TESTING MULTI-AGENT SYSTEMS FOR DEADLOCK DETECTION BASED ON UML MODELS

Nariman Mani *Vahid Garousi* *Behrouz H. Far*

Department of Electrical and Computer Engineering
Schulich School of Engineering, University of Calgary, Canada
{nmani, vgarousi, far}@ucalgary.ca

ABSTRACT

There is a growing demand for Multi-Agent Systems (MAS) in the software industry. The autonomous agent interaction in a dynamic software environment can potentially lead to runtime behavioral failures including deadlock. In order to bring MAS to the main stream of commercial software development, the behavior of MAS must be tested and monitored against the risk of unwanted emergent behaviors including deadlocks. In this paper, (1) we introduce a method for preparing test requirements for testing MAS; and (2) we deploy a MAS monitoring method for deadlock detection in MAS under test. The first method helps create test requirements using a resource requirement table from the MAS analysis and design. The second method monitors the MAS behavior to detect deadlocks at the run-time. Also, as model based software techniques such as Multi-agent Software Engineering (MaSE) are gaining more popularity; these model based approaches can help MAS developers to reduce the risk of having unwanted emergent behaviors such as deadlock in MAS.

Index Terms— Multi-agent system, Software testing, Deadlock detection, UML.

1. INTRODUCTION

Increasing demand for applications which can communicate and exchange information to solve problems collaboratively has led to the growth of distributed software architecture consisting of several interoperable software systems. One of the main difficulties of interoperable software systems is heterogeneity. Heterogeneity reflects the fact that the services offered by constructed components are independent from the designers and the design methodology [1]. Different programs written in different languages by different programmers must operate in a dynamic software environment. Agent based software engineering is one of the approaches devised to handle collaboration and interoperability. An autonomous agent is a computational entity that can perceive, reason, act, and communicate [2]. Multi-Agent Systems (MAS) consists of autonomous agents that try to achieve their goals by interacting with each other by means of high level protocols and languages [1]. However, the agent interaction can potentially lead to runtime behavioral failures including deadlock. Thus, testing and monitoring MAS to eliminate the risk of unwanted emergent behaviors, such as deadlock, is an essential precondition for bringing MAS to the main stream of commercial software. Also, as model-based software development practices are gaining more popularity [3], more and more MAS are developed using model-based practices such as the Multi-agent Software Engineering (MaSE)[4]. Thus, model-based testing techniques for deadlock detection for MAS can be useful since they can help MAS engineers to eliminate the risks of deadlocks in the MAS development.

In this paper we focus on proposing a methodology for testing MAS by preparing test requirements for deadlock detection. The artifacts used are the models prepared during the analysis and design stages of a MAS using the MaSE methodology[4] . Figure 1 illustrates the approach. Using the procedure explained in Section 5.1, resource requirement table is constructed based on Control Flow Paths (CFP) extracted from the MaSE task diagrams. The resource requirement table is used for searching for potential deadlocks (Section 5.2). Test requirements for testing MAS are prepared based on the potential deadlocks. The test requirements are used to generate the test cases. For deadlock detection on MAS under test we deploy our MAS monitoring methodology in [5]. Using the procedure explained in Section 4, a MAS behavioral model, consists of UML sequence diagrams, is constructed using MaSE analysis and design artifacts such as “role sequence diagram”, “agent class diagram” and “task diagram”. Two deadlock detection techniques, introduced in Section 6, are instrumented into the MAS under test’s source code. Test driver executes the test cases on MAS under test and runtime deadlocks are detected using the MAS behavioral model [5].

The remainder of this paper is structured as follows. The related works and background are described in Section 2. The MAS metamodel is introduced in Section 3. Constructing MAS behavioral model based on the MaSE is

discussed in Section 4. Test requirement preparation is described in Section 5. MAS monitoring for deadlock detection is explained in Section 6. Finally conclusions and future work are given in Section 7. An illustrated example is used to explain the methodology in the subsequent sections.

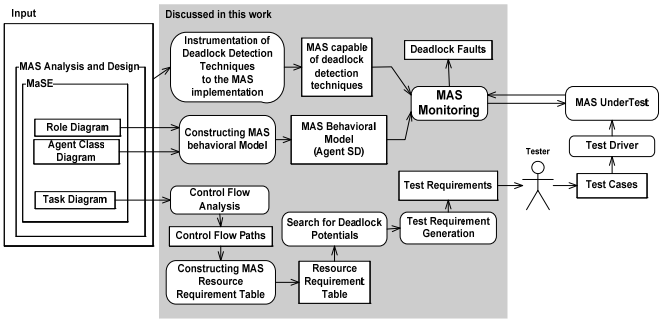

Figure 1 – An overview of our approach

2. RELATED WORKS AND BACKGROUND

2.1 MAS Verification and Monitoring

Existing works on MAS verification are categorized into axiomatic and model checking approaches [2]. In [6], axiomatic verification is applied to the Beliefs, Desires and Intentions (BDI) model of MAS using a concurrent temporal logic programming language. However, it was noticed that this kind of verification cannot be applied when the BDI principles are implemented with non-logic based languages [2]. Also in design by contract [7] pre- and post-conditions and invariants for the methods or procedures of the code are defined and verified in runtime. Violating any of them raises an exception. But as it is also claimed in [2] the problem is that this technique does not check program correctness, it just informs that a contract has been violated at runtime.

