computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

EARCT- ENVIRONMENT FOR AUTOMATED RANK BASED CONTINUOUS AGENT TESTING

Ashok Kumar [1], Vinay Goyal [2]

[1] Department of Computer Science & Applications, Kurukshetra University, Kurukshetra, India
[2] Panipat Institute of Engineering & Technology, Samalkha, Panipat, India
vinaykuk@gmail.com

Abstract: *Testing MAS (Multi-agent System) is a challenging task because these systems are distributed, complex and autonomous in nature. Agents exist in an open environment having their own locus of control and they require context awareness. So due to these agent's characteristics, testing MAS system using existing testing techniques becomes a very tedious job. Agents also pose problems regarding message communication and semantic interoperability, as well as synchronization with other agents existing in the environment. All these features are known to be hard not only to design and to code, but also to test. In this paper we will propose a unique environment EARCT to test MAS keeping in mind the essential software engineering paradigms such as effort consumed, errors revealed etc.*

Keywords: *Agents Testing Autonomous Ranking.*

1. INTRODUCTION

Current research and development on agent oriented technology mainly put emphasis on designing architecture, formalizing protocols, designing frameworks etc.Very limited research work has been carried out on testing multi agent system [1,2,3]. The agent-oriented paradigm is considered a natural extension to the object-oriented (OO) paradigm, but agents are different from objects in many ways [4,5]. Although there are well-defined OO testing techniques, agent-oriented development has neither a standard development process nor a standard testing technique. Since, in MAS (Multi agent system) there are several agents existing in an environment [6,7] as distributed components, which are proactive and autonomous, so it is possible that same inputs can provide different outputs on different execution. The problems posed by the agents and Multi agent system testing are well recognized [8], and some of the more significant ones are discussed in next section.

2. BACKGROUND

As noted above, this section identifies and describes some of the major problems posed by Multi-Agent Systems.

Autonomous and social nature: Agents are autonomous in nature and due to their social capabilities- they cooperate with other agents present in the environment. MAS testing tools must have a comprehensive view over all distributed agents in addition to local knowledge about individual agents, in order to check whether the whole system operate accordingly to the specifications or not. Moreover, it may be possible that a single agent ran successfully and correctly as a stand-alone entity but incorrectly in a community or *vice versa*.

Complexity: As agents are autonomous and are run concurrently in a distributed environment, it becomes very difficult to determine the exact boundary of a test case. Distributed and concurrent environments often pose a challenge before testing team.

Agent Communications: Agents communicate with each other via message passing and not by method invocation as in object oriented technology, so existing object oriented technology testing techniques are not applicable for agent based testing.

Non-Deterministic nature: Agent's nature is non-deterministic in nature because it is not possible to determine in prior all possible interactions of an agent during its execution.

Irreproducibility effect: Due to an agent's pro-active and autonomous nature, we cannot guarantee that two executions of the systems will lead to the same output state, even if the same inputs are used because agents can modify their knowledge between any two executions. As a result, looking for a specific error can be difficult if it cannot be replicated [9].

Amount of data: MAS can constitute numerous agents, as each agent processes its own data. To test such a huge amount of data is itself a very challenging task for the testing team. Also agents operate asynchronously and in a parallel manner which is also challenging for a testing team.

Agent Ranking: There is no way a testing team can assess out the importance or rank of a particular agent. It might be possible that a particular agent is acting as a core element of the system, and another as an outside scaled agent with less knowledge about its internal state and behavior. Agents with top ranking (most important agents) must be given extra attention and resources during testing and third party agents or agents with lower ranking may be tested with limited resources. This will result in more effective resource utilization during testing phase and will improve proper operation of the core functionalities for the whole system.

Agent Inter-Dependency: As agents are social in nature and are autonomous also, they can interact with other agents present in the environment to fulfill their goal. Because of this, there exist many execution paths which can be followed by an agent. It is difficult to test each and every path for correctness, completeness and consistency.

Black Box MAS: MAS can also be considered as «black-box»; that is, they may provide very little or some time no observational primitives to the outside world, resulting in limited access to the internal agents' state, their expected behavior and knowledge. This kind of MAS could be quite difficult to test, in that the test result (PASS or FAIL) may be hard to assess.

3. RELATED WORK

There is a very brief literature available on the testing of Agent. In fact, in recent times, few automated techniques are proposed to test agents and test strategies soon. Also work agent testing was done at various levels of testing such as unit-level (agent level), the level of integration (MAS) and system level. In fact, there is very little that AOSE explicitly define the test phase and test mechanisms. Zhang [10] introduced a framework for Model Based Testing using design patterns of the Prometheus agent development methodology. This framework focuses on testing agent plans (units) and mechanisms to generate test cases and appropriate to determine the order in which units are to be tested. Ekinci [11] stated that the goals of agents are the smallest testable units and MAS proposed to test these units through the test objectives. Each test objective is conceptually divided into three sub-goals: Installation (System Preparation), the objective of the test (perform actions on the objective), and purpose statement (check the satisfaction of goals). The first and last objective is preparing pre-conditions and post-conditions checked by testing the objective under test, respectively. In addition, they introduce a testing tool, called as SEAUnit that provides the infrastructure to support the proposed approach. Agile PASSI [12] proposes a framework to support trials of single agents. They develop a test suite specifically for the verification agent. Test plans are prepared before the coding phase according to the specifications and the tool is also capable of generating agents AgentFactory can also generate driver and stub to accelerate the testing of a specific agent. Lam and Barber [13] proposed a semi-automated process for understanding the behavior of software agents. The approach mimics what a human user (can be a tester) in the understanding of the software: building and refining knowledge base of agent behavior, and use it to verify and explain the behavior of agents at runtime. Nunez [14] introduced a formal framework for specifying the behavior of autonomous e-commerce agents. Desired behaviors of the agents being tested are presented using a new formalism, called state machine utility that embodies the users preferences in their states. Two test methods have been proposed to check if an implementation of a specified agent behaves as expected (i.e. compliance testing). In their approach to asset tests, they used for each test agent test (special agent) who makes the formal specification of the agent to facilitate reaching a specific state. The trace of the operational agent is then compared to the specification in order to detect defects. Moreover, the authors also proposed using the passive test in which the agents being tested, were only observed, not stimulated as in the active test. Invalid traces, if any, are then identified through formal specifications of agents. Coelho [15] proposed a framework for unit testing of MAS based on the use of simulated agents, which simulate real agents to communicate with the test agents were implemented manually, each corresponding to an agent role. Sharing the inspiration of JUnit [16] with Coelho [15], Tiryaki [17] proposed a test-driven development approach that supported MAS iterative and incremental construction MAS. A testing framework called SUnit, which was built above and JUnit Seagent [18], was developed to support the process. The framework allows writing tests for the agents' behavior and interactions between agents. Gomez-Sanz [19] introduces advance in testing and debugging methodology. In fact INGENIAS [5], the meta-model INGENIAS was extended with concepts for defining tests to integrate the reporting of the test, i.e. testing and test packets. Work has also provided facilities for access to mental states of individual agents to check them at runtime. Houhamdi [20] introduced an approach to derive test

