)

Step 13: Add Children (CG) \leftarrow AP

Step 14: Add Constraint(G-Node) \leftarrow AND or OR

Step 15: Add S(P), A(P) and/or C(P)

Step 16: For Each Plan (P, CG)

Step 17: do Add Children (P) \leftarrow (SG, P)

Step 18: Add Constraint(P-Node) \leftarrow AND or OR

Step 19: Set $CG \leftarrow SG$
Step 20: **While** $CG \neq \{\}$
Step 21: Repeat step 11-19
Step 22: End While
Step 23: **If** Goal-Diagrams > 1 and $n = \text{Number of Scenario}$
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 \leftarrow Plan
Step 27: **Return** GPG

6.3 Goal-Plan Graph Test Paths Generator

GPG is constructed by following Algorithm-III while Algorithm-IV is defined to generate test paths from GPG for each coverage criteria. Appendix-A contains details of code for *findpathsbytype ()*, *findpathsbymetadata ()*, *findall ()* and *findloop ()* functions used in test paths generation.

Algorithm III: 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, metadata (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

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. Loop coverage method covers loop coverage criteria.

Loop Coverage

Function to cover the loop is as follows

- Parse the entire tree
- Create a list of nodes containing the loop edges
- Check for the times of occurrence of the loop nodes in the path
- If it occurs for one time then add it to simple path list and delete the occurred node from list of loop nodes
- If it occurs two times then add it to 1 loop path list and delete the occurred node from the list of loop nodes
- If it occurs three times then add it to 2 loop path list and delete the occurred node from the list of loop nodes
- As soon as all the nodes are covered break the process

Metadata Coverage

Function to cover the node with respect to metadata

- Parse the entire tree
- Create a list of metadata info
- Parse the paths one at a time for the required metadata
- When metadata is found delete it from the list and print the path
- Terminate the process when metadata list is empty

Type Coverage

Function to cover the node with respect to metadata

- Parse the entire tree
- Create a list of type of nodes to be covered
- Parse the paths one at a time for the required node
- When node is found delete it from the list and print the path
- Terminate the process when node list is empty

6.3.1 Goal-Plan Graph Test Paths Generator Tool

Figure 6.11 shows the basic architecture of automatic test paths generation by following Algorithm III. For loop coverage we created a list of nodes containing the loop edges and check its zero, one and two occurrences. Test path generation tool takes test model, i.e., GPG as input,

details of test model is stored in text file in which metadata details, nodes and edges information is stored as presented in Figure 6.10 (GPG description of figure 7.10).

3;matadata	Creditaccountplan:[s2,c1,a1];plan;7	10,4;loop
s1,s2,s3,s4,s5	Creditaccountexchangeplan:[s2,c1,a1];plan;7	10,17;or
a1,a2,a3	Creditaccounterrorplan:[s2,c1,a1];plan;7	10,16;or
c1,c2,c3	CurrencyExchange:[s5,a3];goal;8	10,15;or
32;nodes	Sendandwait:[s5,a3];plan;9	11,4;loop
obtain information:[s4,a1];goal;0	PerformExchange:[s5,a2];goal;10	11,13;or
obtain information:[s4,a1];plan;1	SetExchangerates:[s5,a2];goal;10	11,12;or
create Account:[s4,a1];goal;2	PerformExchangep:[s5,a2];plan;11	11,14;or
Account operations:[s1,a1];goal;2	SetExchangerateplan:[s5,a2];plan;11	12,18;
Accountinfoplan:[s4,a1];plan;3	ComputeRate:[s5,c3,a2];goal;12	13,19;
Create Accountp:[s4,a1];plan;3	IdentifyRateplan:[s5,c3,a2];plan;13	14,20;
Accountoperationp:[s1,a1];plan;3	TwostepRateplan:[s5,c3,a2];plan;13	15,21;
Creditaccount:[s2,c1,a1];goal;4	35;edges	16,22;
debitaccount:[s3,c2,a1];goal;4	1,2;	17,23;
Creditaccountp:[s2,c1,a1];plan;5	2,3;	19,24;
debitaccountp:[s3,c2,a1];plan;5	2,4;	22,24;
debitaccount:[s3,c2,a1];goal;6	3,5;	24,25;
debitaccountexchange:[s3,c2,a1];goal;6	3,6;	25,26;
debitaccounterror:[s3,c2,a1];goal;6	4,7;	25,27;
Creditaccount:[s2,c1,a1];goal;6	6,1;loop	26,28;
Creditaccountexchange:[s2,c1,a1];goal;6	7,8;or	27,29;
Creditaccounterror:[s2,c1,a1];goal;6	7,9;or	28,30;
debitaccountplan:[s3,c2,a1];plan;7	8,10;	30,31;or
debitaccountexchangeplan:[s3,c2,a1];plan;7	9,11;	30,32;or
debitaccounterrorplan:[s3,c2,a1];plan;7		

Figure-6.10: GPG (Test Model) Test Paths Generation Tool Input

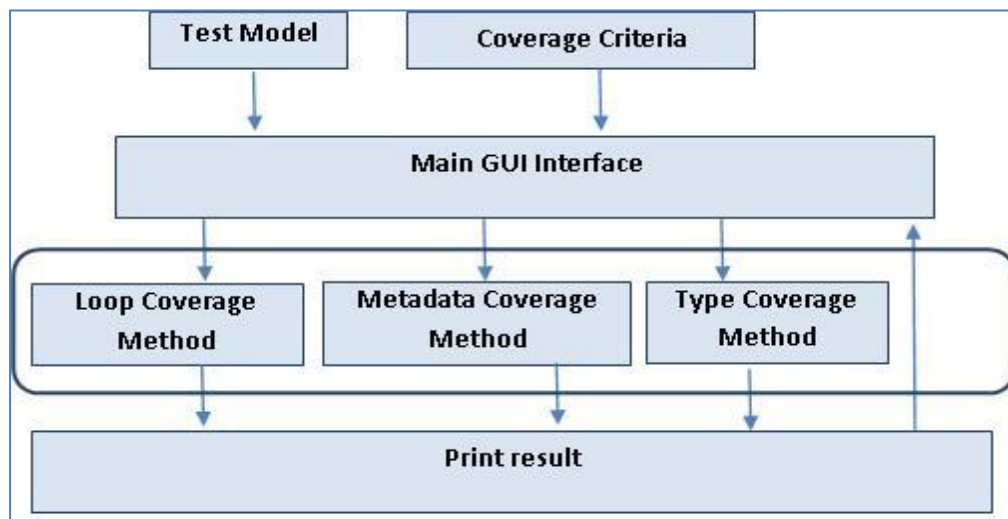


Fig-6.11: Automatic Test Path Generation Architecture Using GPG

Screen shots of test paths generated by tool from GPG for each coverage criteria are shown below.

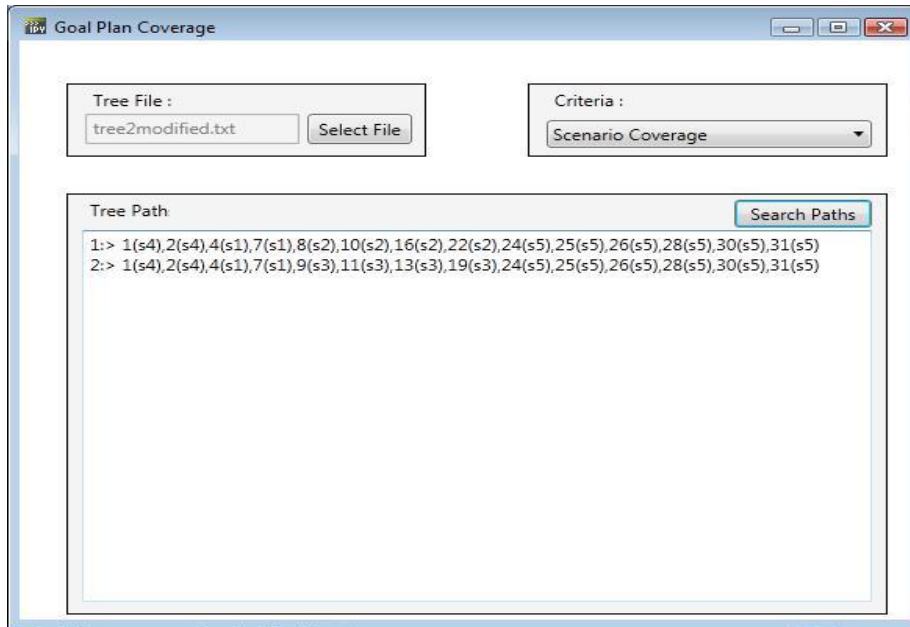


Figure- 6.12: GPG Test Paths Generator (Scenario Coverage Criteria)

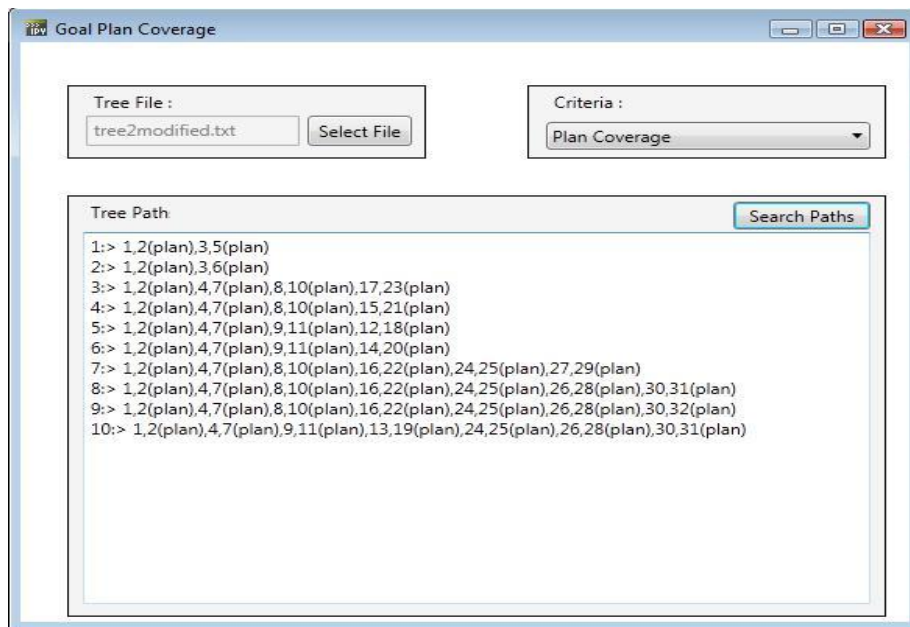


Figure- 6.13: GPG Test Paths Generator (Plan Coverage Criteria)

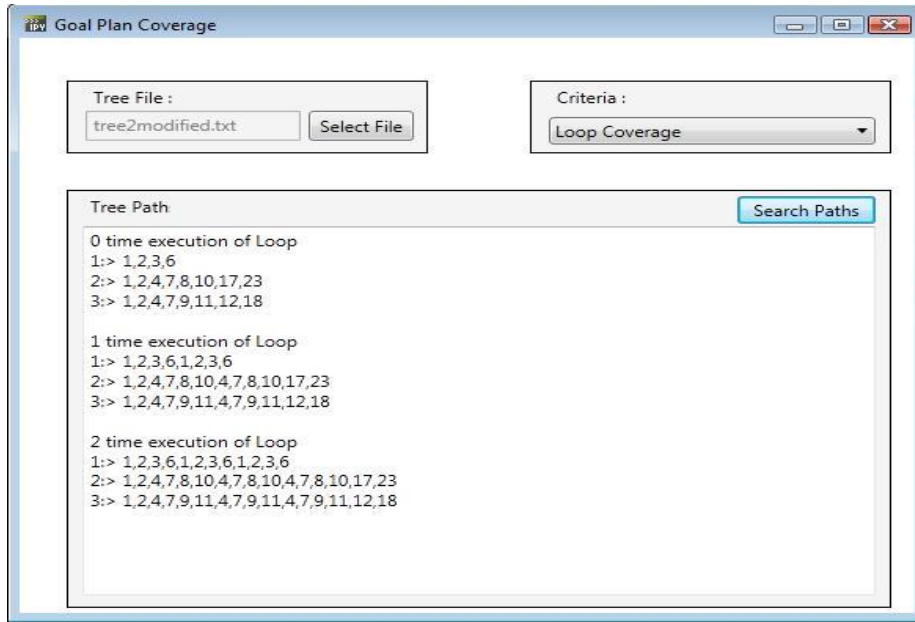


Figure- 6.14: GPG Test Paths Generator (Loop Coverage Criteria)

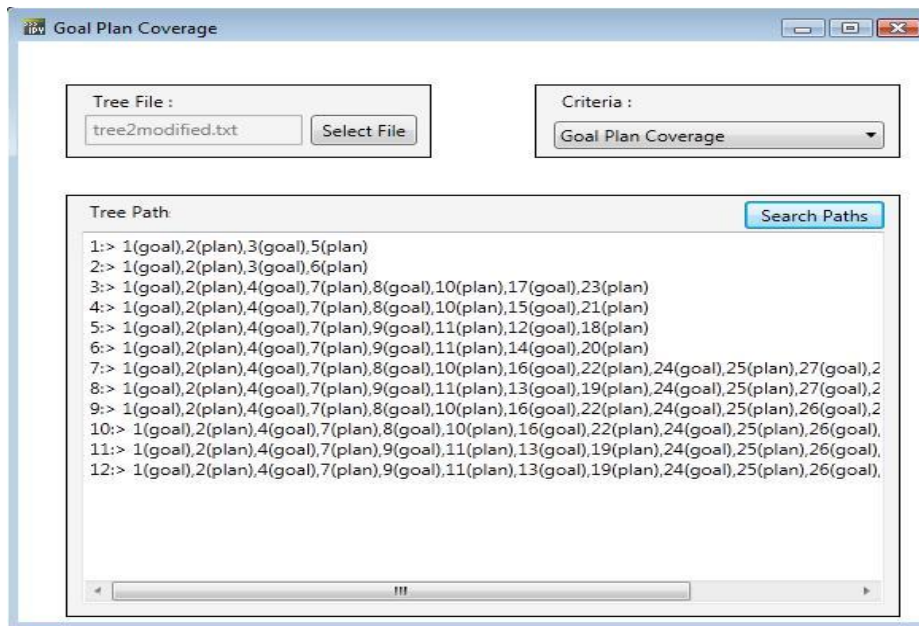


Figure- 6.15: GPG Test Paths Generator (Goal-Plan Coverage)