Model checking approaches seem to be more acceptable by industry, because of less complexity and better traceability as compared to axiomatic. Automatic verification of multi-agent conversations [8] and model checking MAS with MABLE programming language [9] are a few examples of model checking approaches that both use SPIN model checker [10], a verification system for detection of faults in the design models of software systems.

2.2 Deadlock Detection Techniques

Resource and communication deadlocks models are considered in message communication systems. Most deadlock models in distributed systems are resource models [11-13]. In these models, the competition is on acquiring required resources and deadlock happens whenever an entity is waiting permanently for a resource which is held by another. As indicated in [13], the communication deadlock model is general and can be applied to any message communication system. The communication model is an abstract description of a network of entities which communicate via messages. A deadlock detection mechanism based on the communication model deadlock for distributes systems and operating systems is provided in [13]. In literature a deadlock situation is usually defined as "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause" [14]. There are four conditions that are required for a deadlock to occur [14]. They are (1) "Mutual Exclusion" which means each resource can only be assigned to exactly one process; (2) "Hold and Wait" in which processes can hold resources and request more; (3) "No Preemption" which means resources cannot be forcibly removed from a process; and (4) "Circular Wait" which means there must be a circular chain of processes, each waiting for a resource held by the next member in the chain [14]. Similar to other types of the faults there are four techniques commonly used to deal with deadlock problem: ignorance, detection, prevention, and avoidance [14].

2.3 Agent Based Development Methodology: MaSE

MaSE uses several models and diagrams driven from the standard Unified Modeling Language (UML) to describe the architecture-independent structure of agents and their interactions [4]. In MaSE a MAS is viewed as a high level abstraction of object oriented design of software where the agents are specialized objects that cooperate with each other via conversation instead of calling methods and procedures. There are two major phases in MaSE: analysis and design (Table 1). In analysis phase, there are three steps which are capturing goals, applying use cases and refining goals. In the design phase, there are four steps which are creating agent classes, constructing conversations, assembling agent classes and system design[4].

Table 1- MaSE methodology phases and steps [4]

MaSE Phases and Steps	*Associated Models*
1. Analysis Phase	
a. Capturing Goals	Goal Hierarchy
b. Applying Use Cases	Use Cases, Sequence Diagrams
c. Refining Roles	Concurrent task, Role Diagram
2. Design Phase	
a. Creating Agent Classes	Agent Class Diagrams
b. Constructing Conversations	Conversation Diagrams
c. Assembling Agent Classes	Agent Architecture Diagrams
d. System Design	Deployment Diagrams

3. MAS METAMODEL

Figure 2 shows a metamodel for the MAS structure. In this figure, each MAS can be presented by MAS behavioral model in terms of sequence diagrams which shows the conversations of several agents and the message exchanging among them. The way of constructing such kind of behavioral model from MaSE design and analysis diagrams is introduced in Section 4. Each MAS consists of several agents whose roles are the building blocks used to define agent's classes and capture system goals during the design

phase. Associated with each role are several tasks and each task can be presented by MaSE task diagram [4]. A task diagram in MaSE is a UML state machine diagram which details how the goal is accomplished in MAS and can be represented by a Control Flow Graph (CFG) [15, 16]. A CFG is a static representation of a program that represents all alternatives of control flow. For example, a cycle in a CFG implies iteration. In a CFG, control flow paths (CFPs), show the different paths a program may follow during its execution.

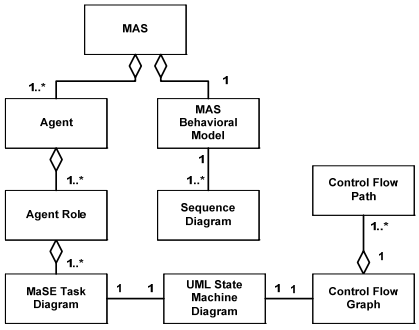

Figure 2- metamodel for MAS

4. CONSTRUCTING MAS BEHAVIORAL MODEL

Agents of a MAS communicate by exchanging messages. The sequence of messages is useful for understating the situation during faults detection conversation. A common type of interaction diagrams in UML is a sequence diagram in which each agent or role is represented by a lifeline in sequence diagram.

We deploy a method for transforming the conversations of agents from MaSE to UML sequence diagrams. These sequence diagrams are used in MAS monitoring method for deadlock detection in MAS under test [5]. The MAS sequence diagrams is not provided by MaSE per se and must be constructed using information provided by the MaSE artifacts such as role diagram and agent class diagrams [2].