suite for testing agent that is goal-oriented requirements analysis artifact that the basic elements for developing test cases. The proposed process has been illustrated with respect to the Tropos development process. It provides systematic guidance for generating test suites, the detailed design agent. These test suites, on the one hand, can be used to refine the analysis of objectives and to detect problems early in the development process. On the other hand, they are subsequently executed to test the objectives from which they were established. Agile [21] defines a test phase based testing framework JUnit [22]. To use this tool, designed for testing OO, MAS test in context, they need to implement a platform agent sequential, strictly used for testing, which simulates asynchronous message passing. The ACLAnalyser [23] tool runs on the JADE [24] platform, it intercepts all the messages exchanged between agents and stores them in a relational database. This approach exploits the clustering techniques to construct graphs of interaction of agents that support the detection of failed communication between agents that are expected to interact, configurations execution asymmetric and the data exchanged between agents. Padgham [25] uses design artifacts (e.g., interaction protocols and agent design specification) to provide automatic identification of the source of errors detected during execution. A debug agent is added to the central MAS to monitor the conversations of agents. It receives a carbon copy of every communication between agents, in a specific conversation. The interaction protocol specifications to the call are taken and analyzed to detect erroneous conditions automatically. Rodrigues [26] proposed to exploit the social conventions, norms, rules, that prescribe the authorizations, obligations, and / or ban agents in MAS open to an integration test. Information available in the specifications of these agreements give rise to a number of types of assertions, such as time to live the role, cardinality, and so on. During the test run of a special agent, called agent will report to observe a events and messages to generate analysis results thereafter. Nguyen [27] propose to use the ontology (s) extracted from MAS under test and a set of OCL constraints, which act as a test oracle. Having as an input a representation of the ontology (s) used, the idea is to build an agent capable of delivering messages whose content is inspired by these ontologies. The resulting behavior is believed to be correct by using the input set of OCL constraints: if the message satisfy the constraints, the message is correct, this procedure is supported by ECAT, a software tool. Houhamdi and Athamena [20] has introduced a new approach to goal-oriented software testing integration. They propose an approach to derive a test suite for integration testing, that takes goal-oriented artifact needs analysis to derive test cases. They discussed how to derive test suites for testing the integration of architectural design and detailed system objectives. These test suites can be used to observe the emergent properties, resulting from potential agents and make sure that a group of agents and contextual resources function properly together. This approach defines a structured test and overall integration and junction sequence of processes for engineering software agents by providing a systematic way to derive test cases from the analysis objective. Houhamdi and Athamena [28] introduced an approach to derive a test suite to test the system that is goal-oriented. Requirement analysis artifact that are the basic elements for developing test cases. The proposed process has been illustrated with respect to the Tropos development process, provides a systematic guidance for generating test suites for modeling artifacts, produced with the development process. They discussed how to derive test suites to test the system and delay the requirement to architectural design. These test suites, on the one hand, can be used to refine the analysis of objectives and to detect problems early in the development process. On the other hand, they are subsequently executed to test the objectives from which they were established.

4. AUTOMATED CONTINOUS TESTING

As observed from the issues highlighted above in the section 2, manual testing of MAS is a troublesome job. Testing MAS can be effectively done by automating the testing job and the process should be done in a continuous fashion. The continuous streaming of testing process is required due to uncertain and complex nature of MAS along with huge data to be tested. The proposed automated continuous testing framework will extensively test the system covering maximum units. The proposed framework complements the manual test case with each other rather than replacing the manual test cases. Due to continuous automated testing, various conditions which are responsible for errors can be reveled which are otherwise hard to reproduce manually. The other issue is to figure out the importance/rank of the module under consideration. The higher rank modules must be tested exhaustively while on the other hand, modules with lower rank of importance do not need exhaustive testing. This differentiation is done to make optimal use of available resources along with making the objective to produce an error free system. We will introduce two different strategies of testing, named as Rank Oriented Random Testing and Rank Oriented Exhaustive Testing. Depending upon the rank of a particular module, the appropriate testing strategies can be applied. The idea of introducing

rank based testing is to save time, cost and other resources incurred in testing process especially during testing of a multi agent system. The basic necessity of the testing process is to figure out the maximum number of errors in the system. The other consideration is of the agent interdependency. As agents are interacting with other agents in a loosely coupled environment, so establishing a valid inter-dependency relationship between the chains of interacting agents to fulfill the goal is really required. This can be achieved naturally because agents interact with each other by message passing. The valid states of the caller and cal lee agents in MAS can be checked during testing process in order to test the agent dependency. As the nature of MAS can change over time, may be between two successive executions, due to their learning capabilities, a single test case execution is not useful to reveal all the errors. The test time and number of test suites can be increased by using autonomous property of the agents. Automated continuous agents can continuously test the MAS units without the need of any human intervention and can proceed on their own without human attention. Continuous testing of MAS requires that the tester agent has the capability to develop existing test suites and to generate new test suites, with an aim of exercising and stressing the application as much as possible. The final goal is to reveal yet unknown faults.