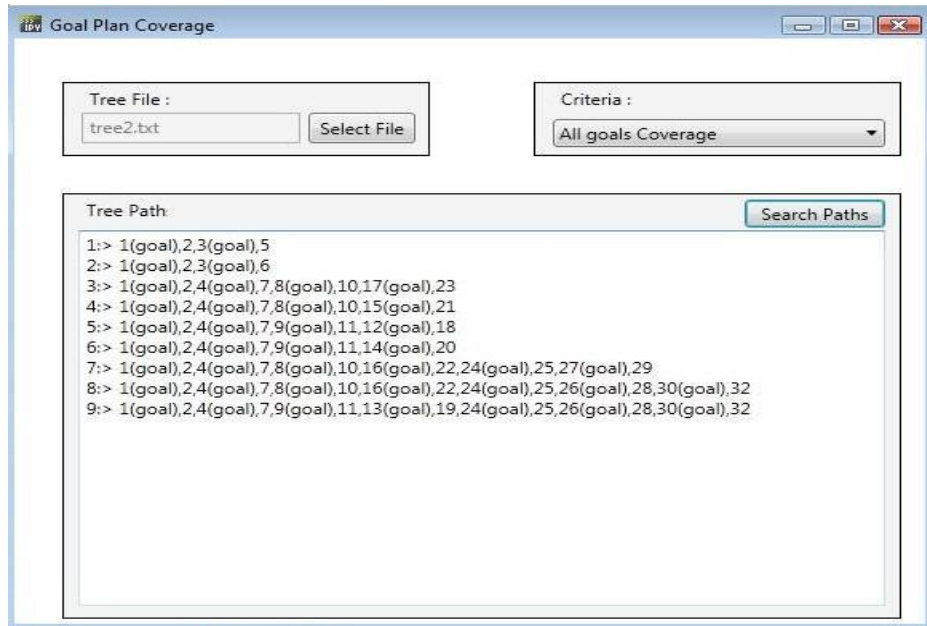


Figure- 6.16: GPG Test Paths Generator (All Goals Coverage)

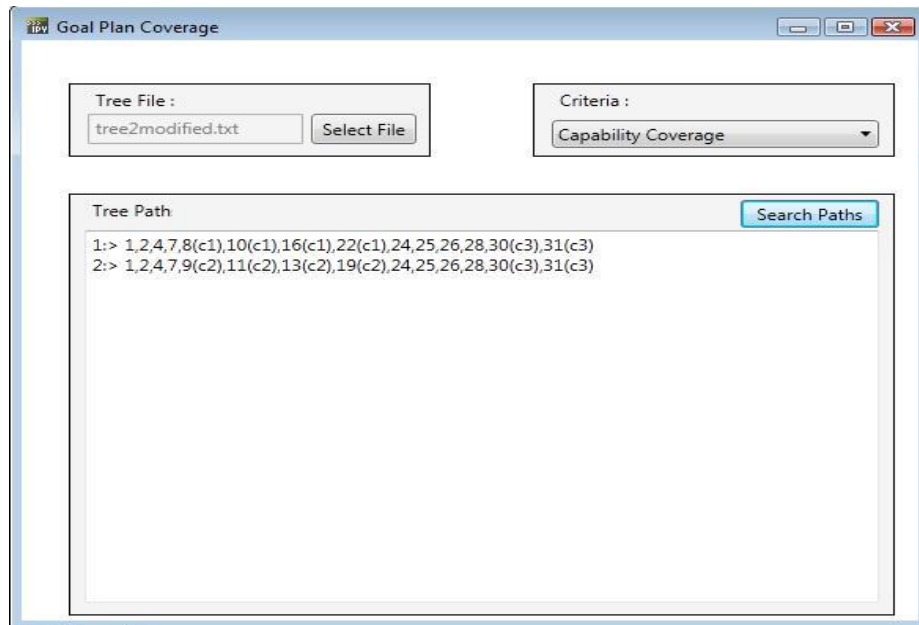


Figure- 6.17: GPG Test Paths Generator (Capability Coverage)

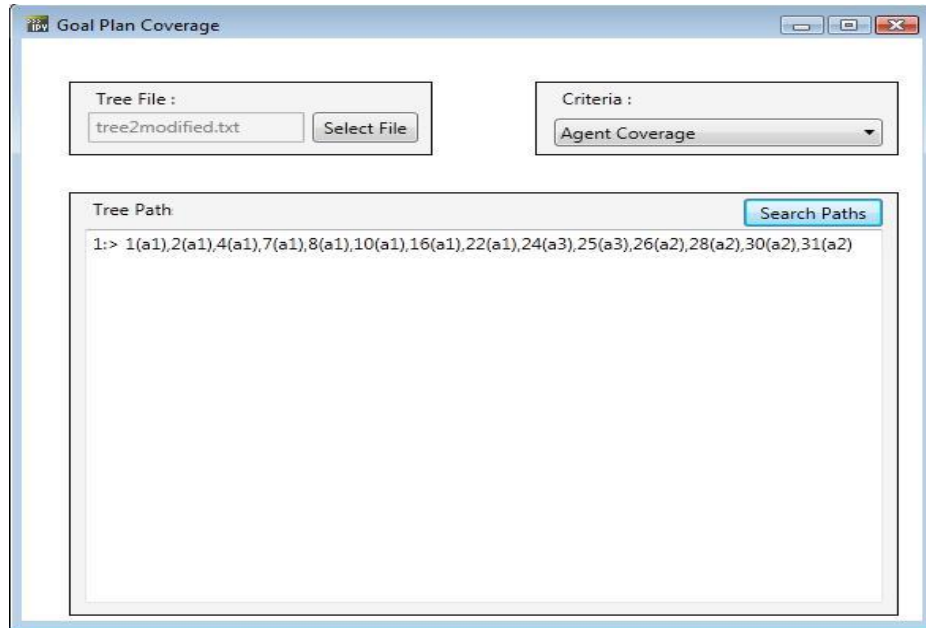


Figure- 6.18: GPG Test Paths Generator (Agent Coverage)

6.4 Test Case Execution

Test case execution process is semi-automatic. We have used a multi-currency banking MAS for evaluation of our testing framework. Three variables are used in our Multi-Currency Banking System i.e. account name, currency and amount. Random data is generated for each variable and assigned. Test case data will be generated by assigning variable values included in functions as shown below:

```

public static String nextName(){
    String ary[] = {"Shafiq","Peter","ali"};
    Random rand = new Random();
    return ary[rand.nextInt(ary.length)];
}
public static String nextCurr(){
    String ary[] = {"USD","AUD","YEN",};
    Random rand = new Random();
    return ary[rand.nextInt(ary.length)];
}
public static double nextAmount(){
    double ary[] = {40,15,100,500, 10000};
    Random rand = new Random();

```

```
return ary[rand.nextInt((ary.length)-1)];
```

After assigning variables values, there is a need to define number of test cases to execute. We have instrumented the code to generate a number of test cases to execute on MAS. Only part of the code added in JACK project is shown below.

```
int length = ary.length, seq_num , loopen2 = 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());

public class Accounts {
public static LinkedList acclist;
public static String nextName();
public static String nextCurr();
public static double nextAmount();
public static void callCommands(int[] s, Communicator comm, String name, String currency,
double amount) throws Exception {
for (int i = 0; i < s.length; i++){
switch (s[i]){
case 0:
acclist.insertNode(new
LinkedList(name, createAccount(comm, name, currency, nextAmount())));
break;
case 1:
creditAccount(comm, acclist.findAccNo(name), currency, nextAmount());
break;
case 2:
debitAccount(comm, acclist.findAccNo(name), currency, nextAmount());
break;
default:
break;
}
```

```
}  
}  
}
```

Details of logs generated after test cases execution is presented in Appendix-B.

CHAPTER 7: EVALUATION: RESULTS AND DISCUSSION

In previous chapter we have presented testing framework for MAS interaction testing and goal-plan based testing. In order to prove effectiveness of testing framework, we have presented a case study of MAS. We have seeded faults in case study implementation and testing framework has been applied.

Our objectives for evaluation are finding the faults in MAS with our testing framework, using test model and test coverage criteria. Generating test data for test cases and executing test cases on actual implementation. MAS designed by Prometheus methodology are implemented in JACK programming language. Fault detection is the main objective after executing all processes of our testing framework. Evaluation process consists of:

- a) Fault Injection in MAS implementation
- b) Test Model Generation from design artifacts
- c) Test case generation and execution (coverage criteria based)
- d) Fault identification by test cases

7.1 Evaluation: Case Study

To validate our testing framework, we have taken case study of Multi-Currency bank account system (Jack intelligent agents, AOSGRP) which maintains bank accounts in nominated currencies, and performs currency conversions transactions against the accounts in any currency. It consists of a *BankAccount agent*, a *CurrencyExchange agent* and a *Communicator agent* which acts as an interface. All steps defined in our testing framework are applied to this case study and faults are then injected for validation purpose. This case study first generates test model from Prometheus design artifacts and then coverage criteria is applied to find test paths automatically with the help of implemented tool. Faults described in Chapter 4 are detected and which coverage criterion led to their discovery is also discussed.

7.1.1 DESIGN ARTIFACTS

Multi Currency Banking multi-agent system has three agents, e.g., *BankAccount agent*, *CurrencyExchange agent*, and *Communicator agent*, which work together to create account, debit account, credit account, debit and credit account with same and different currency and

currency conversion. We have designed artifacts of banking MAS which will be used to generate our test model. Notations used in all design artifacts are standard notations used in Prometheus Design Tool (PDT) (Thangarajah, Padgham & Winikoff, 2005). We will discuss only those design artifacts which are involved as input to our testing framework to generate test models.

In Prometheus; system specification level consists of scenario and goal overview diagrams, architectural design phase consists of protocols and detail design process consists of process diagram i.e. agent and capability overview diagrams. Figure 7.1 presents scenario overview diagrams of our case study example.

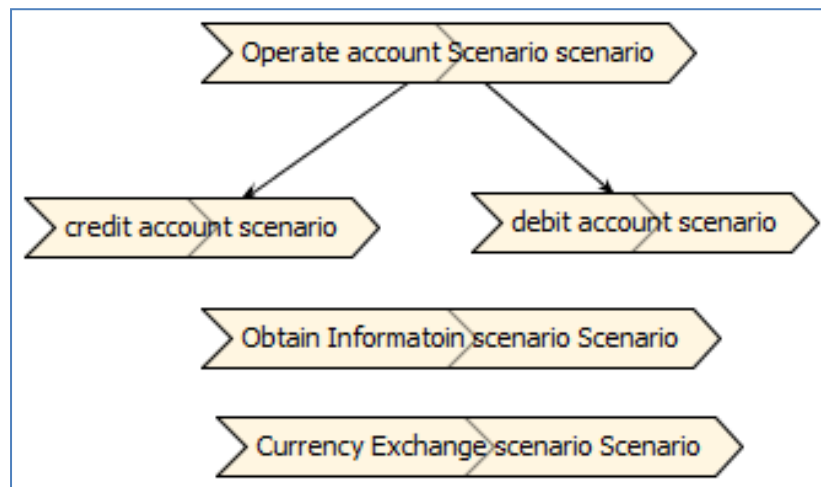


Figure-7.1: Scenario Diagram of Multi Currency Banking MAS

MAS have three main scenarios which consist of a sequence of goals, actions and percept to perform. As depicted in Figure 7.1 the *operate account scenario* has two sub scenarios to handle, e.g., *credit account scenario* and *debit account scenario*. Each scenario has its goal overview diagram as well and collectively all these goal diagrams participate in MAS functioning.

Goal overview diagram of multi currency MAS is presented in Figure 7.2. *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 *TwoStepExchange goal*. *TwoStepExchange goal* is triggered if two step currency conversions are required. In goal

overview diagram, top level goal showing relevant scenario for the MAS. All relevant goal and sub-goals will be listed on internal and lead nodes of goal overview diagram.

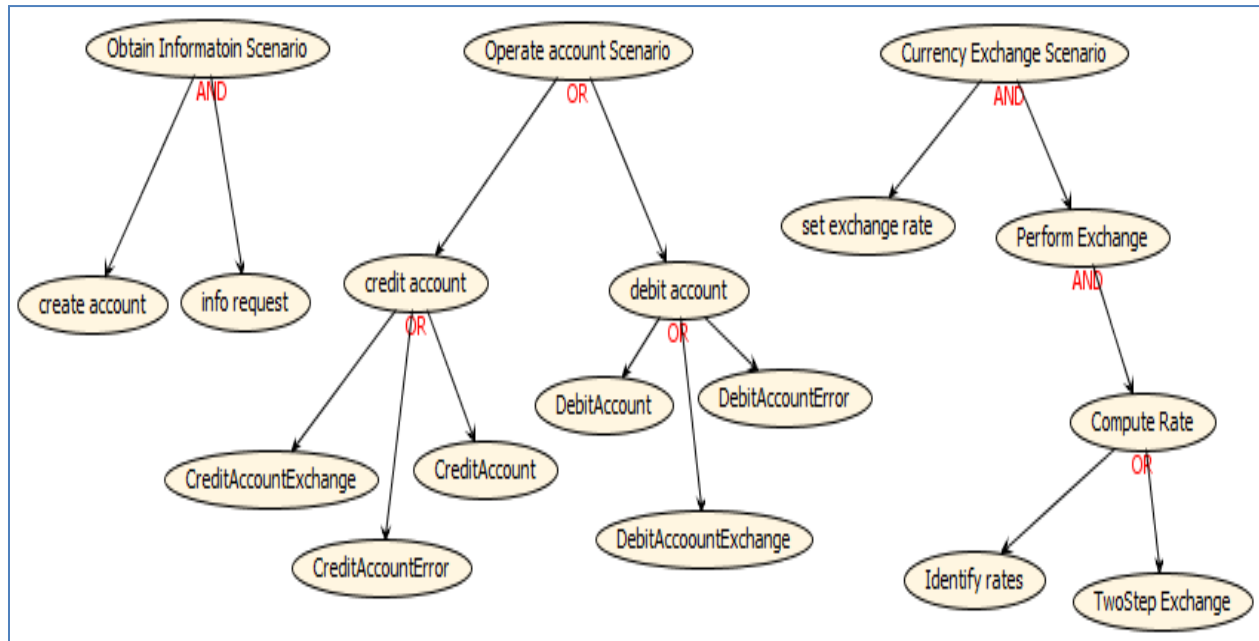


Figure-7.2: Goal Overview Diagram of MAS

We have designed the system overview diagram of account case study. Figure 7.3 shows system overview diagram of multi-agent system.