The role sequence diagram in "Applying Use Cases" step in analysis phase of MaSE shows the conversations between roles assigned to each agent [4]. The agent class diagram is created in the "Constructing Agent Classes" step of MaSE represents the complete agent system organization consisting of agent classes and the high-level relationships among them. An agent class is a template for a type of agent with the system roles it plays. Multiple assignments of roles to an agent demonstrate the ability of agent to play assigned roles concurrently or sequentially. The agent class diagram in MaSE is similar to agent class diagram in object oriented design but the difference is that the agent classes are defined by roles, not by attributes and operations. Furthermore, relationships are conversations between agents [4]. Figure 3 shows examples of MaSE role sequence and agent class diagram.

The approach for constructing sequence diagrams based on the two above mentioned MaSE diagrams is defined as follow [5]. Each role sequence diagram is searched for the roles which are listed in the same agent class in the agent class diagram. Then, all of the roles in each role sequence diagram are categorized based on the agent which they belong to. Therefore, each category corresponds to an agent class in agent class diagram and the messages which it exchanges with other categories are recognizable. On the other hand, a new agent sequence diagram can be generated from agent class diagram which the lifelines are agents' types. The recognized messages between each two categories are entered into agent sequence diagram as a new conversation. For example, in Figure 3, the role sequence diagram 1 is categorized into three different categories, the first one consists of Role 1 and Role 2 and the second one consists of Role 3 and Role 4 and the last one consists of Role 5. The first one corresponds to agent class 1, the second one corresponds into agent class 2, and the third one corresponds to agent class 3. The constructed agent sequence diagrams from role sequence diagram 1, 2 and 3 and agent class diagram in Figure 3 are shown in Figure 4.

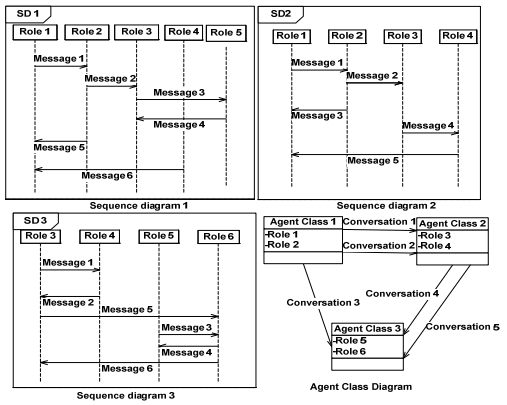

Figure 3- MaSE role sequence and agent class diagrams

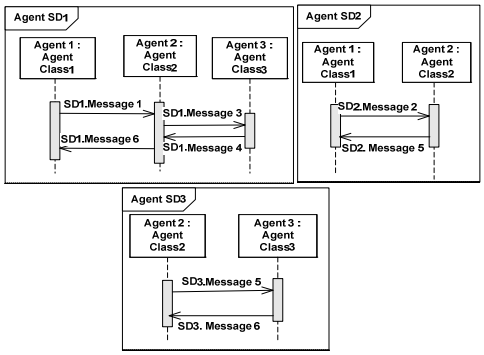

Figure 4- Constructed agent sequence diagrams

UML provides ways to model the behavior of an object oriented system using different types of diagrams such as state machine diagram. UML's state machine diagram is based on finite state machines (FSM) augmented with the concepts such as hierarchical and concurrent structure on

states and the communications mechanism through events transitions[3]. UML's state machine diagram is commonly used to describe the behavior of an object by specifying its response to the events triggered by the object itself or its external environment. State machine diagram has long been used as a basis for generating test data [15-17]. In MaSE [4], roles are the building blocks used to define agent's classes and capture system goals during the design phase. Every goal is associated with a role and every role is played by an agent class. Role definitions are captured in a role model diagram which includes information on communications between roles, the goals associated with each role, the set of tasks associated with each role, and interactions between role tasks. In MaSE, a task is a structured set of communications and activities, represented by a state machine diagram [4]. The MaSE's task diagram is then converted to UML's state machine diagram by converting some MaSE's task diagram notations such as the protocol transition, choices, and junctions to the UML notation. Using the state machine diagram, the CFG and its associated CFPs can be identified [15, 16].

5. TEST REQUIREMENT PREPARATION

In this section we focus on proposing a methodology for testing MAS by preparing test requirements for deadlock detection. Test requirements are generated using resource requirement table defined in Section 5.1.The resource requirement table is used in search for deadlock potentials (Section 5.2). The results from search for deadlock potentials are used for test requirement generation (Section 5.3). The test requirements are used by testers to generate the test cases for deadlock detection in MAS.