5. EARCT (ENVIRONMENT FOR AUTOMATED RANK BASED CONTINOUS TESTING)

We propose a framework for automated rank based continuous testing named EARCT (Environment for Automated Rank based Continuous Testing). We propose three main components: The Autonomous tester agent, the observer agent, the ranking agents and two testing strategies: the rank based random test case generation and rank based progressive mutant test case generation.

The Autonomous Tester Agent: The dedicated autonomous tester agent will continuously test the MAS by means of message passing. It will continuously interact with the agents under test (AUT) by sending message to other agents, simulating the behavior of the caller or cal lee agents analogous to writing stubs and drivers in top down and bottom up testing respectively. The process will be autonomous and will execute in background, without any human intervention and continuous fashion in order to achieve the basic goal of revealing maximum faults in the system. The test suites of tester agents will contain the dummy messages to be sent to AUT. These messages can be extracted from the goal diagram using TROPOS.

The Observer Agent: The observer agent works like a watch dog over the autonomous tester agent and AUT. It observes the communication pattern between both of them. The observer agent will have knowledge about the pre and post state conditions of the AUT, error conditions, crash situations and even deadlock conditions. In case of any of the above mentioned case occurred the responsibility lies with the observer agent to inform about the problem(s) to the testing team. The testing team can then figure out the faults in the system. In figure 1 shown, the observer agent will be working as a master for the local observer agents. This is very much required because MAS works in heterogeneous environment. It is always the case that an agent in one environment will interact with the other agents in some other environment, and it is not necessary that the two environments has to be the same. To avoid side effects, the role of these local observer agents becomes very crucial. These local bodies report to the central observer agent who provides a global view about the agent inter-relationship and the environment. This global view in turn helps the ranking agent to evaluate the AUTs behavior after the inclusion of mutants into them. Thus the role of observing agent is to keep track of the interactions between AUTs and their pre and post conditions along with providing the execution scenario to the ranking agent. This covers the ‘black box’ problem discussed in the previous section.

The Ranking Agent: One of the issues related with the continuous automated testing is that how many test suites must be executed on AUT in order to get a satisfactory condition about functionality of the components or units. Applying an exhaustive testing technique on those functional units which are having low importance or we can say low ranking is not a good idea. In the same way, the AUT having high rankings must be tested fully using exhaustive testing technique discussed in the subsequent sections. For low rank AUT, random testing technique can be applied, which will save time and other efforts of the testing phase. The other issue is related with the agent inter-dependency. As agents interact with each other seamlessly with other agents in the environment, the chain of agent interdependency sometimes grows profusely. It gets difficult to test the long and complex chains of the communicating agents. The role of ranking agent is to provide pathway to the tester and observer agent to limit the number of test cases of AUT based upon their rankings and to limit the length of the chain of interacting agents. Higher the ranking, more rigorous testing will be followed on AUT and maximum inter-dependent chain of agents will be tested and *vice-versa.*

The main aim of EARCT is to enhance the efficiency and quality of one of the most important

phase of software engineering- The testing phase. Agents are very complex in nature and can pose extreme problems before the testing team. Testing based on ranks can help in figuring out maximum errors by using optimal resources in any MAS.

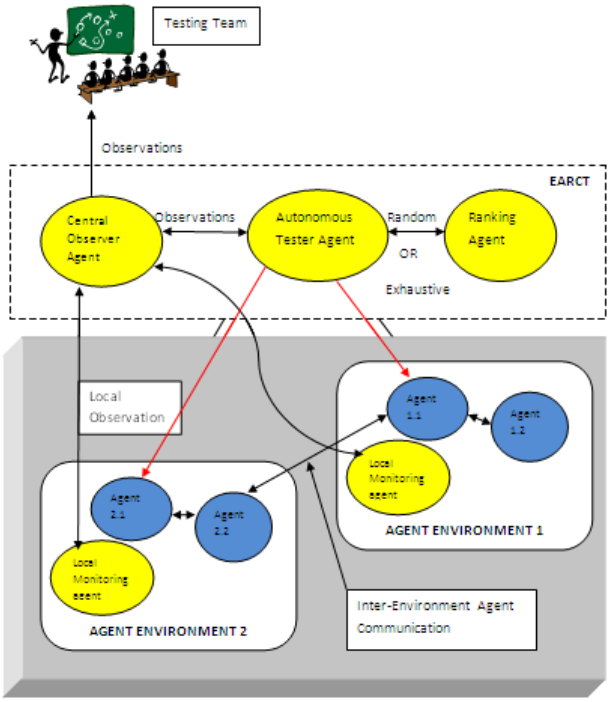

Fig. 1 – EARCT (Environment for Automated Rank based Continuous Testing)

6. TESTING STRAEGIES

As discussed earlier, the exhaustive testing is not possible for all the agents in the environment. The selective or random testing is done for low rank agents and exhaustive testing is done for the high rank agents. The two techniques are discussed in the following section:-