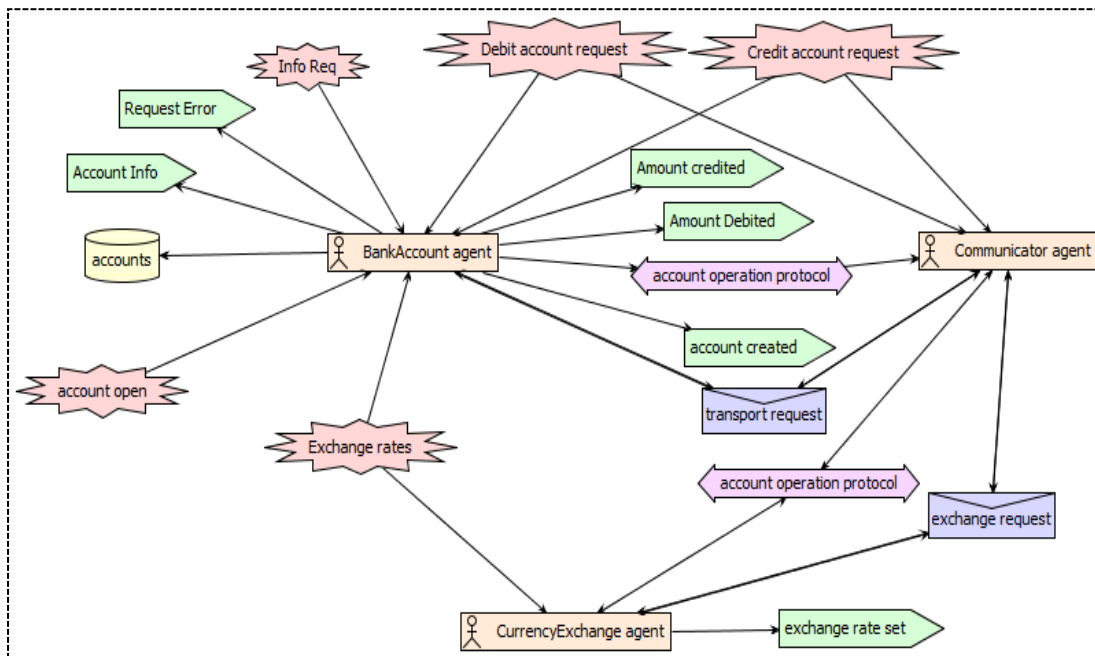


Figure-7.3: System Overview Diagram of Multi-agent system

In System overview diagram different agents have interaction with each other via account operation protocol. Each agent has actions, percepts and messages associated with it. Different interactions between agents and actors are occurring through account operation protocol as depicted in figure 7.4 and 7.5. Each protocol includes different interactions between agents and actors to perform specific tasks, such interactions are modeled in protocol diagram. Content of protocol diagram includes *alternatives*, *loops* and other deviations from a simple sequence are depicted in AUML using nested boxes. Figure 7.4 shows AUML description of account operation protocol diagram used in Prometheus Design Tool.

```

start account operation protocol
actor A user
agent B BankAccount agent
actor C account owner
agent D CurrencyExchange agent
agent E Communicator agent
box loop
    percept A B account open
    action B A account created
end loop
box loop
box alternative
    percept C B Debit account request
    next
    percept C B Credit account request
end alternative
box opt
    message B E TransportRequest
    message E D exchangeRequest
    percept A D exchange rates
    message D E exchangeRequest reply
end opt
box alternative
    action B C amount Debited
    action B C Account Info
    next
    action B C amount credited
    action B C Account Info
    next
    action B C Request Error
end alternative
end loop
finish

```

Figure-7.4: AUML Description of Account Operation Protocol

Figure-7.5 shows an *account operation protocol* diagram, which is further converted to protocol graph for interaction testing by protocol graph convertor process.

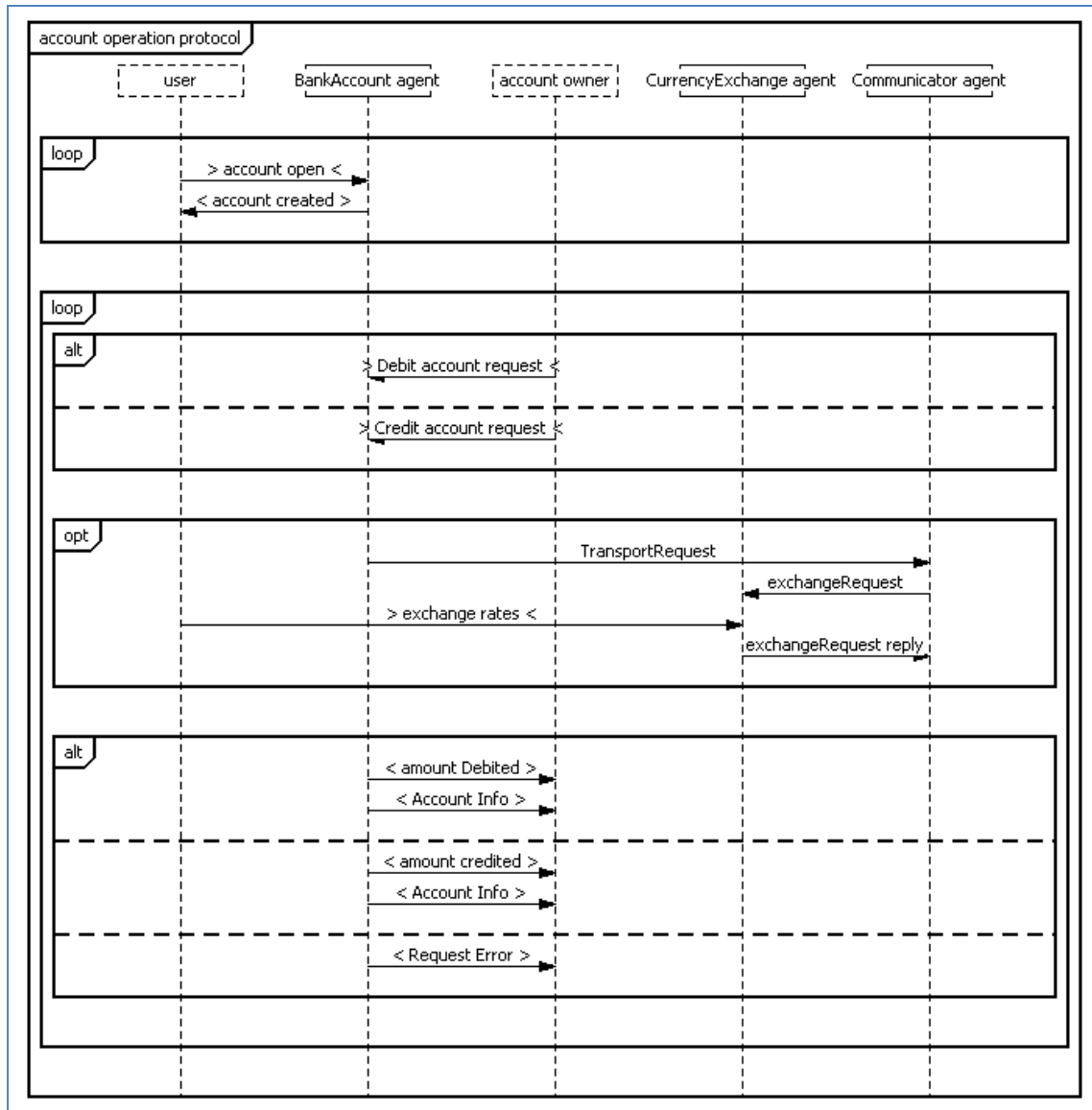


Figure-7.5: Account Operation Protocol Diagram

So far design artifacts related to system analysis and architectural design has been presented for our case study. Now we will discuss how information from these artifacts is used to model detailed design phases design artifacts of Prometheus methodology e.g. agent and capability overview diagrams. For each agent there is an agent overview diagram. The *BankAccount agent* overview diagram is shown in Figure 7.6; it has two capabilities, i.e., *CreditAccountCap* and *DebitAccountCap* and three plans, e.g., *CreateAccountP*, *AccountInfoP*, *AccountOperationP*

which have some goals to achieve. Certain percepts and messages are used as the triggering events for the plans and capabilities as shown in diagram. An arrow shows the flow of information from one entity to other.

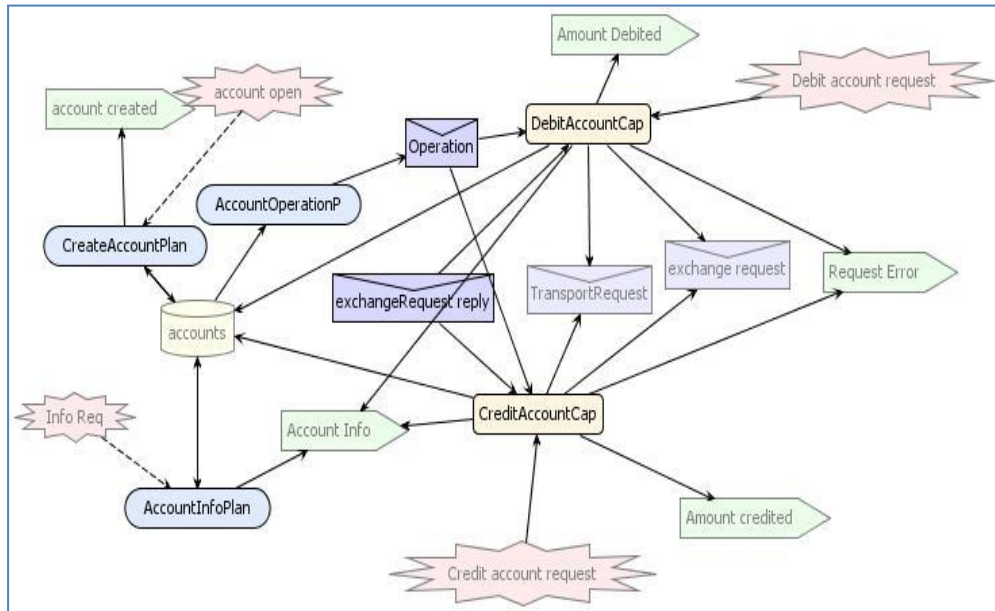


Figure-7.6: BankAccount Agent Overview Diagram of MAS

Each capability is further elaborated in capability overview diagram as depicted in Figure 7.7.

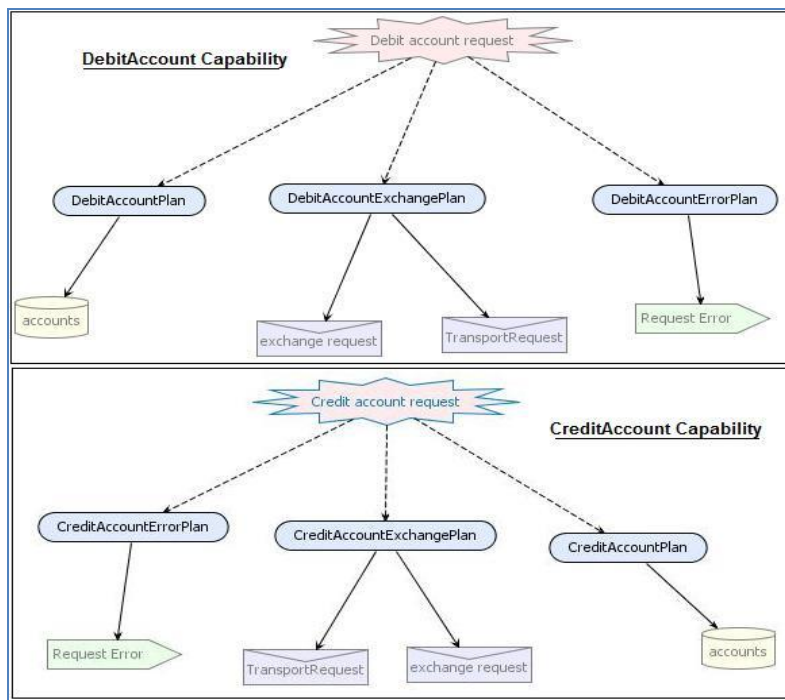


Figure-7.7: DebitAccountCap and CreditAccountCap Capability Overview Diagram of MAS

Each capability has three plans for alternative three goals as depicted in goal overview diagram. *Accounts* and *Rates* are two data stores that contain exchange rates and accounts updated details. On triggering *debit account capability*, there are three plans that can be used, e.g., *DebitAccountP*, *DebitAccountErrorP*, *DebitAccountExchangeP*. In case of *exchange request*, *Communicator agent* get the message of *Transport Request*; which generates *Exchange Request* message for *Currency Exchange agent*. *Currency Exchange agent* and *communicator agent* along with *ComputeRate* capability diagram is shown in Figure 7.8. *Currency Exchange agent* have two plans i.e. *PerformExchange*, *SetExchnageRatePlan* and one capability *ComputeRate*. *ComputeRate* capability needs to execute two plans and replies with a message containing exchange amount.

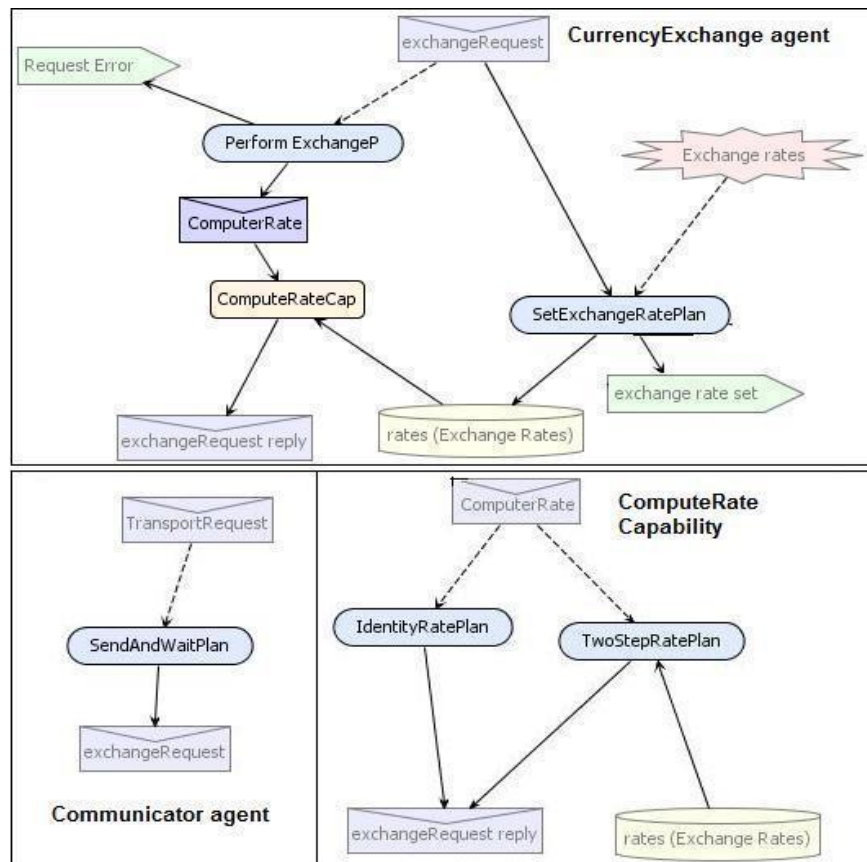


Figure-7.8: CurrencyExchange Agent, Communicator Agent and ComputeRate Capability Overview Diagrams.