5.1 Resource Requirement Table for Agents

As discussed in Section 4, the behavior of each agent can be presented by the several MaSE task diagrams each reflecting a task assigned to a specific role of an agent. Each task consists of several CFPs that represent the different runs of the MaSE task diagram represented by UML state machine diagram. During the execution of each CFP, several resources are held and acquired by an agent. We define resource requirement table for each agent which shows the resource requirement for different tasks which are assigned to different roles of an agent (see Figure 5). Each row in resource requirement table shows the Required Resource Set (RS_{ij}) during execution steps of a specific CFP_i. If the required resources needed by a particular CFP_i are changed during its execution, a new set of the required resources on that stage is added to the resource requirement table for that CFP. Each column in resource requirement table represents the Sequence of Required resource Sets (SRS_i) by one CFP_i . We present the SRS formal definition as below:

$$SRS_i = < RS_{ij} | RS_{ij} \text{ is the } jth \text{ required resource set of the } CFP_i >$$

And

$$RS_{ij} = \{ R_p | R_p \text{ is a required resource by } CFP_i \}$$

The metamodel in Figure 6 depicts the definition of resource requirement tables and its elements.

	Task 1			...		...	Task n			
$Agent_i$	$CFP_{1,1}$	$CFP_{1,2}$	$CFP_{1,3}$	...	$CFP_{1,n}$	...	$CFP_{k,1}$	$CFP_{k,2}$	...	$CFP_{k,n}$
	$\{R_1\}$									
	$\{R_1,R_2\}$									
	$\{R_3\}$									
	$\{R_4\}$									
	$\{R_4,R_5\}$									
	⋮									
	$\{R_3,R_{10}\}$									

Figure 5- An example for Resource Requirement Table for Agent

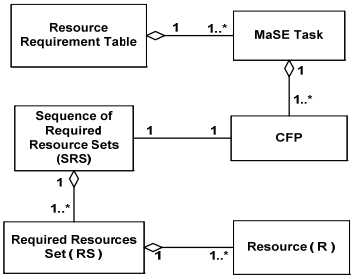

Figure 6- Resource requirement table metamodel

5.2 Search for Potential Deadlocks

In order to prepare the test requirement for deadlock detection between the CFPs, we first describe a scenario in which a deadlock happens. Figure 7 shows an example of resource allocations and resource requests (wait-for graph [18]) in deadlock situation.

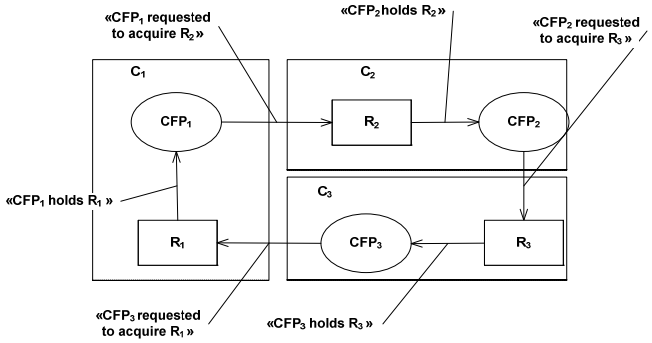

Figure 7- Resource allocations and requirements in deadlock situation (wait-for graph)

For explanation simplicity, we consider a situation that the RS set for each CFP has only one member .Each CFP holds one and request for acquiring the next required resource. The required resource may have already been acquired by another resource and the requestor has to wait for that resource. In resource model, CFP_1 is said to be dependent on another CFP_k if there exists a sequence of CFPs such as $CFP_1, CFP_2, \ldots, CFP_k$ where each CFP in sequence is idle and each CFP in sequence except the first one holds a resource for which the previous CFP is waiting.

If CFP_1 is dependent on CFP_k, then CFP_1 has to remain in idle status as long as CFP_k is idle. CFP_1 is deadlocked if it is dependent on itself or on a CFP which is dependent on itself. The deadlock can be extended to a cycle of idle CFPs, each dependent on the next in the cycle. Therefore, the deadlock detection approach is to declare the existence of that cycle.

The information for each SRS_i for each CFP_i can be retrieved from the resource requirement table defined in Section 5.1. In the Figure 7 wait-for graph [18], each resource set (each resource set has just one member in this example) in the cycle will be in sequence of required resource sets (SRS) of two CFPs. One CFP is holding the resource set and the other one requesting for acquiring it. As an example, $\{R_1\}$ is required by both CFP_1 and CFP_3 as it is shown below:

$$\{R_1\} \in SRS_1 =< \{\mathrm{R}_1\},\{\mathrm{R}_2\} > \; and \; SRS_3 =< \{\mathrm{R}_3\},\{\mathrm{R}_1\} >$$

The procedure of finding potential deadlocks in the behavioral model of MAS is defined as follow. The sequence of required resources set by a CFP_i (SRS_i) is retrieved for the all the CFPs in the MAS from the resource requirement table (Section 5.1). For each CFP, we assume that it is holding one of its required resource sets (RS_{ij}). $RS_{i,j}$ represents the jth required resource set of the SRS_i. Then, the next required resource set by that CFP, $Next_i$ is identified using the SRS_i for CFP_i.We search inside the SRSs for other CFPs that require at least one resource from $Next_i$ and assume in the worst case, they are holding it. We repeat this procedure until we find one CFP requiring a resource which is held by the CFP that we have already traversed by our procedure. In this case a deadlock cycle is detected. We consider this cycle as a potential deadlock cycle.