Rank Oriented Random Testing: The tester agent is capable of sending random generated data through random test data generation mechanisms [30, 31] to the AUT, using some communication protocol. The tester agent has to select one of the communication protocols available from the domain specification, to include meaningful and domain specific data into the communication messages. A model of the domain data, coming from the business domain of the MAS under test, must be also supplied. Various types of communication protocols are described and practiced such as UML based sequence diagram, complex cooperation protocol, activity diagrams, collaboration diagrams and the most widely accepted, the FIPA Interaction Protocol [29] in JADE [24]. In addition to the protocol, the format for the message passing between the tester agent and the AUT must also be supplied. One such type is FIPA ACLMessage [29]. Various ACLs (Agent Communication Language) are proposed by many researchers but the two most discussed and practiced ACLs are KQML (Knowledge Query and Manipulation language) and FIPA-ACL (Foundation for intelligent physical agent ACL). Both of them rely on speech act theory developed by Searle in 1960 and enhanced by Winograd and Flores in the 1970s. Speech act theory is derived from the linguistic analysis of human communication, based on the idea that with language the speaker not only makes statements, but also performs actions. But out of these two ACLs, FIPA-ACL is a standardization consortium. The JADE platform (Java Agent Development Environment) provides basis for FIPA-ACL. We will be using FIPA Interaction Protocol for agent communication and FIPA-ACL as the communication language. Due to their wide acceptability, the random test data generation technique then has to be selected by the tester agent. The random testing technique will be initiated by the rank agent based upon the ranking of the AUT. The testing team can decide the minimum (Min-R) and maximum rank (Max-R) number to initiate the testing strategy. Moreover, as discussed above, the number of AUTs collaborating together in a chain can form a huge series. The ranking of agents will be used to limit the chain for testing purpose. Higher the rank, more AUTs in the chain will be tested. The testing team can decide the minimum and maximum rank numbers and these values can be encoded in ACL message as parameters. The communication protocol can be extended to accommodate the ranks of the AUTs. In order to define the rigor of testing effort and to limit testing effort in the chain of interacting agents, the overall model of the rank based agent random testing will serve the following purpose:

- The model will prescribe the range and the structure of the data that are produced randomly, either in terms of generation rules or in the (simpler) form of sets of admissible data that are sampled randomly.
- Long and meaningful data and interaction sequence using random sampling is very hard to generate. The ranking mechanism of agents will limit the data and number of interactions between AUTs, making rank based random testing a cheap and efficient testing technique to reveal faults in the agent based system. Experimental details will be presented in the subsequent sections.

Randomly generated messages generated by the tester agent are then sent to AUT and response is observed by the observer agent. The response can be a successful state transition, an exception, a deadlock condition or a crash etc. Whenever the observer agent observes a divergence from the agent's expected behavior, the exception(s) is reported back to the testing team. It is the

responsibility of the observer agent to keep track of the actions and reactions occurring between the tester agent and the AUTs. The idea is not to fully automate the testing process but to support the manual testing in order to improve overall testing experience by reducing testing effort.

Rank Oriented Exhaustive Testing: Ranking Agent help the autonomous tester agent to decide on which AUT, random testing technique has to be applied. In the case where it has been decided by the Ranking and tester agent on the basis of ranking that random testing is not sufficient for the AUT, another testing strategy has to be applied. For those AUTs, having higher order of ranking, exhaustive testing is useful. It is evident that longer the sequence generated for the series of interacting agents, the likelihood of revealing faults is maximized, required for higher degree ranking holder AUTs. Sophisticated techniques as compared to random testing are required. One such proposed technique is rank oriented exhaustive testing. We are proposing the combination of progressive testing and mutation testing with the name- agent oriented rank based exhaustive testing. Simple exhaustive testing for a chain of agents is not possible due to amount of data to be tested. So for higher ranking agents, a combination of progressive and mutation testing [32, 33] can be done. Mutation testing is a kind of software testing, which involves modifying agent's source code in small ways. Mutation operators will be applied to original source code to inject the artificial and known defects. A mutant can be a modified branch condition, a wrong variable name or a modified method invocation process etc. A test suite will be considered as defective which does not detect and reject the mutated code. On the contrary, if the test case is able to detect the artificially seeded defect in the program, the mutant is considered to be 'killed'. The adequacy of the test case is measured as the ratio of total killed mutants over total number of mutants generated. The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution. This is required for higher ranking AUTs or chain of AUTs to make them more reliable and robust. The next step is to combine the mutation testing with the concept of progressive testing and ranks of the AUTs. For moderate or higher ranking AUTs, this kind of exhaustive may be applied. The decision it again left with the testing team depending upon the project management and software engineering requirements. In exhaustive testing, we assume that if we want to have a best test suit, it can be developed gradually in a progressive way [34, 35]. The idea is to evolve test suites by applying mutation operators to the test cases themselves by the tester agent. The ranking agent will provide a pathway to the tester agent in the form of a heuristic value, which is the defined as the shortfall of the test case to achieve the testing goal. Lower the value (i.e. test case is an effective one), it is more likely to happen that ranking agent will choose the particular test case for progression into another stringent version. It is assumed that test suites, which can kill maximum mutants will have higher probability to reveal faults. We have designed a simulator for the same as shown in figure 2.