While considering the protocol diagram loop can be on creation of accounts or on debiting or crediting the account more than one time.

7.2 Faults Seeded in MAS implementation

We have defined fault model for interaction testing and goal, sub-goal and plan based testing in Chapter 4. We have injected interaction faults in implementation of our case study for validation of our testing framework.

7.2.1 FAULT TYPE SEEDED: INTERACTION FAULTS

We have defined three types of faults in our interaction fault model presented in Section 4.1.1. Protocol diagram is used for interaction in MAS. Protocol diagram consists of three type of interaction i.e. action, percept and message. Dependency faults can occur if percept interaction is missed. Operational faults occur if any action is not executed. Synchronization faults occur if any action/percept is missed then wrong message content can be conveyed to other agent.

Table-5: Injected Faults for Interaction Testing

Fault ID	Fault Type	Injected Faults details
F-1	Operational Faults	Account open action not performed #uses interface AccountServices creator; #modifies data Account accounts; not executed
F-2		Credit account action not executed
F-3	Dependency Faults	Credit account percept not received #posted as credit(int accountNumber, String currency, double amount) not executed
F-4		Exchange Rate percept not received #posted as conversion(String currency1,String currency2,double rate) not executed due to which exchange request reply message will not occur
F-5	Synchronization faults	TransportRequest message will not execute due to F-1 and F-3
F-6		#complex query conversion(String currency1,String currency2) { logical double rate; return conversion(currency1,currency2,rate); } query conversion not return any value exchange request reply message will not occur Wrong account info shown

We have discussed design artifacts of our case study. Table 5 shows injected faults in multi-currency banking case study implementation developed in JACK.

7.2.2 FAULT TYPE SEEDED: GOAL, SUB-GOAL AND PLAN FAULTS

Goals and Plans are the key to MAS execution and any fault relevant to goal and plans makes it difficult for MAS to execute correctly. We have defined a fault model for goal, sub-goal and plans in section 4.1.2. Fault types are; inaccurate goal achievement, plan failure, internal agent fault, missing functionality, scenario fault and deliberate faults. For each fault type at least one fault is injected in the MAS implementation. Table-6 provides the details of injected faults for the sake of validation in multi-currency banking MAS. Fault ID is different from interaction faults. We have deliberately instrumented the JACK code to inject faults 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 a particular scenario or a capability not to execute etc.

Table-6: Injected Faults in Multi-Currency Banking MAS

Fault ID	Fault Type	Injected Faults details
F-11	Plan failure	CreditAccountPlan not covered- Making its context false
F-22	Inaccurate goal achievement	Debit Account goal not triggered – event for debit account not posted “#posted as” not working
F-33	Scenario fault, Internal agent fault	#posts event TransportRequest tev; not posted - Agent functionality missed, currency exchange scenario missed
F-44	Plan failure, Missing functionality	Compute rate event handler made false - Capability missed
F-55	Deliberate faults, Internal agent fault	#reads data Account “accounts” not allowed – database reading/writing not allowed - Deliberate faults
F-66	Missing functionality, Deliberate Fault	Obtain Information event not triggered after Node 6 in test model etc - loop not executed

After faults injection in MAS, we use Prometheus design artifacts described earlier in this chapter to create the test model. The test model is used to create test paths and test cases in order to identify injected faults.

7.3 Test Model Generation from design artifacts

We have defined design artifacts in section 7.1.1. For interaction testing we use protocol diagram and convert it into our test model, i.e., protocol graph by PD meta-model to PG meta-model ATL transformation rules described in our testing implementation chapter 6. Figure 7.9 is the protocol graph for Account Operation protocol diagram.

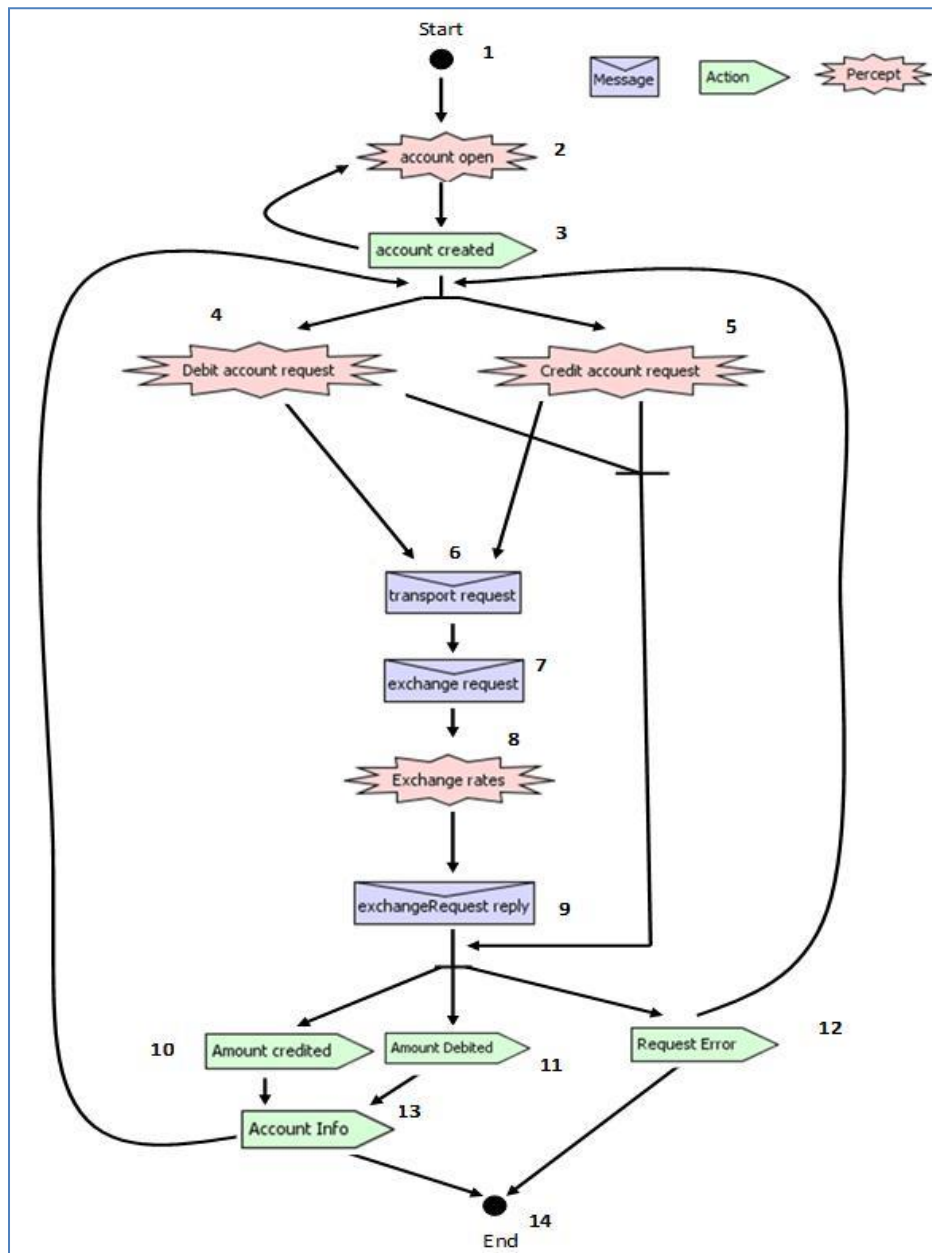


Figure-7.9: Protocol Graph (Test Model) for Account Operation Protocol Diagram

We use the details contained in design artifacts for our case study and generate test model for goals, sub-goals and plans based fault identification. Goal-Plan Graph (GPG) is our test model. Algorithm-III in chapter 6 provides step by step guidance to generate a test model. Figure 7.10 is the result of applying Algorithm-III on design artifacts as described in testing framework.

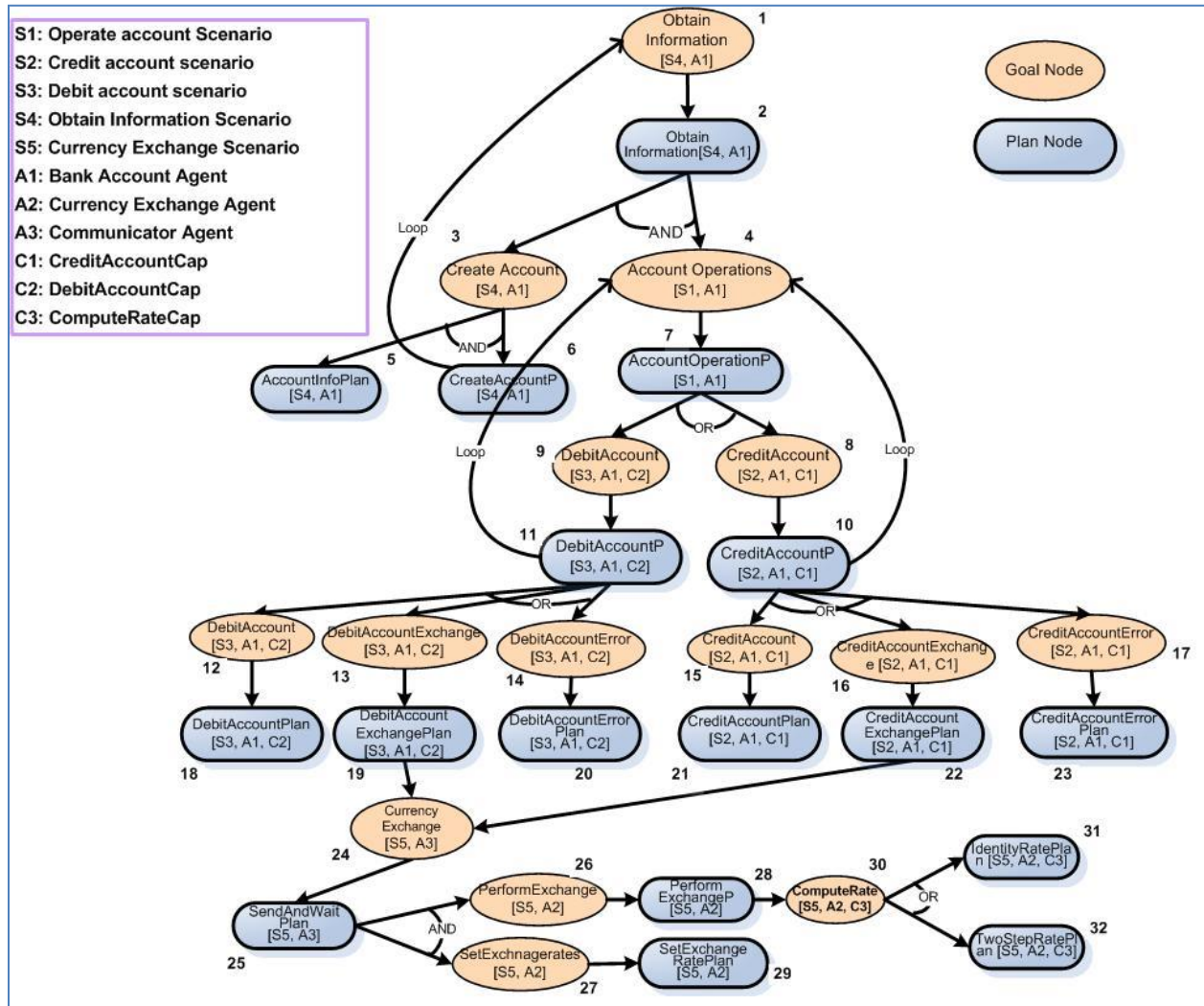


Figure 7.10: Goal-Plan Graph (Test Model) for Multi Currency MAS system

GPG is constructed by using scenario, goal, agent and capability overview diagrams. We also consider loops information from protocol diagram to add loops to our generated test model. Goal-Plan Graph contains all plans and goals involved in MAS functioning. The ‘AND’ and ‘OR’ constraints between goals and plans are also considered and presented.

7.4 Test Cases Generation And Execution (Coverage Criteria Based)

Test cases are aimed to run on MAS implementation and their results will be compared to those of identified test paths generated from test models. We have protocol graph and Goal-Plan Graph (GPG) as the test model for interaction testing and goal plan testing. Defined coverage criteria presented in Section 5.3.2 and 5.4.2 are applied on test models to generate test paths.

7.4.1 INTERACTION TEST PATHS GENERATION

We have developed a tool to illustrate our testing framework presented in implementation chapter 6. Protocol diagram converted to protocol graph on which different coverage criteria are applied to generate paths with respect to them as defined in Section 5.3.2. Test paths generation tool process is presented in Section 6.1.3.

Table-7 shows test path against each coverage criteria we have defined and applied on our test model i.e. protocol graph. Screen shots of protocol graph test paths generation tool are shown in section 6.1.3.

