International Journal on Artificial Intelligence Tools
Vol. 19, No. 4 (2010) 417–437

DOI: 10.1142/S0218213010000261

World Scientific
www.worldscientific.com

SEARCH-BASED TESTING OF MULTI-AGENT MANUFACTURING SYSTEMS FOR DEADLOCKS BASED ON MODELS

NARIMAN MANI
Department of Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada
nmani@ucalgary.ca

VAHID GAROUSI
Department of Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada
vgarousi@ucalgary.ca

BEHROUZ H. FAR
Department of Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada
far@ucalgary.ca

Multi-Agent Systems (MAS) have been extensively used in the automation of manufacturing systems. However, similar to other distributed systems, autonomous agents' interaction in the Automated Manufacturing Systems (AMS) can potentially lead to runtime behavioral failures including deadlocks. Deadlocks can cause major financial consequences by negatively affecting the production cost and time. Although the deadlock monitoring techniques can prevent the harmful effects of deadlocks at runtime, but the testing techniques are able to detect design faults during the system design and development stages that can potentially lead to deadlock at runtime. In this paper, we propose a search based testing technique for deadlock detection in multi-agent manufacturing system based on the MAS design models. MAS design artifacts, constructed using Multi-agent Software Engineering (MaSE) methodology, are used for extracting test requirements for deadlock detection. As the case study, the proposed technique is applied to a multi-agent manufacturing system for verifying its effectiveness. A MAS simulator has been developed to simulate multi-agent manufacturing system behavior under test and the proposed testing technique has been implemented in a test requirement generator tool which creates test requirements based on the given design models.

Keywords: Multi-Agent Systems (MAS); Automated Manufacturing Systems (AMS); software testing; deadlock; Multi-agent Software Engineering (MaSE).

1. Introduction

Agent based software application is a system composed of multiple interacting intelligent agents that can perceive, reason, act, and communicate autonomously.[1] The agent based software engineering methodologies has been widely used in the automation of

manufacturing systems such as concurrent engineering, collaborative engineering design, manufacturing enterprise integration, supply chain management, manufacturing planning, scheduling and control, and material handling. An Automated Manufacturing System (AMS) is an integrated system of equipments and processes controlled via computer applications or a network of them that is capable of producing a variety of products with flexibility and efficiency.[2] A manufacturing system automated by agent based technology is composed of several autonomous and intelligent agents that can communicate and exchange information to manage the product line processes and solve challenging problems collaboratively.

A system composed of multiple interacting intelligent agents is called Multi-Agent System (MAS).[3] Therefore, in this paper we call a manufacturing system automated by means of several interacting agents as multi-agent manufacturing system. Similar to other distributed computing systems, multi-agent manufacturing systems are prone to the conflicts such as deadlock situations, wherein two or more competing agent actions are waiting for the other to finish, and thus neither ever does. In multi-agent manufacturing systems, the agents are responsible for managing the production jobs such as resource allocation to the production tasks. Considering machines and robots as the system resources in the multi-agent manufacturing systems, deadlock arises when resources are allocated to the production tasks in a way that makes task flow impossible. This can cause the major interruption to the manufacturing process by affecting the production cost and time.[4] The deadlock monitoring techniques similar to one that we have previously proposed in Ref. 5 are able to detect deadlock situations in a running system and warn the system user right before or upon occurrence for taking an appropriate action. But for reducing the risk of these unwanted emergent behaviors (i.e. deadlocks) at runtime the MAS environment should be tested for deadlocks while the system is under development process and before the release. If the testing comes late in the development process the changes can be expensive specifically if they are fundamental, such as architectural changes. But with the help of model based testing techniques the system behavior can be verified and tested during the early stages of development process while the system is under design.

They are many model-based approaches addressing deadlock problem in manufacturing systems. The key benefits of using models at runtime to detect deadlocks in manufacturing systems is that models can provide a rich semantic base for decision-making related to deadlock problem. However, to the best of our knowledge, none of the previously proposed strategies addresses the deadlock problem in the multi-agent manufacturing systems or in the other words manufacturing systems that have been automated by using agent based software engineering methodologies. In this paper we tackle the problem of deadlock in the multi-agent manufacturing systems analyzed and designed by means of Multi-agents Software Engineering (MaSE) methodology.[6] MaSE is one of the Agent Oriented Software Engineering (AOSE)[3] methodologies that uses several UML-like models and diagrams to describe the architecture-independent structure of agents and their interactions.

In this paper, we use the same technique that we used in Ref. 5 for extracting the potential deadlock information from the design models. But instead of using them for monitoring at runtime, we propose a metamotel for formatting them into test requirement data that can be used for testing for deadlocks. A test requirement generator tool has been also implemented which is able to extract the test data from the design artifacts in the proposed metamodel format. For verifying the efficiency of the testing technique in this paper, an experiment has been designed by simulating the system under test behavior. The experiment shows that the deadlock situation extracted by our testing technique from the design models can potentially appear in the running system (i.e. a released system) after passing a considerable amount of time from the beginning of the system operating. Therefore the testing techniques such as one proposed in this paper can be helpful identifying and resolving deadlocks faults in the system during the system design.

The remainder of this paper is structured as follows. The related work and background are described in Section 2. The proposed technique overview in this paper is discussed in Section 3. The input design model to our monitoring technique is introduced in Section 4. Constructing the machine requirement table used for finding the potential deadlocks is discussed in Section 5. An algorithm to extract potential deadlocks from the design models is described in Section 6. The proposed metamodel for test requirement is discussed in Section 7. The communication protocol for deadlock detection is discussed in Section 8. The case study and the experimental results are discussed in Section 9. Finally, Section 10 concludes the paper.