We explain the procedure with an example shown in Figure 8. For explanation simplicity, we consider a situation that the RS set for each CFP has only one member .We start the procedure from CFP_1 and assume that it holds its first resource set $\{R_1\}$. The next required resource set by CFP_1 is $\{R_2\}$. We assign the next required resource set by CFP_1 , $Next_1$ as $\{R_2\}$ and search the CFPs which has at least one resource from $Next_1$ (in this case just $\{R_2\}$) as the required resource in their sequence of required resources sets SRS_i. CFP_2 is found and it is assumed that in the worst situation it holds R_2. So, if CFP_2 holds R_2, the next resource set required by CFP_2 is $\{R_3\}$ according to the SRS_2. The search is started again for finding the CFPs which require R_3 as the required resource. CFP_3 is found and it is assumed that it holds R_3 . So, the next resource set required by CFP_3 if it holds R_6 is $\{R_1\}$. We find out that R_1 has been already assumed to be held by CFP_1 when we wanted to start the procedure. Therefore, a deadlock potential cycle is detected.

The pseudocode of searching for the potential deadlock cycles is shown in Figure 10. In finding the potential cycles function we define potential deadlock cycle data structure $PotentialDeadlockCycle_r$ which illustrates r-th potential deadlock in the MAS as below:

$$PotentialDeadlockCycle_r = \{(CFP_{i,}, RS_{p,}RS_q) \mid CFP_i \text{ is holding required resource set } RS_p \text{ and requesting for acquiring required resource set } RS_q\}$$

So, bases on the explained procedure the potential deadlock cycle, $PotentialDeadlockCycle_1$ for the example provided in Figure 8 is created as follow:

$$PotentialDeadlockCycle_1 = \{(CFP_{1,}, \{R_1\}, \{R_2\}), (CFP_{2,}, \{R_2\}, \{R_3\}), (CFP_{3,}, \{R_3\}, \{R_1\}) \}$$

These potential deadlocks which are found by the explained procedure are used for test requirement generation in the Section 5.3.

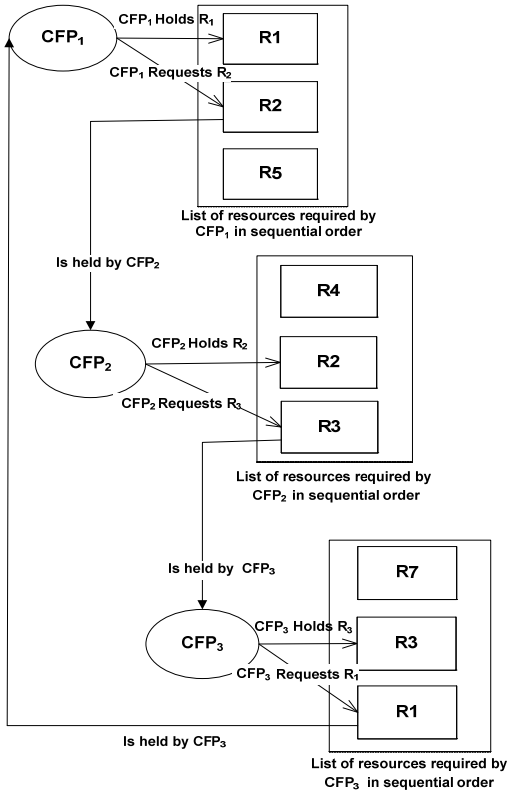

Figure 8- An example of finding deadlock potentials in the behavioral model of MAS

5.3 Test Requirement

We define the test requirement metamodel for testing MAS for deadlock detection and for each deadlock cycle in Figure 9. As it can be seen in Figure 9 test requirement for deadlock detection for a single deadlock cycle is divided into two parts. The first part is the Hold Set (HS) which represents the resource holdings in the MAS and is defined as below:

Hold Set: HS= $\{(CFP_i, RS_{p,})|\ CFP_i$ is holding required resource set $RS_{p,}\}$

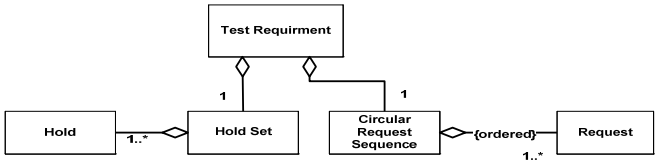

Figure 9 - Test requirement metamodel

The second part is the Circular Request Sequence (CRS) which shows the sequence of resource requests in the MAS and since it represents a cycle of requests we call it Circular Requests Sequence. It is defined as follow:

Circular Requests Sequence: CRS = $< Req_1, Req_2, \ldots, Req_n >$
$Req_i = (CFP_i, RS_q)|CFP_i$ requests for acquiring required resource set RS_q