Table-7: Test Paths for Account Operation Protocol Diagram

S. #	Coverage Criteria	Test Paths
1	Message Coverage	<ul style="list-style-type: none"> ▪ 1→2→3→4→6(message)→7(message)→8→9(message)→ 11→13→14
2	Action Coverage	<ul style="list-style-type: none"> ▪ 1→2→3(action)→5→6→7→8→9→10(action)→13(action)→14 ▪ 1→2→3→5→6→7→8→9→12(action)→14 ▪ 1→2→3(action)→4→6→7→8→9→11(action)→13(action)→14
3	Percept Coverage	<ul style="list-style-type: none"> ▪ 1→2(percept)→3→5(percept)→6→7→8(percept)→9→10→13→14 ▪ 1→2(percept)→3→4(percept)→6→7→8(percept)→9→11→13→14
4	Message Action Coverage	<ul style="list-style-type: none"> ▪ 1→2→3→5→6→7→8→9 (message)→10(action)→13→14 ▪ 1→2→3→5→6→7→8→9 (message)→12(action)→14 ▪ 1→2→3→4→6→7→8→9(message)→11(action)→13→14
5	Action Percept Coverage	<ul style="list-style-type: none"> ▪ 1→2→3(action)→5(percept)→6→7→8→9→10→13→14 ▪ 1→2→3(action)→4(percept)→6→7→8→9→11→13→14 ▪ 1→2→3→5→6→7→8→9→12(action)→5(Percept)→6→7→8→9→12→14 ▪ 1→2→3(action)→4(percept)→6→7→8→9→11→13→4→6→7→8→9→11

		→13→14
6	Percept Message Coverage	<ul style="list-style-type: none"> ▪ 1→2→3→5(percept)→6(message)→7→8(percept)→9(message)→10(action)→13→14 ▪ 1→2→3→4(percept)→6(message)→7→8(percept)→9(message)→11(action)→13→14
7	Pairwise Message Coverage	<ul style="list-style-type: none"> ▪ 1→2→3→5→6(message)→7(message)→8→9 →10→13→14
8	All round trip paths	<ul style="list-style-type: none"> ▪ 1→2→3→2→3→5→6→7→8→9 →10→13→5→12→14 ▪ 1→2→3→5→11→13→5→12 →5→10→13→14
9	All Paths coverage	<ul style="list-style-type: none"> ▪ Infinite # of Paths

7.4.2 GOALS AND PLANS TEST PATHS GENERATION

For goals, sub-goals and plans coverage, we have generated Goal-Plan Graph test model of our case study. Coverage criteria defined in Section 5.3.2 are applied on the Goal-Plan Graph and automatic test paths are generated. GPG test paths generation process defined in Section 5.3.3 has been followed. The GUI of GPG test path generation tool is shown in 6.3.1, which implements the process presented in Section 5.3.3. For loop coverage we consider two iterations, in the case of loop executing more than once. GPG structure or input of GPG used for automatic test paths generation is presented in Table-8. For understanding purpose only one node structure has been presented here. Table-9 shows paths generated from GPG by applying our defined coverage criteria.

Table-8: Structures of the Goal-Plan Graphs used as Input to Tool

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				

Table-9: Test Paths for Goal-Plan Graph

S. No	Coverage Criteria	Test Paths
1	All goals Coverage	<p>1(goal)→2→3(goal)→6</p> <p>1(goal)→2→3(goal)→5</p> <p>1(goal)→2→4(goal)→7→8(goal)→10→15(goal)→21</p> <p>1(goal)→2→4(goal)→7→8(goal)→10→17(goal)→23</p> <p>1(goal)→2→4(goal)→7→8(goal)→10→16(goal)→22→24(goal)→25→27(goal)→29</p> <p>1(goal)→2→4(goal)→7→8(goal)→10→16(goal)→22→24(goal)→25→26(goal)→28→30(goal)→31</p> <p>1(goal)→2→4(goal)→7→9(goal)→11→12(goal)→18</p> <p>1(goal)→2→4(goal)→7→9(goal)→11→14(goal)→20</p> <p>1(goal)→2→4(goal)→7→9(goal)→11→13(goal)→19→24(goal)→25→27(goal)→29</p>
2	Scenario Coverage	<p>1(S4)→2(S4)→4(S1)→7(S1)→8(S2)→10(S2)→17(S2)→ 23(S2)</p> <p>1(S4)→2(S4)→4(S1)→7(S1)→9(S3)→11(S3)→13(S3)→19(S3)→24(S5)→25(S5)→27(S5)→29(S5)</p>
3	Agent Coverage	<p>1(A1)→2(A1)→4(A1)→7(A1)→9(A1)→11(A1)→13(A1)→19(A1)→24(A3)→25(A3)→27(A2)→29(A2)</p>
4	Capability Coverage	<p>1→2→4→7→8(C1)→10(C1)→15(C1)→ 21(C1)</p> <p>1→2→4→7→9(C2)→11(C2)→13(C2)→ 19(C2) →24→25→26→28→30(C3)→32(C3)</p>
5	Plan Coverage	<p>1→2(Plan)→3→5(Plan)</p> <p>1→2(Plan)→3→6(Plan)</p> <p>1→2(Plan)→4→7(Plan)→8→10(Plan)→17→23(Plan)</p> <p>1→2(Plan)→4→7(Plan)→8→10(Plan)→15→21(Plan)</p> <p>1→2(Plan)→4→7(Plan)→8→10(Plan)→16→22(Plan)→24→25(Plan)→27→29(Plan)</p> <p>1→2(Plan)→4→7(Plan)→8→10(Plan)→16→22(Plan)→24→25(Plan)→26→28(Plan)→30→31(Plan)</p> <p>1→2(Plan)→4→7(Plan)→8→10(Plan)→16→22(Plan)→24→25(Plan)→26→28(Plan)→30→32(Plan)</p> <p>1→2(Plan)→4→7(Plan)→9→11(Plan)→12→18(Plan)</p> <p>1→2(Plan)→4→7(Plan)→9→11(Plan)→14→20(Plan)</p>

		1→2(Plan)→4→7(Plan)→9→11(Plan)→13→19(Plan)→24→25(Plan)→27→29(Plan)
6	Goal Plan Coverage	<p>1(goal)→2(Plan)→3(goal)→6(Plan)</p> <p>1(goal)→2(Plan)→3(goal)→5(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→17(goal)→23(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→15(goal)→21(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→16(goal)→22(Plan)→24(goal)→25(Plan)→27(goal)→29(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→16(goal)→22(Plan)→24(goal)→25(Plan)→26(goal)→28→30(goal)→31(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7→8(goal)→10→16(goal)→22→24(goal)→25→26(goal)→28(Plan)→30(goal)→32(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→14→20(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→12→18(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→13(goal)→19(Plan)→24(goal)→25(Plan)→27(goal)→29(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→13(goal)→19(Plan)→24(goal)→25(Plan)→26(goal)→28(Plan)→30(goal)→31(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→13(goal)→19(Plan)→24(goal)→25(Plan)→26(goal)→28(Plan)→30(goal)→32(Plan)</p>
7	Loop Coverage	<p>1(goal)→2(Plan)→3(goal)→6(Plan)</p> <p>1(goal)→2(Plan)→3(goal)→5(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→15(goal)→21(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→14→20(Plan)</p> <p>1(goal)→2(Plan)→3(goal)→6(Plan)→ 1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→14→20(Plan)</p> <p>1(goal)→2(Plan)→3(goal)→6(Plan)→ 1(goal)→2(Plan)→3(goal)→6(Plan)→ 1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→14→20(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→4(goal)→7(Plan)→8(goal)→10(Plan)→17(goal)</p> <p style="text-align: right;">} Loop 0 time</p> <p style="text-align: right;">} Loop 1 time</p> <p style="text-align: right;">} Loop 2 times</p>

		<p>→23(Plan)</p> <p>1(goal)→2(Plan)→4(goal)→7(Plan)→9(goal)→11(Plan)→4(goal)→7(Plan)</p> <p>→9(goal)→11(Plan)→ 4(goal)→7(Plan)→9(goal)→11(Plan)→ 12(goal)</p> <p>→18(Plan)</p>
--	--	--

7.4.3 TEST CASE GENERATION AND EXECUTION

Once we have created test models by using Prometheus design artifacts and test paths from test model by applying defined coverage criteria, our next step is to generate and execute test cases. For interaction testing test cases are designed to force program to traverse those execution paths that reflect actions, message and action events in MAS. Event is generated by executing test case that causes message and actions interactions. As discussed in Section 5.2.4 test cases are generated by constructing NDT of each node in test paths. For example a test path against ‘action percept’ coverage criterion: 1→2→3(action)→5(percept)→6→7→8→9 →10→13→14 has its relevant NDT shown in Table 10. Node 1 and 14 are the start and end node. Test cases are constructed by using NDT entries for each path. Our test case should target to generate test path that leads the execution of MAS on action and then percept. But if there is some error or fault in MAS implementation then it will not traverse interactions in the desired order. We will generate test case for each coverage criteria so that all possible interaction have been tested. Test cases are generated by assigning variable values and forming the variable’s combinations for test cases execution.

Table-10: Node Description Table Interaction Test Paths Nodes

Node No.	Node Type	Associated Variables/functions
2	Percept (Account Open)	String = Account Title String = Currency Integer = Amount
3	Action (Account Created)	Triggering event = account open Function = Create account
5	Percept (Credit Account request)	String = Account Title String = Currency Integer = Amount
6	Message (Transport Request)	Triggering event = Percept Function = Transport Request
7	Message (Exchange Request)	Function = Exchange Request
8	Percept (Exchange Rate)	String = Currency Float = currency rate

9	Message (Exchange Request Reply)	String = Currency Long = Amount converted
10	Action (Amount Credited)	Function = Credit account
13	Action (Amount Info)	Triggering event = Action Integer = Amount String = Account Title

For goal and plan coverage, test cases are executed to traverse goals and plans in MAS. Plans are used to satisfy certain goal and sub-goals.

Table-11: Node Description Table for Goal-Plan 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

Our test cases traverse possible plans satisfying goal or sub-goals. As discussed in Section 5.3.4 test cases are generated by constructing NDT of each node in test paths. 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 11. Only part of NDT has been shown here.

Test data generation for test cases is generated semi-automatically. Test cases are generated by assigning variable values and forming the variable’s combinations for test cases execution. In our case study we have three main calling functions from main Java file in JACK code i.e. *createAccount*, *creditAccount* and *debitAccount*. There are 8 plans for accounts operations handled by *BankAccount 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. Variables values are passed in functions to generate events. JACK code has been instrumented for automatically assigning

variable values for test case generation and test cases are executed by just providing the total number test cases to execute. Patterns of variables for test case executions are then automatically formed. 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. For each coverage criterion path, different test cases can be generated with different value combinations. Instrumented code will generate output showing details of executed or traversed plans in test case execution.

We have provided list of values for all three test parameters, i.e, name, currency and amount. These values are hard coded in MAS JACK implementation and randomly these values are called along with function name automatically once we run the implementation. Sample execution code instrumentation and sample execution log after test cases execution is presented in implementation chapter section 6.4.

Based on our case study test paths and NDT following is the structure of test case for our MAS under test is as follows:

Operation. type <name.Value, currency.value, amount.value>

For example:

createAccount(name, currency, amount)

creditAccount (name, currency, amount)

debitAccount (name, currency, amount)

We have instrumented the MAS implementation to generate test case execution log for interaction testing to show which action, percept and message have been executed on a test case as shown below. Sample log is presented in appendix-B.

createAccount (shafiq, AUD, 100)| credit (shafiq, AUD, 40)

Result:

User to BankAccount agent --> Percept (account open)

BankAccount agent to User --> Action (account created)

Created account 1 for Shafiq in AUD with opening balance 100.0

accountowner to BankAccount agent --> Percept (Credit account request)

BankAccount agent to account owner --> Action (amount credited)

BankAccount agent to accountowner --> action (Account info)

Credited account 1 with AUD40.0. Balance: 140.0

Executed paths are automatically extracted from test case execution log which can be easily compared with test paths identified on test model of MAS. Execution path against the above execution log is shown below.

1→2(Percept) →3(Action)→5(Percept)→ 10(action)→13(action)→14

For goals, Sub-goals and plans coverage and testing, implementation has been modified to execute and log plans for goals and sub-goal, also logging of events is done while executing the test cases as shown below. Appendix-C contains a log generated for goal, sub-goals and plan coverage while executing test cases.

createAccount (peter,YEN,40)

Result:

obtain information(goal)

obtain informationP(Plan)

Create Account(goal)

Test CreateAccountPlan

Create AccountP(Plan)

Account InfoPlan(Plan)

We have manually calculated expected output of the MAS. For example action (account created) interaction is expected after percept (account open) in case of interaction testing. For system testing, an account debit request is made with 50 dollar then expected output shown that 50 dollars debited from given account etc. For each operation request we calculated the expected output and then compared the expected output with test case results and declared the test case as a pass or a fail. An output is produced after execution of correct plan triggered for a goal. Actions, Percepts, Messages, Goals and plans are shown as nodes in our test models.

Expected output and test results evaluation process is manual. Against a certain input, MAS has to produce some output after executing its plans. A specific path is followed for a given input.

Based on the event and plan execution of the system we calculate expected output for each test case. The effort to evaluate test case results may be estimated as below:

Let t_1 = average time required to produce expected result for a test case

t_2 = average time required to compare output of a test case with expected output

n = number of test cases

Then time required for oracle generation and result evaluation will be $t = n (t_1 + t_2)$.

7.5 Fault Identification By Test Cases

In this section we discuss faults detected in relation to the fault model. Our testing framework ensures to identify faults which occur due to non coverage or unordered execution of goals and plans as defined in design artifacts. We have our test models to identify and capture faults in MAS. We have defined our fault model which covers possible faults types that could occur in MAS. 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 have been identified by applying different coverage criteria and test cases that lie within coverage criteria. Coverage criteria ensure certain types of faults detection and identification with a system (Tian, 2001). Table-5 & 6 summarizes injected faults for interaction testing and goals-plans testing of Multi-Currency Banking MAS.

We have our defined coverage criteria for interaction testing and goal-plan testing. Coverage criteria have been discovered to be very effective in identifying injected faults in MAS. 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. For example after injecting Credit account Plan Fault in MAS.

A test case is considered as a pass if actual result matches with the expected result. Actual results are according to expected results only if the action, percept and message interaction are executed according to sequence as defined in interaction protocol. A test case is failed if it does not traverse the defined path against certain input, which may be due to an injected fault.

Interaction testing results are compared with the expected results and a test case is declared as either pass or fail. A failed test case is further analyzed to know the exact node/interaction type not covered that cause wrong output to trigger.

Test path = 1→2(Percept)→3(Action)→5(Percept)→10(action)→ 13(action) →14

Injected Fault = F2 (Credit account action not executed)

Test cases: *createAccount(John,USD,100)/creditAccount(John,USD,40)*

Actual output = 1→2(Percept)→3(Action)→5(Percept)→ {Not triggered}.....

Nodes were not traversed in case there was some fault or deviation in MAS implementation from design models.