Phase1:	**Mutants and Test Case Generation:** 1.1 ← For test on AUTs in MAS, generate mutants {M_1, M_2... M_n} 1.2 ← Get Rank values {R_1, R2... R_n} from the ranking agent for each AUT 1.3 ← Apply all/maximum mutants to higher rank AUTs and one or more to lower rank AUTs. 1.4 ← Randomly generate test cases {TC_1, TC_2... TC_n}
Phase 2:	**Execution Phase and heuristic value evaluation:** 2.1 ← Autonomous tester agent to execute each test case on all mutants generated in the phase 1. 2.2 ← Ranking agent will compute the heuristic value of each test case: HV[TC_i]=MK_1/n, where HV stands for heuristic value, MK stands for mutants killed and n stands for total number of mutants generated.
Phase 3:	**Test case progression:** 3.1 ← Select test cases with higher order of heuristic value HV[TC_i]. 3.2 ← Apply another set of generated mutants on the test case selected by the ranking agent. 3.3 Add the new mutated test cases to the existing set of test cases.
Phase 4:	**Check Status:** 4.1 ← If maximum numbers of generations are achieved then STOP and EXIT. else 4.2 ← Check improvements in the heuristics value for all the progressive iterations. 4.3 ← If there is no improvement, go to step 1.1 else 4.4 ← Report test results to testing team.

Fig. 2 – Simulator design for Rank Oriented Exhaustive Testing

7. EXPERIMENTAL DETAILS

We have simulated the EARCT environment using a hypothetical case study derived from a hospitality sector. We have also used TAOM4E (Tool for Agent Oriented Visual Modeling for the Eclipse platform) for goal oriented modeling, code generation and testing for goal-directed system. As TAOM4E follows TROPOS methodology as its basis, we will also use the same methodology to implement our case study. TAOM4E supports TROPOS's early and late requirement modeling requirements, architectural design and also provides support for automated agent oriented implementation and testing. TAOM4E is developed by the Software Engineering unit at Fondazione Bruno Kessler (FBK), Trento. The current version

0.6.3 is downloadable under GPL license from the tool homepage http://selab.fbk.eu/taom. The t2x code generation tool provided by TAOM4E can transform problem domain to solution domain by mapping goal models to a goal-directed implementation on the Jadex BDI agent platform. Explicitly preserving goal models at run-time and providing the proper middleware for navigating this model and acting according to it. Agent code can be generated from the graphical interface, and the implemented prototypes are executable directly from the Eclipse user interface. The recent addition in TAOM4E framework is Goal Oriented Testing tool to support testing and validation along the process phases. The EARCT framework will directly derive test cases from the goal models of TROPOS and uses them to implement agents.

8. RESULT AND CONCUSION

We have done manual testing as well as testing using EARCT framework on the case study. In manual testing, we applied two manual testing techniques viz. random testing which includes branch coverage and mutation testing. The choice is obvious, we will perform the same type of testing using EARCT framework by rank oriented random testing and rank oriented exhaustive testing strategies. The results will be compared then. The results are compared on the basis of three parameters: 1) Number of errors revealed and error type, where error type could be classified as fatal, Moderate or Low based upon the severity impact of these errors on the software, if these errors would have remained hidden during testing. 2) Effort utilized; this is measured as the time taken in minutes to figure out the errors(s) in a particular module/agent. 3) As discussed, the agents can form a complex chain reaction to fulfill the social goal. We are considering one more parameter i.e. number of linked branches/modules/AUTs tested to test the unit in question. Longer the tested chain, more the team will have confidence in the testing process.

The desirable software engineering scenario is to have maximum number of errors revealed by testing maximum units in the chain and utilizing minimum effort in terms of time spent on testing.

Further, as already discussed, using agent based testing we are also implementing the concept of ranks for the agents. The ranking is done in ascending order i.e. agents having ranking «1» will be the most important module and module with ranking «2» is the next unit in the importance list. The ranks will be fetched directly from the requirement phase. Again the idea is to put maximum effort on the important modules and pay somewhat less emphasis on the less important module. This is necessary to as to optimize the quality-effort ratio. As agents based systems are very complex in nature, it is not feasible to test each and every agent exhaustively, rather few important one can be tested fully and rest partially to save time, resources and efforts.

In the table 1 shown below, we have manually tested four modules from the case study using Random Branch Testing. Table 1 shows the time taken (in minutes) to test the module, number of errors revealed, error description, error type and number of linked branches tested. In table 2, the table is having one more column in the end- the rank of the module. Again table 3 and 4 shows the parameters obtained using manual mutation testing and agent based exhaustive testing respectively.

Table 1. Random Branch Testing (Manual)

Random Branch Testing (Manual)					
Module	**Effort (Time taken in Mins) (Manual Mode)**	**No. of errors Revealed (Manual Mode)**	**Error Description**	**Error Type**	**Branch level covered (Manual Mode)**
1	20	3	Calculation Error	FATAL	3
2	34	2	Boundary-check related error	MODERATE	2
3	17	4	Boundary-check related error	MODERATE	5
4	23	5	Compatibility and intersystem defect	MODERATE	3

Table 2. Agent Based Continuous Rank Oriented Random Testing

EARCT-Agent Based Continuous Rank Oriented Random Testing						
Module	**Effort (Time taken in Mins.) (Agent Based)**	**No. of errors Revealed (Agent Based)**	**Error Description**	**Error Type**	**Number of AUTs Covered**	**Rank**
1	11	4	Calculation Error, Control Flow error	Fatal	9	**2**
2	6	3	Boundary check missing	Medium	6	**3**
3	13	7	Unhandled condition, Calculation logic error, Control Flow defects	Fatal	13	**1**
4	10	4	Performance bug	Low	6	**4**

Table 3. Mutation Testing (Manual)

Manual Mutation Testing					
Module	**Effort (Time taken in Mins) (Manual Mode)**	**No. of errors Revealed (Manual Mode)**	**Error Description**	**Error Type**	**Modules Covered (Manual Mode)**
1	20	2	Keyword constraint violation	FATAL	3
2	34	2	Update Anomaly	FATAL	2
3	17	4	Return Empty	LOW	5
4	23	5	Un-expected Result	MODERATE	3