In Figure 8 example, the test requirement based on the potential deadlock cycle ($PotentialDeadlockCycle_1$) found in the Section 5.2 is prepared as below:

HS= $\{(CFP_{1,}, \{R_1\}), (CFP_{2,}, \{R_2\}), (CFP_{3,}, \{R_3\})\}$
CRS = $<(CFP_{1,}, \{R_2\}), (CFP_{2,}, \{R_3\}), (CFP_{3,}, \{R_1\})>$

SoFarTraversed= $\{(CFP_{i,}, RS_{p,} RS_q)\ |\ CFP_i$ is holding required resource set RS_p and requesting for acquiring required resource set $RS_q\}$
FindPotentialDeadlocks(SRS_i, RS_{ij})
{
$Next_i$= NextRequiredResourceSet (SRS_i, RS_{ij})
Add $(CFP_{i,}, RS_{ij,} Next_i)$ to SoFarTraversed list
Add (CFP_i, RS_{ij}) to the HS set
If there is any CFP_j other than SoFarTraversed in MAS $Next_i$ is in its SRS_j
Then For each CFP_j in MAS that $Next_i$ is in its SRS_j
- Assume that each CFP_j holds $Next_i$
- Find the next required resource set for each of them ($Next_j$)
- Search in HS to see if the $Next_j$ that they require has been already assumed to be held
- If $Next_j$ exists in HS
  Then so add SoFarTraversed+$(CFP_{j,}, Next_i, Next_j)$ as a new $PotentialDeadlockCycle_r$, Return
  Else
  Call FindPotentialDeadlock (SRS_j, $Next_j$)
  End if

End For
Else
Return
End if
}
Main ()
{
For all SRS_i in MAS
For all $RS_{ij,}$ in each SRS_i
FindPotentialDeadlocks (SRS_i, RS_{ij})
}

Figure 10- A pseudo-code for searching potential deadlocks

5.4 Testing MAS for Deadlock Detection

The test requirement prepared in Section 5.3 is used by a tester to generate the test cases for deadlock detection. In each test case the hold (HS in test requirement) and request (CRS test requirement) situations should be created and the system is tested to check if deadlock happens. Generated test cases are executed using the test driver. In this step, a deadlock detection methodology for executing system at runtime is required to detect the deadlocks and report them as fault. Our monitoring method for deadlock detection [5] can be used as the deadlock detection methodology to monitor the system behavior at runtime to detect deadlocks. This methodology focuses on model based deadlock detection by checking MAS communication for existence of deadlock. During the next section (Section 6) the method and its application to our approach in this paper is presented.

6. MAS MONITORING FOR DEADLOCK DETECTION

MAS monitoring for deadlock detection [5] is a model based deadlock detection which checks MAS communication for existence of deadlocks. The artifacts used are the models prepared during the analysis and design stages of a MAS using the MaSE methodology[4]. An overview of monitoring approach is also illustrated Figure 1. In the MAS monitoring the source code of the system is instrumented with two deadlock detection techniques discussed in this section to enable runtime deadlock detection in MAS under test.

6.1 Deadlock Detection in Resource Deadlock Model

Resource model of MAS consists of agents, resources and controller agents. A controller agent is associated with a set of agents and a set of resources which it manages [5]. Agents request for acquiring resources from their controller. Also, the controller can communicate with other controllers in case of requesting resources from other controllers. In [5] a gateway agent is proposed as a translator of the controllers' communications. In, a communication protocol is defined for controller agents to communicate, acquire resources and handle behavioral faults such as deadlock.

Whenever agent A_a in controller C_i needs to acquire a resource R_i associated to another controller C_j, it sends its request to its controller C_i. C_i communicates with controller C_j regarding the requested resource R_i. If the required resource is available, C_j provides that resource for agent A_a in controller C_i .But if it is hold by another agent A_b, C_i provides the identification of agent A_b to controller C_i . So, each controller agent has information about the internal resource allocation inside its set and the external resources that each agent in its set has already acquired or wants to acquire.

In order to determine for an idle agent A_a whether it is in deadlock state or not, a communication is initiated by its controller agent. In deadlock detection communication, controller agents send an *investigator* message to each other. An *investigator* message is of the form

$Investigator(n,m,a,b,r,c)$ denoting that it is initiated by controller of agent A_a for process P_n and transaction T_m regarding agent A_b which requested to acquire resource R_r that it is currently held by A_c . It follows that if C_i receives $Investigator(n,m,a,b,r,a)$ from another controller for any possible b and r and if R_r is one of the resources which is held by A_a, a circular wait is detected and C_i declare A_a as deadlocked.