For goal-plan testing, we executed test case:

createAccount (ali,GBD,40) / credit(ali,GBD,500)

and expected MAS output against the test cases may be:

Created account for ali in GBD with opening balance 40:

Path = 1(goal)→2→3(goal)→6

Credited account title ali with GBD 500.0. Balance: 540.0:

Path = 1(goal)→2→3(goal)→5

Credited account title ali with GBD 500.0. Balance: 540:

Path = 1(goal)→2→4(goal)→7→8(goal)→10→15(goal)→21

Actual MAS output after executing this test case was:

Created account for ali in GBD with opening balance 40:

Path = 1(goal)→2→3(goal)→6

Credited account title ali with GBD 500.0. Balance: 540.0:

Path = 1(goal)→2→3(goal)→5

Could not debited

1(goal)→2→4(goal)→7→8(goal)→10.....?// Plan is not executed, comparison showed that injected fault was identified successfully.

Table-12 shows minimum number of test cases that are required for MAS interaction and goal, sub-goal and plan coverage.

Table-12: Test Cases for MAS testing

TC ID	Test Case
TC-1	createAccount(John,USD,100)
TC-2	createAccount(Ali, GBD,200) debit ((Ali, GBD,100))
TC-3	createAccount(Oliver,YEN,150)
TC-4	createAccount(Jack, YEN,50)
TC-5	createAccount(Shafiq,AUD,250)
TC-6	creditAccount (John,USD,50)
TC-7	creditAccount (Ali,YEN,50)
TC-8	creditAccount (John,AUD,200)
TC-9	creditAccount (Ali,AUD,200)
TC-10	debitAccount (John,USD,40)
TC-11	debitAccount (Shafiq,AUD,400)
TC-12	debitAccount (John,AUD,50)
TC-13	debitAccount (Ali,AUD,50)
TC-14	createAccount (Shafiq, YEN,15) credit (Shafiq, YEN,100)
TC-15	createAccount (ali,GBD,40) credit(ali,GBD,500) debit (ali,GBD,15)

We built a test model and applied coverage criteria to get expected execution of interactions and goals and plans in MAS. Injected faults were successfully identified by applying coverage criteria. For each coverage criteria we need certain test cases which ensure its coverage. These coverage criteria proved very useful identification of different types of faults defined and seeded in MAS. Table-13 summarizes interaction fault types identified by different coverage criteria. Operational faults occurs due to non-execution of action event; action coverage criterion and action-percept coverage criterion ensures that all action nodes must be covered so to avoid operational faults. Percepts are used to input environment information into MAS. Percept and percept-message coverage criterion ensures coverage of all percept nodes/edges so to avoid dependency faults. Synchronization faults are avoided by executing message nodes, message-message edges, action-percept edges and percept-message edges covered by coverage criteria presented in table 13.

Table-13: Interaction Fault Types Vs Coverage Criteria

S. No	Interaction Fault Type	Coverage Criteria (Interaction Testing)
1	Operational Faults	Action Coverage, Action Percept Coverage
2	Dependency Faults	Percept Coverage, Percept Message Coverage
3	Synchronization faults	Message Coverage, Action Percept Coverage, Percept Message Coverage, Pair wise Message Coverage

Table-14 summarizes goal-plan fault types identified by different coverage criteria.

Table-14: Goal-Plan Fault Types Vs Coverage Criteria

S. No	Goal-Plan Fault Type	Coverage Criteria (Goal-Plan)
1	Inaccurate goal achievement	All goals Coverage
2	Plan Failure	Plan Coverage
3	Internal Agent fault	Agent and capability Coverage
4	Missing functionality	Goal Plan Coverage, Loop Coverage
5	Scenario Fault	Scenario Coverage
6	Deliberate Fault	Goal Plan Coverage, Loop Coverage

We have executed test cases on the instrumented JACK implementation of MAS after seeding interaction faults. Our testing framework detected seeded faults by applying coverage criteria. Table-15 summarizes faults detected, which were injected in MAS shown in Table-5, by coverage criteria and minimum required test cases to cover test criterion. It shows the usefulness of coverage criteria in identifying seeded faults. Different coverage criteria revealed faults of different nature in MAS as depicted in Table-15. At least 8 test cases are required to cover nodes and edges coverage in different coverage criteria. All round trip paths coverage criterion is the strong among other criteria and it also covers all edges, that loops back on the same node, which have been missed by other criteria. For any coverage criterion, there is a test case to create the account before starting debit/credit operations test cases' execution.

Table-15: Detected Interaction Faults by Coverage Criteria and Minimum Test Cases

S. No	Coverage Criteria	Test cases	Faults Detected (Interaction)
1	Message Coverage	2 Test cases	F-5, F-6
2	Percept Coverage	4 Test Cases	F-3, F-4
3	Action Coverage	4 Test Cases	F-1, F-2
4	Action Percept Coverage	6 Test Cases	F-2, F-3
5	Percept Message Coverage	6 Test Cases	F-3, F-5
6	Message Action Coverage	5 Test Cases	F-2, F-6
7	Pair wise Message Coverage	3 Test Cases	F-6
8	All Round Trip Paths	8 Test Cases	F-1,F-2, F-3, F-4, F-5, F-6

Figure 7.11 presents graphics for minimum number of test cases required for a coverage criterion to identify number of interaction faults in MAS.

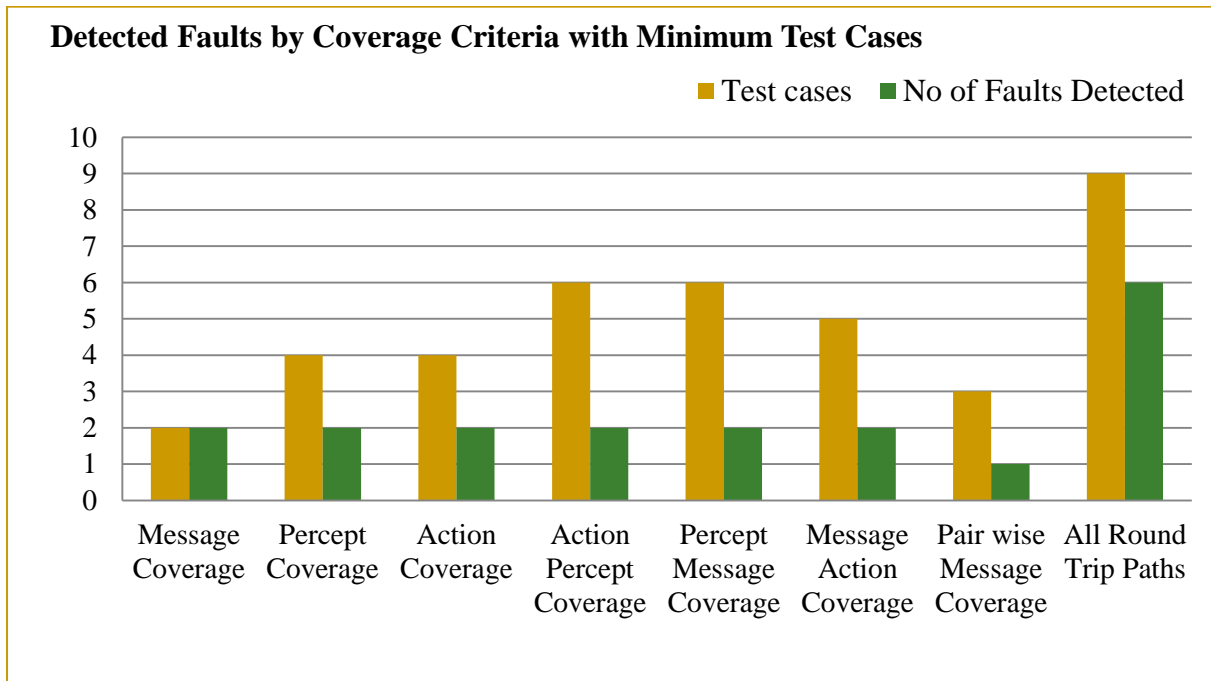


Figure 7.12: Chart with Test Cases and Coverage Criteria Detecting Types of Faults

Table-16 shows goal-plan related detected faults by applying coverage criteria which are injected in MAS shown in Table-6 and minimum number of test cases required to cover a specific test criterion. These minimum numbers of test cases were chosen after multiple executions and observing their result with respect to faults identified. It shows effectiveness coverage criteria in identifying injected faults. Different coverage criteria reveal different faults in MAS. There are certain overlaps in test cases. Test cases were applied on the instrumented code and it was found that by applying our coverage criteria injected faults were successfully identified.

Table-16: Detected Faults by Coverage Criteria and Minimum Test Cases Required

S. No	Coverage Criteria	Test cases	Faults Detected (Goal-Plan)
1	All goals Coverage	5 Test cases	F-22, F-66
2	Scenario Coverage	6 Test Cases	F-33, F-66
3	Agent and capability Coverage	6 Test Cases	F-44, F-55
4	Plan Coverage	8 Test Cases	F-11, F-44
5	Goal Plan Coverage	13 Test Cases	F-11, F-22, F-33, F-44, F-55
6	Loop Coverage	8 Test Cases	F-22, F-66

In Table-16 goal-plan coverage identifies five faults by executing 13 test cases but F-66 is not identified by goal-plan coverage criterion. Because fault 66 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.

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.

Figure 7.12 shows graphical representation of number of test cases executed for each coverage criterion and types of faults identified. The more faults types are detected, the more test cases are required.

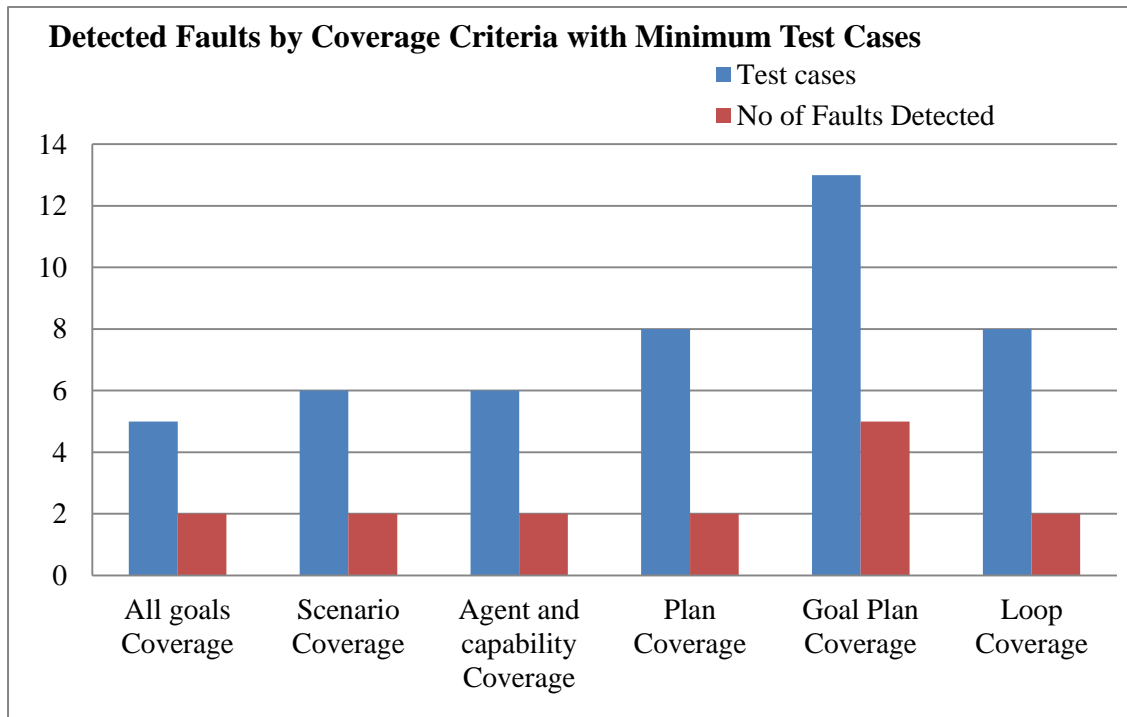


Fig. 7.13: Chart of Test Cases and Coverage Criteria Detecting Goal-Plan Faults

Currently we are testing all possible calls in a single test case; therefore the numbers of test cases are minimum. Our test case can be broken down into smaller test cases in which only one function call can be made with one test case. In that case we may need three times more test cases to uncover all defined faults. Our technique successfully finds the faults that reside on the MAS implementation through test cases execution.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

In this research, we have proposed a novel approach to test multi-agent systems based on design artifacts following 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 of Prometheus methodology, therefore no functionality is missed or remain uncovered. Testing a multi-agent system is a challenging task due to dynamic behavior of agents. We have focused on interaction testing and system of MAS.

Integration of MAS components involves interactions between the agents as well as agent and environment. Interaction protocol diagram contains all sorts of interactions between agents and between agent-actor actor like message, action and percepts. We have presented a fault model for MAS interactions and evaluated the proposed testing technique by experiments. We have proposed a testing framework which Prometheus design model is used to generate test model, i.e., protocol graph. The previously proposed protocol graph has been extended to include action and percepts along with messages. Messages are passed between agents and percepts/actions are used as the interaction mechanism between agents and actors. We have identified different coverage criteria which includes nodes and arcs of the protocol graph. These coverage criteria are used to generate test paths. Test cases are generated from coverage criteria's test paths. Test cases have been executed on MAS implementation in agent development environment. We have validated our technique on a MAS case study. We have injected faults in MAS and executed our test cases. Different coverage criteria have different test cases, from which, different faults are identified that also validate the usefulness of coverage criteria.

Another important aspect of multi-agent systems is how goals and plan are covered in MAS execution. Relationship between goals and plans is the key to test them. We have defined a fault model for testing of MAS with respect to goals, plans and sub goals. For system level testing, a testing framework is defined which uses scenario overview, goal overview, protocol diagram and process diagrams to generate test model for the actual system. 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 automatically generate and 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 specially to find goal and plan related faults are very effective in building trust in MAS. The defined coverage criteria make this very easy to find the root-cause of any identified fault.

Future Directions:

Although we have applied this approach on design artifacts Prometheus methodology along with its JACK implementation, but this approach can be applied to other Belief Desire Intension like agent system with slight modifications. In future work, test result evaluation process can also be automated. System models checking according to the requirement specification may also be performed in future before creating test models.

Test oracle is a process by which testers can produce expected output to decide whether the output of the program under testing is correct or not. In our proposed system, test oracle is a manual process to validate the system. A possible future direction is to use metamorphic testing approach (Chen 2015) for oracle, which is based on the simple intuition that although we may not be able to know the correctness of the computed output for any particular input, we may know the relationship between relevant inputs and their outputs. Such a relation is referred to as a Metamorphic Relation (MR). MRs can be used to generate follow-up test cases and verify the outputs automatically. In future work, Metamorphic Testing technique can be applied to high level test case generation.