Table 4. Agent Based Continuous Exhaustive Testing

EARCT-Agent Based Continuous Exhaustive Testing						
Module	**Effort (Time taken in Mins) (Agent Based)**	**No. of errors Revealed (Agent Based)**	**Error Description**	**Error Type**	**Number of AUTs Covered**	**Rank**
1	11	4	Calculation Error, Control Flow error	Fatal	9	**2**
2	6	3	Boundary check missing	Medium	6	**3**
3	13	7	Unhandled condition, Calculation logic error, Control Flow defects	Fatal	13	**1**
4	10	4	Performance bug	Low	6	**4**

As shown in figures 3, 4, 5, 6, 7, 8 the comparative results are highly noticeable. The time taken by the automated agent based testing is substantially low as compared to the manual one for each of the module. The evaluated error count and linked units/agents covered are also on the higher end as compared to the manual one. One noticeable observation is that in case of module 4, the performance delivered by the agent based testing mechanism is not that good in terms of evaluated error count because of its low ranking. The exhaustive strategy is not applied on this module due to its low ranking, resulting in less error discovery and less linked units covered. This is one compromise the testing team has to make while taking decisions about the subsequent testing strategy and other software engineering paradigms like correctness, completeness, consistency and of course time and quality. If the module is having higher ranking the resources utilized will be on the higher end and vice-versa.

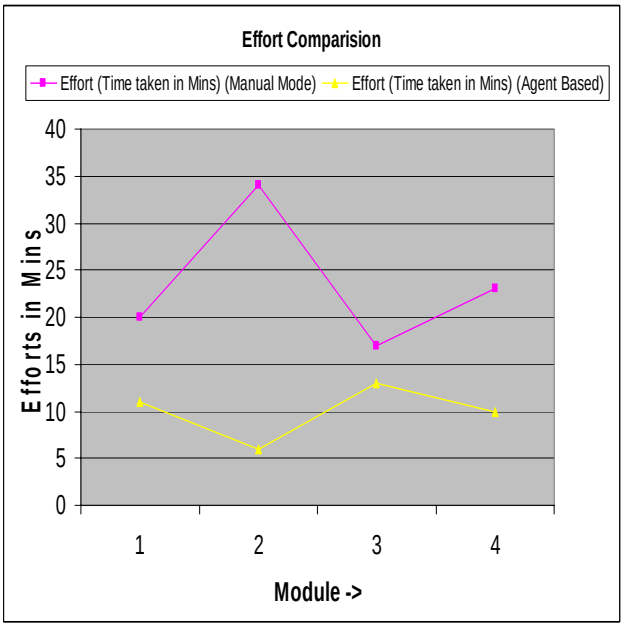

Fig. 3 – Effort Comparison- Manual Random Branch Testing Vs Agent Based Continuous Rank Oriented Random Testing

Fig. 4 – Evaluated Error Count Comparison- Manual Random Branch Testing Vs Agent Based Continuous Rank Oriented Random Testing

In this paper, we have shown that current manual techniques for testing are not good enough to answer the issues mentioned in the earlier sections. The problems like autonomy, social nature, complexity, agents' inter-dependency and non-deterministic nature can be answered using continuous testing only. Further to optimize various software

engineering paradigms like number of errors revealed in unit time and effort spent on the chain of interacting or inter-dependent agents etc, the agent ranking method is useful to help the testing team to take decisions about the subsequent steps.

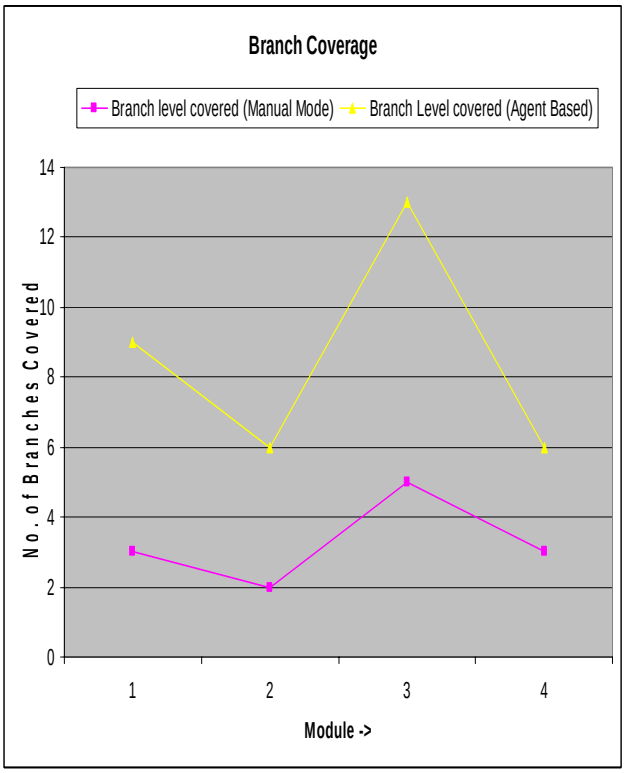

Fig. 5 – Branch/AUTs Coverage Comparison- Manual Random Branch Testing Vs Agent Based Continuous Rank Oriented Random Testing