Figure 11 shows the message communication between controllers for deadlock detection for the wait-for graph scenario discussed in Figure 7 (Section 5). Agent A_1 is holding resource R_1 associated to its controller C_1 and requested to acquire R_2 associated to controller C_2. A_2 in C_2 is holding R_2 and requested to acquire R_3 from C_3. A_3 in C_3 is holding R_3 and requested to acquire to acquire R_3 which is held by A_1 . According to our assumptions for resource deadlock model, three of four deadlock conditions are true in this example which are (1) "Mutual Exclusion"; (2) "Hold and Wait"; (3) "No Preemption". Also as it can be seen in Figure 11 the fourth condition circular wait can be detected after receiving $Investigator(n,m,\ A_1,A_3,R_3,A_1)$ by C_1 and identifying that R_3 requested by A_3 and is held by A_1. So all the four condition of deadlock is true and C_1 can declare A_1 as deadlocked.

6.2 Deadlock Detection in Communication Deadlock Model

In the communication deadlock model of MAS there is no controller agent or resources. Associated with each idle agent is a set of dependent agents which is called its *dependency set*. Each agent in that set can change its state about one particular task from idle to executing upon receiving a message from one of the members of its dependency set regarding that task. We define a nonempty set of agents as deadlocked if all agents in that set are permanently idle regarding a special task. An agent is called permanently idle, if it never receive a message from its dependency set to change its state. In the more precise definition, a none empty set of agents *S* is called deadlock if and only if all agents in *S* are in idle state, the dependency set of every agent in *S* is a subset of *S* and there are no active conversation between agents in *S*. Based on this definition, all agents in the set can be called permanently idle, if the dependency set of each agent such as A_i is in *S*, so they are all in idle state, and also if there is not any trigger message in transit between agents just because there is no active conversation.

An idle agent can determine if it is deadlocked by initiating a deadlock detection conversation with its dependency set when it enters to idle state. An agent A_i is deadlock if and only if it initiates a query conversation to all the agents in its dependency set and receives reply to every query that it sent. The dependency set for each agent is identified using the MAS behavioral model constructed in Section 4. The purpose of initiating this query is to find out if the agent A_i belongs to a deadlock set S with the mentioned conditions above. On receiving a query by an idle agent in dependency set, it should forward the query to its dependency set if it has not done already.

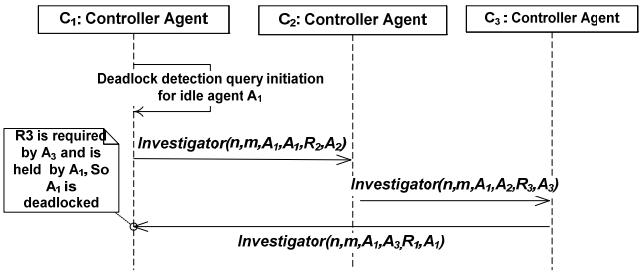

Figure 11 - deadlock detection example in resource deadlock model

Each query has the form $DeadlockDetectionQuery(n,m,i,a,b)$ denoting this message belongs to mth deadlock detection communication initiated for nth communication in MAS by agent A_i which is sent from agent A_a to agent A_b .Each agent A_k keeps the latest deadlock detection communication which it has been participated in it by $LatestDeadlockDetectComm(i)$ denoting the latest deadlock detection communication number that A_k was participated in it and initiated by A_i. The state of each agent (idle/executing) also is stored by $State(n,m,i)$ denoting the state of agent A_k for mth deadlock detection communication initiated by A_i for nth communication in MAS. Also the number of replies which should be received by an agent for mth deadlock detection communication initiated by A_i for nth communication in MAS is stored in $NumOfReplies(i)$.

We present the following scenario as an example for deadlock detection in the communication deadlock model of a hypothetical MAS with the sequence diagrams shown in Figure 4 (Section 4). In this scenario, agent A_1 is executing and has not received any message in one of its communications (in this case the communication in Agent SD 1) for a defined time *T* from its dependency set $\{A_2, A_3\}$. This is since A_2 and A_3 are both in waiting state and A_1 is not aware of it. After the defined time *T* , A_1 identifies itself as an idle agent and initiates a deadlock detection conversation with each agent in its dependency set. An agent A_1 declare itself as deadlocked if and only if it initiates a query conversation to all the agents in its dependency set and receives reply to every query that it had sent.

The complete deadlock detection scenario for the mentioned scenario is shown as a sequence diagram in Figure 12. A_1 initiates two query conversations with its dependency set which are A_2 and A_3 . A_2 and A_3 propagate the queries to their own dependency set which are $\{A_1, A_3\}$ for A_2 and $\{A_1, A_2\}$ for A_3 . Respectively A_2 and A_3 receive reply from their own dependency sets. Thus they both replies to A_1 which is the initiator of deadlock queries

in this scenario. So, A_1 receives reply for all quires which it had initiated to its dependency set. So, is declares itself as deadlocked.