Another possible future direction is to explore how high-level test cases map to code-level test cases to check whether some code-level test cases are missing for verifying some important high-level requirement.

APPENDIX -A

A-1: Code for Tests Path Generation Tool: Goal-Plan Paths

Algorithm Implementation for Test Path generation Using Test Model (Goal-Plan Graph)

findpathsbytype ()

```
def findpathsbytype(self,node,start,coveragetype = 'goal',noit = 0, path = ""):
    res = []
    #if self.checknodes(node) == 1:
    #    return []
    if self.isleafnode(start):
        if (start['Visited'] == 0):
            start['Visited'] += 1
            path += ',' + start['number']
            return [path]
        else:
            if self.checknodes(node,path) == 1:
                return []
            elif self.checknodes(node,path) == 2:
                path += ',' + start['number']
                for x in path.split(','):
                    node[x]['Visited'] = 2
                return [path]
            else:
                return []
    else:
        if path.count(start['number']) <= noit:
            if start['number'] == '1' and start['Visited'] == 0:
                path += start['number']
                start['Visited'] = 1
            else:
                path += ',' + start['number']
            start['Visited'] += 1
            for child in start['andedgh']:
                res += self.findpathsbytype(node,node[child],coveragetype,noit,path)
            for child in start['loopedgh']:
                res += self.findpathsbytype(node,node[child],coveragetype,noit,path)
            if len(start['oredgh']) != 0 :
                if self.checkCommonNodes(start,node) == 1:
```

```

        child = random.choice(start['oredgh'])
        res += self.findpathsbytype(node,node[child],coveragetype,noit,path)
    else:
        for child in start['oredgh']:
            res += self.findpathsbytype(node,node[child],coveragetype,noit,path)
return res

```

findpathsbymetadata ()

```
def findpathbymatadata(self,nodes,matadata,data,dataini):
```

```

    retc = 0
    paths = self.findall(nodes,nodes['1'])
    paths.sort(lambda x,y: cmp(len(y), len(x)))
    goalnodes = []
    for x in nodes:
        if (nodes[x]['metadata'].count(dataini)) == 1:
            goalnodes += [x]
    goalnodes_forprint = goalnodes
    goalnodes = set(goalnodes)
    result = []
    for x in paths:
        if len(goalnodes&set(x)) > 0:
            result.append(x)
    cnt = 1
    skip = 0
    for x in result:
        msg = "
        pthis = 0
        temp = x.split(',')
        for y in range(len(temp)):
            if temp[y] in goalnodes_forprint:
                z = ((nodes[temp[y]]['metadata'].Split(',')))
                for o in z:
                    if o.ToString().count(dataini) >= 1:
                        if o.count '[' >= 1:
                            z = o.replace('[','")
                        elif o.count ']' >= 1:
                            z = o.replace(']','")
                        else:
                            z = o
                    if matadata[data].Contains(z):

```

```

        matadata[data].remove(z)
        pthis = 1
        break
    temp[y] = temp[y]+'(+z+)'
if pthis == 1:
    self.txtb_output.Text += str(cnt)+':> '
    for y in range(len(temp)-1):
        self.txtb_output.Text += temp[y]+' '
    self.txtb_output.Text += temp[-1] + '\n'
    cnt += 1
    pthis = 0
if 0 == len(matadata[data]):
    break

```

findall ()

```

def findall(self,node,start,noit = 0, path = ""):
    res = []
    if len(start['andedgh']) == 0 and len(start['oredgh']) == 0:
        if (start['Visited'] == 0):
            #start['Visited'] = 1
            path += ',' + start['number']
            return [path]
        else:
            return []
    else:
        if path.count(start['number']) <= noit:
            if start['number'] == '1' and start['Visited'] == 0:
                path += start['number']
                start['Visited'] = 1
            else:
                path += ',' + start['number']
            for child in start['andedgh']:
                res += self.findall(node,node[child],noit,path)
            for child in start['loopedgh']:
                res += self.findall(node,node[child],noit,path)
            if len(start['oredgh']) != 0 :
                for child in start['oredgh']:
                    res += self.findall(node,node[child],noit,path)
                #child = random.choice(start['oredgh'])
                #res += self.findall(node,node[child],noit,path)

```

```
return res
```

findloop ()

```
def findloop(self,node,start,noit = 0, path = ""):
    res = []
    if len(start['andedgh']) == 0 and len(start['oredgh']) == 0:
        if (len(start['loopedgh']) == 1):
            #start['Visited'] = 1
            path1 = path + ',' + start['number']
            for child in start['loopedgh']:
                res += self.findloop(node,node[child],noit,path1)
            return [path1] + res
        else:
            path += ',' + start['number']
            return [path]
    else:
        if path.count(start['number']) <= noit:
            if start['number'] == '1' and start['Visited'] == 0:
                path += start['number']
                start['Visited'] = 1
            else:
                path += ',' + start['number']
            for child in start['andedgh']:
                res += self.findloop(node,node[child],noit,path)
            for child in start['loopedgh']:
                res += self.findloop(node,node[child],noit,path)
            if len(start['oredgh']) != 0 :
                for child in start['oredgh']:
                    res += self.findloop(node,node[child],noit,path)
                #child = random.choice(start['oredgh'])
                #res += self.findloop(node,node[child],noit,path)
    return res
```

APPENDIX-B

Test Cases Execution Log

Sample Interaction Testing Log:

1--> createAccount(Peter, YEN, 15) |

Result:

User to BankAccount agent --> Percept (account open)

BankAccount agent to User --> Action (account created)

Created account 1 for Peter in YEN with opening balance 15.0

2--> createAccount (shafiq, AUD, 100)| credit (shafiq, AUD, 40)

Result:

User to BankAccount agent --> Percept (account open)

BankAccount agent to User --> Action (account created)

Created account 2 for Shafiq in AUD with opening balance 100.0

accountowner to BankAccount agent --> Percept (Credit account request)

BankAccount agent to account owner --> Action (amount credited)

BankAccount agent to accountowner --> action (Account info)

Credited account 2 with AUD40.0. Balance: 140.0

3--> createAccount(ali, USD, 40)| credit(ali, USD, 15)| debit(ali, USD, 15) |

Result:

User to BankAccount agent --> Percept (account open)

BankAccount agent to User --> Action (account created)

Created account 3 for ali in USD with opening balance 40.0

accountowner to BankAccount agent --> Percept (Credit account request)

BankAccount agent to account owner --> Action (amount credited)

BankAccount agent to accountowner --> action (Account info)

Credited account 3 with USD15.0. Balance: 55.0

accountowner to BankAccount agent --> Percept (Debit account request)

BankAccount agent to account owner --> Action (amount debited)

BankAccount agent to accountowner --> action (Account info)

Debited account 3 with USD40.0. Balance: 15.0

4--> createAccount (ali, YEN, 50) | credit (ali, AUD, 40) | debit (ali, AUD, 40) |

Result:

User to BankAccount agent --> Percept (account open)

BankAccount agent to User --> Action (account created)

Created account 5 for ali in YEN with opening balance 50.0

accountowner to BankAccount agent --> Percept (Credit account request)

BankAccount agent to Communicator agent --> Message (TransportRequest)

Communicator agent to CurrencyExchange agent --> Message (ExchangeRequest)

user to CurrencyExchange agent --> Percept (exchange rates)

Communicator agent to CurrencyExchange agent --> Message (ExchangeRequest reply)

CONVERTED 40.0

BankAccount agent to account owner --> Action (amount credited)

BankAccount agent to accountowner --> action (Account info)

Credited account 3 with YEN1000.0. Balance: 1050.0

accountowner to BankAccount agent --> Percept (Debit account request)

BankAccount agent to Communicator agent --> Message (TransportRequest)

Communicator agent to CurrencyExchange agent --> Message (ExchangeRequest)

user to CurrencyExchange agent --> Percept (exchange rates)

Communicator agent to CurrencyExchange agent --> Message (ExchangeRequest reply)

CONVERTED 40.0

BankAccount agent to account owner --> Action (amount debited)

BankAccount agent to accountowner --> action (Account info)

Debited account 3 with YEN1000.0. Balance: 50.0

Sample Goal-Plan Coverage Log:

1--> createAccount (ali,GBD,40) | credit(ali,GBD,500)| debit (ali,GBD,15) |

Result:

obtain information(goal)

obtain informationP(Plan)

Create Account(goal)

Test CreateAccountPlan

Create AccountP(Plan)

Account InfoPlan(Plan)

Created account 1 for ali in GBD with opening balance 40.0

Account Operations(goal)

Account OperationsP(Plan)

Credit Account(goal)

Credit AccountP(Plan)

CreditAccount (goal)

CreditAccountPlan (Plan)

CreditAccountExchange (goal)

CreditAccountExchangePlan (Plan)

Credited account 1 with GBD500.0. Balance: 540.0

Account Operations(goal)

Account OperationsP(Plan)

Debit Account(goal)

Debit AccountP(Plan)

Debit Account(goal)

Debit AccountPlan(Plan)

DeditAccountExchange (goal)

DebitAccountExchangePlan (Plan)

Debited account 1 with GBD15.0. Balance: 525.0

2--> createAccount (peter,YEN,40) |

Result:

obtain information(goal)

obtain informationP(Plan)

Create Account(goal)

Test CreateAccountPlan

Create AccountP(Plan)

Account InfoPlan(Plan)

Created account 2 for Peter in YEN with opening balance 40.0

3--> createAccount (Shafiq,YEN,15) | credit (Shafiq,YEN,100) |

Result:

obtain information(goal)

obtain informationP(Plan)

Create Account(goal)

Test CreateAccountPlan

Create AccountP(Plan)

Account InfoPlan(Plan)

Created account 3 for Shafiq in YEN with opening balance 15.0

Account Operations(goal)

Account OperationsP(Plan)

Credit Account(goal)

Credit AccountP(Plan)

CreditAccount (goal)

CreditAccountPlan (Plan)

CreditAccountExchange (goal)

CreditAccountExchangePlan (Plan)

Event ExchangeRequest

CurrencyExchange (goal)

SendAndWait(Plan)

SetExchangeRates (Goal)

SetExchangeRates (Plan)

PerformExchange (Goal)

PerformExchange (Plan)

ComputeRate (goal)

TwoStepRatePlan (Plan)

CONVERTED 100.0

Credited account 4 with YEN 500.0. Balance: 515.0

REFERENCES

- Abushark Y., Thangarajah J., Miller T., Harland J., Winikoff M., 2015, “Early Detection of Design Faults Relative to Requirement Specifications in Agent-Based Models”, Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May 4–8, Istanbul, Turkey.
- Abushark Y., Thangarajah J., Miller T., Harland J., 2014, “Checking Consistency of Agent Designs Against Interaction Protocols for Early-Phase Defect Location”, Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), Paris, France.
- Alonso F., Fuertes J., Martínez L. and Soza H., 2009, “Towards a set of Measures for Evaluating Software Agent Autonomy”. Eighth Mexican International Conference on Artificial Intelligence, 978-0-7695-3933-1/09, DOI 10.1109/MICAI.2009.15.
- Busetta, P., Rönquist, R., Hodgson, A. & Lucas, A. (1998), JACK Intelligent Agents - components for intelligent agents in Java, Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia.
- Bashir M. B., Nadeem A., 2009, "Fitness Function Design for Evolutionary Testing of Object-Oriented Programs: A Survey", International Conference on Software, Knowledge and Information Management and Applications (SKIMA 2009), Fes, Morocco. October 21-23.
- Bordini R. H., Wooldridge M., and Hubner J. F., 2007, “ Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)”. John Wiley & Sons, New York, NY, USA, 2007.
- Bratman, M.: Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge, MA, 1987.
- Brazier M. T., Dunin-Keplicz B. M., Jennings N. R., and Treur J., 1997, “DESIRE: Modelling multi-agent systems in a compositional formal framework”, International Journal of Cooperative Information Systems, 1(6):67–94.

Bresciani P, Giorgini P, Giunchiglia F, Mylopoulos J, and Perini A., 2002, “Troops: An agent-oriented software development methodology”, Technical Report DIT-02-0015, University of Trento, Department of Information and Communication Technology.

Caire G., Leal F., Chainho P., Evans R, Garijo F, Gomez J, Pavon J, Kearney P, Stark J, and Massonet P., 2001, “Agent oriented analysis using MESSAGE/UML”. In Michael Wooldridge, Paolo Ciancarini, and Gerhard Weiss, editors, Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), pages 101-108.

Chen T. Y., 2015. Metamorphic testing: a simple method for alleviating the test oracle problem. In Proceedings of the 10th International Workshop on Automation of Software Test (AST '15). IEEE Press, Piscataway, NJ, USA, 53-54.

Cheon, Y., Kim, M., 2006, "A specification-based fitness function for evolutionary testing of object-oriented programs", Proceedings of the 8th annual conference on Genetic and evolutionary computation, Washington, USA.

Duff S., Thangarajah J., and Harland J., 2014, “Maintenance goals in intelligent agents”, *Computational Intelligence*, 30(1), 71–114, (2014).

D’Inverno M., Kinny D., Luck M., and Wooldridge M., 1998, “A Formal Specification of dMARS”. In M. Singh, A. Rao, and M. Wooldridge, editors, *Intelligent Agents IV Agent Theories, Architectures, and Languages*, volume 1365 of *Lecture Notes in Computer Science*, pages 155–176. Springer Berlin / Heidelberg.

Dam K. H., 2003, “Evaluating and Comparing Agent-Oriented Software Engineering Methodologies”, Master of Applied Science in Information Technology thesis, School of Computer Science and Information Technology, RMIT University, Australia.

Dam K. H., 2008, “Supporting Software Evolution in Agent Systems”, PhD Thesis, School of Computer Science and Information Technology, Science, Engineering, and Technology Portfolio, RMIT University, Melbourne, Victoria, Australia.

Dam K. H., Winikoff M., 2003, “Comparing Agent-Oriented Methodologies”, Fifth International Bi-Conference Workshop on Agent-Oriented Information Systems, Melbourne, July (AAMAS03).

DeLoach S. A., 1999, “Multiagent systems engineering: A methodology and language for designing agent systems”. In Agent-Oriented Information Systems '99 (AOIS'99), Seattle WA.

DeLoach S. A., Wood M. F., and Sparkman C. H., 2001, “Multiagent systems engineering.” *International Journal of Software Engineering and Knowledge Engineering*, 11(3):231-258.