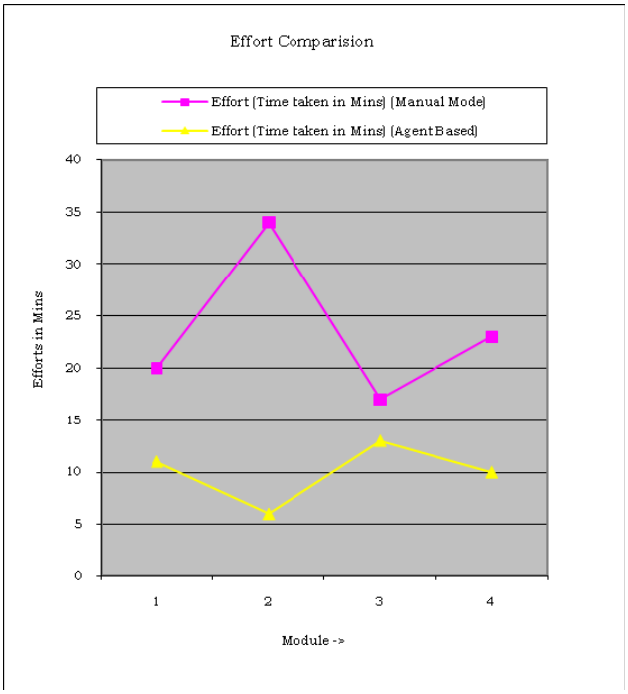

Fig. 6 – Effort Comparison- Manual Mutation Testing Vs Agent Based Continuous Rank Oriented Random Testing

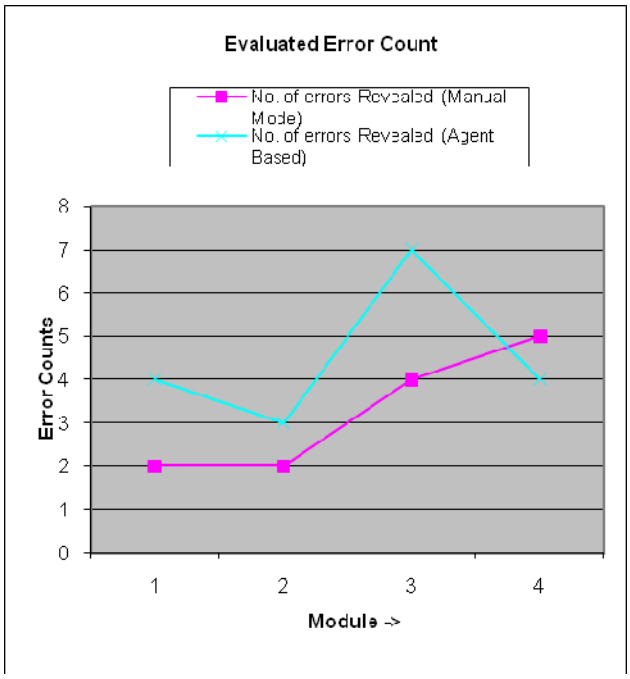

Fig. 7 – Evaluated Error Count Comparison- Manual Mutation Testing Vs Agent Based Continuous Rank Oriented Random Testing

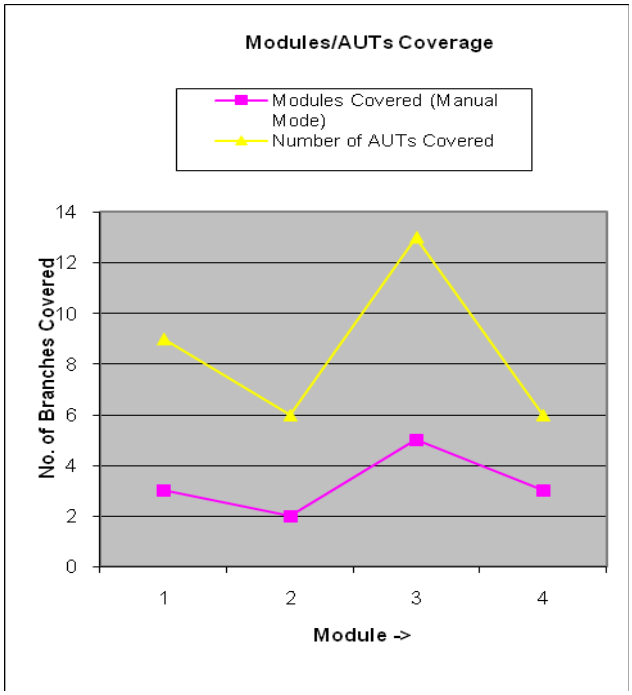

Fig. 8 – Modules/AUTs Coverage Comparison- Manual Mutation Testing Vs Agent Based Continuous Rank Oriented Random Testing

8. REFERENCES

[1] Tiryaki A.M., Oztuna S., Dikenelli O., and Erdur Sunit R., A unit testing framework for test driven development of multi-agent systems, In *7th International Workshop on Agent Oriented Software Engineering*, 2006.

[2] Roberta Coelho A.S., Kulesza U and Lucena

C., Unit testing in multi-agent systems using mock agents and aspects, *International Workshop on Software Engineering for Large-scale Multi-Agent Systems*, May 2006.

[3] Blaya J. A. B., Hernansaez J. M., and Gomez Skarmeta A. F., Towards and approach for debugging multi-agent systems through the analysis of agent messages" Comput. Syst. Sci. Eng., (20) 4 (2005).

[4] Cossentino M., *From requirements to code with PASSI methodology*, In Vijayan Sugumaran (Ed.), Intelligent Information Technologies: Concepts, Methodologies, Tools, and Applications, USA, 2008.

[5] Pavon J., Gomez-Sanz J., and Fuentes-Fernandez R., *The INGENIAS methodology and tools*, In Agent Oriented Methodologies (eds. Henderson-Sellers and Giorgini), Idea group, (2005), pp. 236-276.

[6] Huget M., and Demazeau Y., Evaluating multi agent systems: a record/replay approach, *Intelligent Agent Technology, IAT 2004, Proceedings IEEE/WIC/ACM International Conference*, (2004), pp. 536-539.

[7] Jennings N.R., An agent-based approach for building complex software systems, *Communications of the ACM*, (44) 4 (2001), pp. 35-41.

[8] Rouff C., *A Test Agent for Testing Agents and Their Communities*, IEEE, 2002.

[9] Sommerville I., *Software Engineering*, 9th edition, Addison Wesley, 2011.