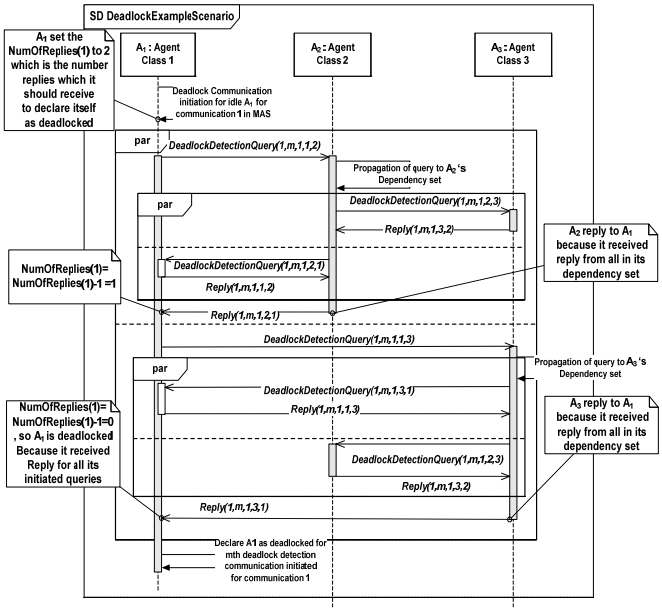

Figure 12 – A deadlock detection scenario in communication deadlock model using our technique

7. CONCLUSION AND FUTURE WORK

In this paper, we presented a methodology for test requirement generation and monitoring the behavior of MAS. The methodology is used to detect deadlocks as one of the unwanted emergent behaviors. Test requirements for deadlock detection are prepared using a resource requirement table. The test requirements are used by a tester in test case generation process. Test cases are executed using a test driver on the MAS under test. For deadlock detection in the MAS under test, a MAS monitoring methodology is proposed using our work in [5]. The source code of the system is instrumented with specific instructions in terms of deadlock detection techniques to enable runtime deadlock detection. As the future work, we plan to automate the test case generation process based on the test requirement for deadlock detection in MAS. Also, our methodology will be applied to a few more MAS case studies to evaluate its coverage, effectiveness and efficiency.

8 . REFERENCES

[1] M. R. Genesereth and P. K. Ketchpel, "Software agents " *Commun. ACM* vol. 37 pp. 48-53, 1994.

[2] F. Bergenti, M.P.Gleizes, and F. Zambonelli, *Methodologies and Software Engineering for Agent System* vol. 11. New York: Kluwer Academic Publishers, 2004.

[3] Object Management Group (OMG), "UML 2.1.1 Superstructure Specification," 2007.

[4] S. A. DeLoach, "The MaSE Methodology," in *Methodologies and Software Eng. for Agent System*. vol. 11, F. Bergenti, M.P.Gleizes, and F. Zambonelli, Eds. New York: Kluwer Academic Publishers, 2004, pp. 107-147.

[5] N. Mani, V. Garousi, and B. Far, "Monitoring Multi-agent Systems for Deadlock Detection Based on UML Models," in *IEEE CCECE 08 - Computer Systems and Applications*, 2008.

[6] M. J. Wooldridge and P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art," in *Proc. of the Workshop on Agent-Oriented Soft. Eng.*, 2000, pp. 1-28.

[7] B. Meyer, "Applying Design by Contract," *IEEE Computer,* vol. 25, pp. 40–51, 1992.

[8] H. L. Timothy and S. A. DeLoach, "Automatic Verification of Multiagent Conversations," in *the Annual Midwest Artificial Intelligence and Cognitive Science* Fayetteville, 2000.

[9] M. J. Wooldridge, M. Fisher, M. Huget, and S. Parsons, "Model Checking Multi-Agent Systems with MABLE," in *Proc. of the Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 2002, pp. 952–959.

[10] G. J. Holzmann, "The Model Checker Spin," *IEEE Trans. on Soft. Eng.,* vol. 23, pp. 279–295, 1997.

[11] V. Gligor and S. Shattuck, "Deadlock detection in distributed systems," *IEEE Trans. Soft. Eng.,* pp. 435-440, 1980.

[12] B. Goldman, "Deadlock detection in computer networks," Massachusetts Institute of Technology, Cambridge, Mass., Tech. Rep. MIT-LCS-TR185, 1977.

[13] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed Deadlock Detection," *ACM Trans. on Computer Systems,* vol. 1, pp. 144-156, 1983.

[14] A. Tanenbaum, *Modern Operating Systems*. Englewood Cliffs: Prentice Hall Inc., 1992.

[15] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. on Software Eng.,* vol. 4, pp. 178-187 1978.

[16] H. S. Hong, Y. G. Kim, S. D. Cha, D. H. Bae, and H. Ural, "A test sequence selection method for statecharts," *Software Testing, Verification and Reliability,* vol. 10, pp. 203 - 227, 2000.

[17] L. C. Briand, Y. Labiche, and Q. Lin, "Improving Statechart Testing Criteria Using Data Flow Information," in *Proc. of the 16th IEEE Int. Symposium on Software Reliability Engineering*, Washington, DC, USA, 2005, pp. 95 - 104

[18] D.A. Menasce and R.R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Transactions on Software Eng.,* vol. 5, pp. 195-202, 1979.