Glaser N., 1996, “Contribution to knowledge modelling in a multi-agent framework (the CoMoMAS approach)”, PhD Thesis, L'Universite Henri Poincare.

Harland J., Morley D. N., Thangarajah J., Yorke-Smith N., 2014, “An operational semantics for the goal life-cycle in BDI agents”, *Autonomous Agent Multi-Agent System*, 28:682–719, DOI 10.1007/s10458-013-9238-9.

Huber M. J., 1999, “JAM: A BDI-theoretic Mobile Agent Architecture”. In *Agents'99: Proceedings of The 3rd International Conference on Autonomous Agents*, pages 236–243, New York, NY, USA, ACM.

Huguet M. P. and Odell J., 2004, “Representing agent interaction protocols with agent UML”. In *Proceedings of the Fifth International Workshop on Agent Oriented Software Engineering (AOSE)*.

IEEE, 1998, *Standard for Software Test Documentation*. IEEE STD 829, 1998. URL <http://standards.ieee.org/findstds/standard/829.html>.

Iglesias C., Garijo M., Gonzales J.C., and Velasco J.R., 1998, “Analysis and design of multi-agent systems using MAS-CommonKADS”. In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, *Intelligent Agents IV. Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, Lecture Notes in Artificial Intelligence Vol. 1365, pages 313-326. Springer-Verlag.

Ingrand F. F., Georgeff M. P., and Rao A. S., 1992, “An Architecture for Real-Time Reasoning and System Control”. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44.

Jack intelligent agents, <http://aosgrp.com/products/jack/> accessed on July 2014.

Jan T, 2004, “Model-Based Testing: Property Checking for Real”. Keynote Address at the International Workshop for Construction and Analysis of Safe Secure and Interoperable Smart Devices. <http://www-sop.inria.fr/everest/events/cassis04>.

Joe M. L., 1983, “A Theory of Error-based Testing”. Ph.D. thesis, University of Maryland at College Park, College Park, MD, USA.

Joe M. L., 1990, “A Theory of Fault-Based Testing”. In: IEEE Transactions on Software Engineering, volume 16(8):pp. 844– 857, 1990. ISSN 0098-5589. doi: <http://dx.doi.org/10.1109/32.57623>.

Juneidi S.J., and Vouros G.A., 2010, “Survey and Evaluation of Agent-Oriented Software Engineering Main Approaches”, International Journal of Modeling and Simulation – 2010, 10.2316/Journal.205.2010.1.205-4306.

Low C. K., Chen T. Y., Ronnquist R., 1999, “Automated test case generation for BDI agents”. Autonomous Agents and Multi-Agent Systems, 2(4), pp. 311-332.

Mathieson I., Dance, S., Padgham, L., Gorman, M., Winikoff, M., 2004, “An open meteorological alerting system: issues and solutions”. In Estivill-Castro, V., ed.: Proceedings of the 27th Australasian Computer Science Conference. Volume 26 of Conferences in Research and Practice in Information Technology, The University of Otago, Dunedin, New Zealand.

Miller T, Padgham L., Thangarajah J, 2010,: “Test Coverage Criteria for Agent Interaction Testing”, Agent-Oriented Software Engineering (AOSE) Workshop at AAMAS.

Munroe S., Miller T., Belecheanu R. A., Pechoucek M., McBurney P., and Luck M., 2006, “Crossing the agent technology chasm: Lessons, experiences and challenges in commercial applications of agents”, The Knowledge Engineering Review, Volume 21 Issue 4, Pages 345-392, Cambridge University Press New York, NY, USA.

Myers G. J., Sandler C., Badgett T., and Thomas T. M., 2004, The Art of Software Testing, Second Edition. Wiley.

Nguyen C. D., Miles S., Perini A., Tonella P., Harman M., and Luck M., 2012 “Evolutionary Testing of Autonomous Software Agents”. *Autonomous Agents and Multi-Agent Systems*, Volume 25, Issue 2, pp 260-283.

Nguyen C. D., Perinirini A., Tonella P., 2007, “Automated continuous testing of multi-agent systems”, In *Proceedings of the Fifth European Workshop on Multi-Agent Systems (EUMAS)*.

Nguyen C., Perini A., Tonella P., 2008, “eCAT: A tool for automating test case generation and execution in testing multi-agent systems”. In: *Proceedings of AAMAS*, Estoril, Portugal, pp. 1669–1670.

Nguyen D., Perini A., and Tonella P., 2008, “A Goal-Oriented Software Testing Methodology”, Springer Berlin / Heidelberg 10.1007/978-3-540-79488-2_5,

Nunez M., Rodriguez I., Rubio F., 2005, “Specification and testing of autonomous agents in e-commerce systems”. *Software Testing, Verification and Reliability*. Wiley Inter Science, DOI:10.1002/stvr.323.

Odell J., Parunak H. V. D., Bauer B., 2000, “Extending UML for Agents” , *Proc. of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence*, Gerd Wagner, Yves Lesperance, and Eric Yu eds., Austin, TX, pp. 3-17 accepted paper, AOIS Workshop at AAAI.

Omicini, 2001, “SODA : Societies and infrastructures in the analysis and design of agent-based systems”, In P. Ciancarini and M.J. Wooldridge, editors, *Agent-oriented software engineering. Proceedings of the First International Workshop (AOSE-2000)*, Lecture Notes in Artificial Intelligence, Vol.1957, pages 185-194. Springer-Verlag.

Object Management Group, UML 2.0 superstructure specification. Object Management Group, available from www.omg.org, document ptc/03-08-02., 2003.

Padgham L., Zhang Z., Thangarajah J. and Miller T., 2013, “Model-based test oracle generation for automated unit testing of agent systems”. *IEEE Transactions on Software Engineering*, 39(9):1230–1244.

Padgham L. and Winikoff M., 2004, “Developing Intelligent Agent Systems: A Practical Guide”. John Wiley and Sons, New York, NY, USA.

Padgham L., and Winikoff M., 2003, “Prometheus: A Methodology for Developing Intelligent Agents”, *Agent-Oriented Software Engineering III, Lecture Notes in Computer Science Volume 2585*, pp 174-185.

Padgham L., Thangarajah J. and Winikoff M., 2006, “The Prometheus design tool - a conference management system case study”. *Proceedings of the 8th international conference on Agent-oriented software engineering VIII*, Pages 197-211.

Padgham L., Thangarajah J., and Winikoff M., 2014, “Prometheus Research Directions”, chapter 8, O. Shehory and A. Sturm (eds.), *Agent-Oriented Software Engineering*, DOI 10.1007/978-3-642-54432-3__8, Springer-Verlag Berlin Heidelberg.

Paul A. Offutt J., 2008, “Introduction to Software Testing”. Cambridge University Press, New York, NY, USA, 2008. ISBN 9780521880381.

Pokahr A., Braubach L., and Lamersdorf W., 2003, “JADEX: Implementing A BDI-Infrastructure for JADE Agents. *EXP – In Search of Innovation*”, 3(3):76–85, 2003.

Rao A. S. and Georgeff M. P., 1995, “BDI Agents: From Theory to Practice”. In *ICMAS-95: Proceedings of The 1st International Conference of Multi-Agent Systems*, pages 312–319, Menlo Park, CA, USA, 1995. AAAI Press.

Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In: J. Allen, R. Fikes, E. Sandewall (eds.) *Principles of Knowledge Representation and Reasoning, Proceedings of the Second International Conference*, pp. 473–484. Morgan Kaufmann (1991)

Rao, A.S., 1996, : *AgentSpeak(L): BDI agents speak out in a logical computable language*. In: Van de Velde, W., Perram, J.W. (eds.) *MAAMAW 1996. LNCS (LNAI)*, vol. 1038, pp. 42–55. Springer, Heidelberg.

Rehman S U., Nadeem A., 2011, “AgentSpeak (L) bases testing of autonomous agents”, *The 2011 International Conference on Advanced Software Engineering & Its Applications (ASEA 2011)*, Jeju Island, Korea, Science and Engineering Research Support Society, pp. 11–20, © Springer-Verlag Berlin Heidelberg.

Rehman S. U., Nadeem A., 2013, “Testing of Autonomous Agents: A Critical Analysis”, Saudi International Electronics, Communications and Photonics Conference (SIEPC-13). DOI: 10.1109/SIEPC.2013.6550990, Page(s): 1 – 5

Riemsdijk M. B. van and Yorke-Smith N.,2010, ‘Towards reasoning with partial goal satisfaction in intelligent agents’, in Proc. of ProMAS’10, pp. 41–59.

RMIT, agent research Group, Australia <http://www.cs.rmit.edu.au/agents/pdt/tutorial/Tutorial.html> accessed on July 2014.

Software Testing. <http://standards.ieee.org/findstds/standard/29119-1-2013.html>. Accessed on July 2014.

Spillner, 1995, “Test criteria and coverage measures for software integration testing”. Software Quality Journal 4(4), 275–286.

Shaw P. and Bordini R. H., 2011, “An Alternative Approach for Reasoning about the Goal-Plan Tree Problem”, Languages, Methodologies, and Development Tools for Multi-Agent Systems Volume 6822 of the series Lecture Notes in Computer Science pp 115-135.

Shaw P., Farwer B., and Bordini R. H., 2008, “Theoretical and experimental results on the goal-plan tree problem”, In Proc. 7th Int. Conf. on Autonomous Agents and Multi-agent Systems.

Taipale o, Smolander k, and Kalviainen H, 2005, “Finding and Ranking Research Directions for Software Testing”, In I. Richardson, P. Abrahamsson, and R. Messnarz, editors, Software Process Improvement, volume 3792 of Lecture Notes in Computer Science, pages 39–48. Springer Berlin / Heidelberg.

Tian J, “Quality Assurance Alternatives and Techniques: A Defect Based Survey and Analysis,” Software Quality Professional, vol. 3, no. 3, pp. 6-18, 2001.

Thangarajah J., Sardina S., Padgham L., 2012, “Measuring Plan Coverage and Overlap for Agent Reasoning”, Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012), Conitzer, Winikoff, Padgham, and van der Hoek (eds.), June, 4–8, Valencia, Spain.

Thangarajah J. and Padgham L., 2011, “Computationally Effective Reasoning About Goal Interactions”, *Journal of Automated Reasoning* (2011) 47:17–56, DOI 10.1007/s10817-010-9175-0.

Thangarajah J., Jayatilleke G., and Padgham L., 2011, “Scenarios for system requirements traceability and testing”. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 285–292. International Foundation for Autonomous Agents and Multiagent Systems.

Thangarajah J., Harland J., Morley D. N., and Yorke-Smith N., 2014, “Quantifying the Completeness of Goals in BDI Agent Systems”, *ECAI*, T. Schaub et al. (Eds.).

Thangarajah J., Harland J., Morley D. N. and Yorke-Smith N., 2014, “Towards quantifying the completeness of BDI goals”. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems (AAMAS '14)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1369-1370.

Thangarajah J., Padgham L., and Winikoff M., 2003 ‘Detecting and avoiding interference between goals in intelligent agents’, in *Proc. of IJCAI’03*, pp. 721–726.

Thangarajah J., Harland J., and Yorke-Smith N., 2007, ‘A soft COP model for goal deliberation in a BDI agent’, in *Proc. of CP’07 Workshop on Constraint Modeling and Reformulation (ModRef’07)*, pp. 61–75.

Thangarajah J., Harland J., Morley D. N. and Yorke-Smith N., 2014, “ Quantifying the Completeness of Goals in BDI Agent Systems”, *ECAI*, T. Schaub et al. (Eds.)

Thangarajah J., Padgham L., and Winikoff M., 2005, “Prometheus Design Tool”, *Proceedings of the 4th International Conference on Autonomous Agents and Multi Agent Systems (AAMAS’05)*, July 25-29, 2005, Utrecht, Netherlands.

Utting M., Legeard B. 2006, “Practical Model-Based Testing: A Tools Approach”. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123725011.

Utting M., Legeard B., 2007, “Practical Model-Based Testing: A Tools Approach”. Morgan-Kaufmann, San Francisco.

Wegener J., Baresel A., Sthamer H., 2001, “Evolutionary test Environment for automatic structural testing”, *Information and software technology*, PP 841-854.

Winikoff M, Cranefield S., 2010, “On the testability of BDI agents”, *European Workshop on Multi-Agent Systems* .

Winikoff M., 2005, “JACK Intelligent Agents: An Industrial Strength Platform”. In G. Weiss, R. Bordini, M. Dastani, J. Dix, and A. Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15, pages 175–193. Springer US.

Winikoff M., 2005, “Towards making agent UML practical: a textual notation and a tool”, *Fifth International Conference on Quality Software, (QSIC 2005)*. 10.1109/QSIC.2005.69, PP 401 – 406.

Wooldridge M, 2002, *An Introduction to MultiAgent Systems*. John Wiley & Sons, Chichester, UK. ISBN 0 47149691X, <http://www.csc.liv.ac.uk/mjw/pubs/imas/>.

Wooldridge M., Jennings N.R., and Kinny D., 2000, “The Gaia methodology for agent-oriented analysis and design”. *Autonomous Agents and Multi-Agent Systems*, 3(3).

Zhang Z., Thangarajah J., and Padgham L., 2009, “Model Based Testing for Agent Systems”. In J. Filipe, B. Shishkov, M. Helfert, and L. A. Maciaszek, editors, *Software and Data Technologies, Communications in Computer and Information Science*, volume 22, pages 399–413. Springer Berlin Heidelberg,.

Zhang Z., Thangarajah J., and Padgham L., 2011, “Automated testing for intelligent agent systems”. In Marie-Pierre Gleizes and Jorge Gomez-Sanz, editors, *Agent-Oriented Software Engineering X*, volume 6038 of *Lecture Notes in Computer Science*, pages 66–79. Springer Berlin / Heidelberg.

Zhang Z., Thangarajah J., Padgham L., 2007, “Automated Unit Testing For Agent Systems”, *Proceedings of the 7th international joint conference on Autonomous agents and multi-agent systems*.

Zheng M., Alagart V. S., 2005, “Conformance Testing of BDI Properties in Agent-based Software System”, *APSEC '05 Proceedings of the 12th Asia-Pacific Software Engineering Conference*, 0-7695-2465-6/05.

Zhou Y., Torre L. van der, and Zhang Y., 2008, 'Partial goal satisfaction and goal change: Weak and strong partial implication, logical properties, complexity', in Proc. of AAMAS'08, pp. 413–420.