[10] Zhang Z., Thangarajah J., and Padgham L., Automated unit testing for agent systems, *2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering, ENASE'07*, Spain, (2007), pp. 10-18.

[11] Ekinci E., Tiryaki M., Cetin O., and Dikenelli O., Goal-oriented agent testing revisited, *Proceeding of the 9th International Workshop on Agent-Oriented Software Engineering*, (2008), pp. 85-96.

[12] Caire G., Cossentino M., Negri A., and Poggi A., Multi-agent systems implementation and testing, *Proceedings of the 7th European Meeting on Cybernetics and Systems Research – EMCSR2004*, Vienna, Austrian Society for Cybernetic Studies, (2004), pp. 14-16.

[13] Lam D., and Barber K., Debugging agent behavior in an implemented agent system, *2nd International Workshop, ProMAS, Springer, Berlin*, (2005), pp. 104-125.

[14] Nunez M., Rodriguez I., and Rubio F., Specification and testing of autonomous agents in e-commerce systems, *Software Testing, Verification and Reliability*, (15) 4 (2005), pp. 211-233.

[15] Coelho R., Kulesza U., Staa A., and Lucena C., Unit testing in multi-agent systems using mock agents and aspects, *Proceedings of the international workshop on Software engineering for large-scale multi-agent systems*, ACM Press, New York, (2006), pp. 83-90.

[16] Gamma E., and Beck K., JUnit: a regression testing framework, http://www.junit.org, 2000.

[17] Tiryaki A.M., Oztuna S., Dikenelli O., and Erdur Sunit R., A unit testing framework for test driven development of multi-agent systems, *In 7th International Workshop on Agent Oriented Software Engineering*, 2006.

[18] Dikenelli O., Erdur R., and Gumus O., Seagent: a platform for developing semantic web based multi agent systems, *AAMAS'05 Proceedings of the fourth International Joint Conference on Autonomous agents and multi-agent systems*, ACM Press, New York, (2005), pp. 1271-1272.

[19] Gomez-Sanz J., Botia J., Serrano E., and Pavon J., Testing and debugging of MAS interactions with INGENIAS, *Agent-Oriented Software Engineering IX*, Springer, Berlin, (2009), pp. 199-212.

[20] Houhamdi Z., Test suite generation process for agent testing, *Indian Journal of Computer Science and Engineering*, (2) 2 (2011).

[21] Knublauch H., Extreme programming of multi-agent systems, *International Joint Conference on Autonomous Agent and Multi-Agent Systems*, Bologna. ACM Press, (2002), pp. 704-711.

[22] Gamma E., and Beck K., JUnit: a regression testing framework, http://www.junit.org, 2000.

[23] Botia J., Lopez-Acosta A., and Skarmeta G., ACLAnalyser: a tool for debugging multi-agent systems, *Proceeding of the 16th European Conference on Artificial Intelligence*, IOS Press, (2004), pp. 967-968.

[24] TILAB, Java agent development framework, http://jade.tilab.com/. Accessed on 17th May 2011.

[25] Padgham L., Winikoff M., and Poutakidis D., Adding debugging support to the Prometheus methodology, *Engineering Applications of Artificial Intelligence*, (18) 2 (2005), pp. 173-190.

[26] Rodrigues L., Carvalho G., Barros P., and Lucena C., Towards an integration test architecture for open MAS, *1st Workshop on Software Engineering for Agent-Oriented Systems/SBES*, (2005), pp. 60-66.

[27] Nguyen C., Perini A., and Tonella P., Goal-oriented testing for MAS, *Agent-Oriented Software Engineering VIII, Lecture Notes in*

Computer Science, (4951) (2008), pp. 58-72.

[28] Houhamdi Z., and Athamena B., Structured system test suite generation process for multi-agent system, *International Journal on Computer Science and Engineering*, (3) 4 (2011), pp.1681-1688.

[29] Foundations for Intelligent Physical Agents, FIPA-specifications. http://www.fipa.org/ specifications. Accessed on 29th Nov, 2011.

[30] Mills H. D., Dyer M. D., and Linger R. C., Cleanroom software engineering, *IEEE Software*, (4) 5 (1987), pp. 19-25.

[31] Fosse P. Thevenod and Waeselynck H., Statemate: applied to statistical software testing, *In Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA)*, (June 1993), pp. 78-81.

[32] DeMillo R. A., Lipton R. J., and Sayward F. G., Hints on test data selection: help for the practicing programmer, *IEEE Computer*, (11) 4 (1978), pp. 34-41.

[33] Hamlet R. G., Testing programs with the aid of a compiler, *IEEE Transactions on Software Engineering*, (3) 4 (1977), pp. 279-290.

[34] Wegener J., *Stochastic algorithms: foundations and applications*, *In Evolutionary Testing Techniques, Springer Berlin, Heidelberg*, Chapter 9, 2005, pp. 82-94.

[35] McMinn P., and Holcombe M., The state problem for evolutionary testing, *Proceedings of the International Conference on Genetic and Evolutionary Computation*, Springer, Berlin, (2003), pp. 2488-2498.

Dr. Ashok Kumar *is working as Professor and Chairman at Department of Computer Science and Applications, Kurukshetra University, Kurukshetra (India). He is PhD in Computer Science with vast teaching and research experience of more than 30 years.*

He had supervised more than 35 PhD scholars in different computing areas including operation research, software engineering, testing and designing etc.

Mr. Vinay Goyal is working as Asst. professor and Head of Department (MCA) at Panipat Institute of Engineering and Technology, Panipat (India). He had published 6 research papers in International Journals of repute on the topic Agent Oriented Software Engineering.

Besides this he had conducted an international conference with the title ICACCT 2008 at APIIT SD INDIA, Panipat (India) in Nov 2008.