2. Background and Related Work

In this section, the background information required for describing the proposed technique in this paper is discussed in Sections 2.1 and 2.2. Then, we discuss the related work on deadlock detection in manufacturing systems and the techniques that they used in Section 2.3.

2.1. *Deadlock in automated manufacturing systems*

An automated manufacturing system usually consists of a set of cells, a material handling system connecting the cells, and service centers including material warehouse, tools room, and equipment repair. A cell can be either a machine, inspector, or a load/unload robot. Therefore, an automated manufacturing system can also be defined as a set of machines in which parts are automatically transported from one machine to another for processing. The deadlock problem has been already addressed in the Operating Systems (OS) and Distributed Database Systems (DDS) contexts. In those contexts, 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".[7] In this paper, we tackle the challenge of deadlock detection on resources which model physical objects (e.g. machines, robots, drives, etc.). In automated manufacturing system, those resources are shared by processes (i.e. agent tasks).

Coffman *et al.*[8] provided four conditions that must be held for a deadlock to occur: (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. In automated manufacturing system, the first three conditions given by Coffman *et al.*[8] are always satisfied. Agent tasks (i.e. manufacturing processes) use resources (i.e. machines) in an exclusive mode, they hold resources while waiting for the next resources specified by their operation sequence, and resources cannot be forcibly removed from the parts utilizing them until operation is completed. Therefore, a deadlock can only occur if the forth condition (i.e. circular wait) is held.[8] In this paper, we talk the problem of deadlock detection by finding the situations that can potentially lead to a circular wait. An example of system deadlock involving four machines is illustrated in Figure 1. In this figure, the moving parts (parts 1 and 2) are shown by white circles while the machines (machines A-D) are shown by grey rectangles. Part 1 on machine A has to reach machine D and part 2 on machine D has to reach machine A. A deadlock defiantly occurs between machines B and C, when the part 1 is moving to machine C and part 2 is moving to machine B.

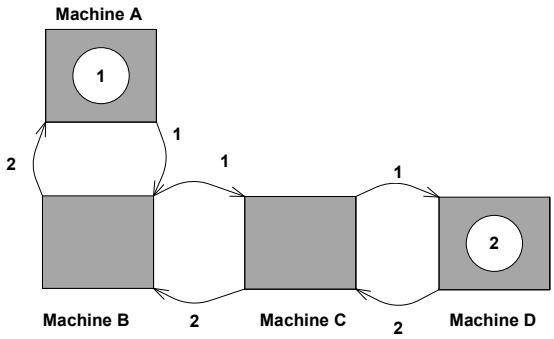

Fig. 1. A deadlock situation involving manufacturing machines and parts.

2.2. *Agent based development methodology*: *MaSE*

Artificial Intelligence (AI) techniques have significantly impacted manufacturing systems. Many automated manufacturing system application ranging from material handling and assembly controllers to long-term planning have been developed by means of MAS[9] technologies. As a result of the growing demand in MAS for industrial applications, many AOSE methodologies such as MaSE have been evolved to assist the development of agent-based applications.[6]

MaSE uses several models and diagrams driven from the standard UML[10] to describe the architecture-independent structure of agents and their interactions.[6] The main focus in MaSE is to guide a MAS engineer from an initial set of requirements through the

analysis, design and implementation of a working MAS. 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 and act proactively to accomplish individual and system-wide goals instead of calling methods and procedures. In the other words, MaSE builds upon logical object oriented techniques and deploys them in specifications and design of MAS. There are two major phases in MaSE: analysis and design (Table 1).

Table 1. MaSE methodology phases and steps.[6]

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	Task Diagram, 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

In this paper, we propose our deadlock detection technique by using the MaSE task diagrams as the input model. We propose that in the multi-agent manufacturing systems, MaSE task diagrams are used for modeling the defined production and assembly jobs. More details on MaSE task diagrams as the input to our monitoring technique and using them for deadlock detection in automated manufacturing systems designed by MaSE methodology are discussed in subsequent sections.

2.3. *Deadlock detection strategies for automated manufacturing systems*

Four main strategies addressing deadlock issues in manufacturing systems are prevention, detection, recovery, and avoidance methods. These strategies can be further categorized based on the model used to describe the AMS and, in particular, the interaction between jobs and resources. Three modeling methods are usually used: graph-theoretic, automata, and Petri nets (PN). The graph-theoretic approaches such as Ref. 11 are simple and intuitive solutions appropriate for describing the interactions between jobs and resources. This allows an efficient deadlock depiction even in complex Resource Allocation Systems (RAS) and allows the derivation of deadlock detection and avoidance strategies. Finite state automata can formally model the automated manufacturing systems behavior and have been used in establishing new deadlock avoidance control techniques such as Ref. 12. Petri nets (PN) also have been used extensively by researchers for modeling automated manufacturing systems (e.g. Ref. 13) and to develop suitable deadlock resolution methods (e.g. Ref. 14). But to the best of our knowledge, despite the growing demand of MAS applications in design, analysis, and development of automated

manufacturing systems,[9] there is no deadlock resolution or detection strategy based on the models created for multi-agent manufacturing systems based on the one of AOSE methodologies[3] such as MaSE.[6] In the following sections, we propose a testing technique for deadlock detection in multi-agent manufacturing systems based on the models constructed during MaSE methodology analysis phases.

3. The Technique Overview

An overview of the proposed technique in this paper is illustrated in Figure 2. As Figure 2 shows, the proposed technique has three main phases which are briefly described in this section of the paper.

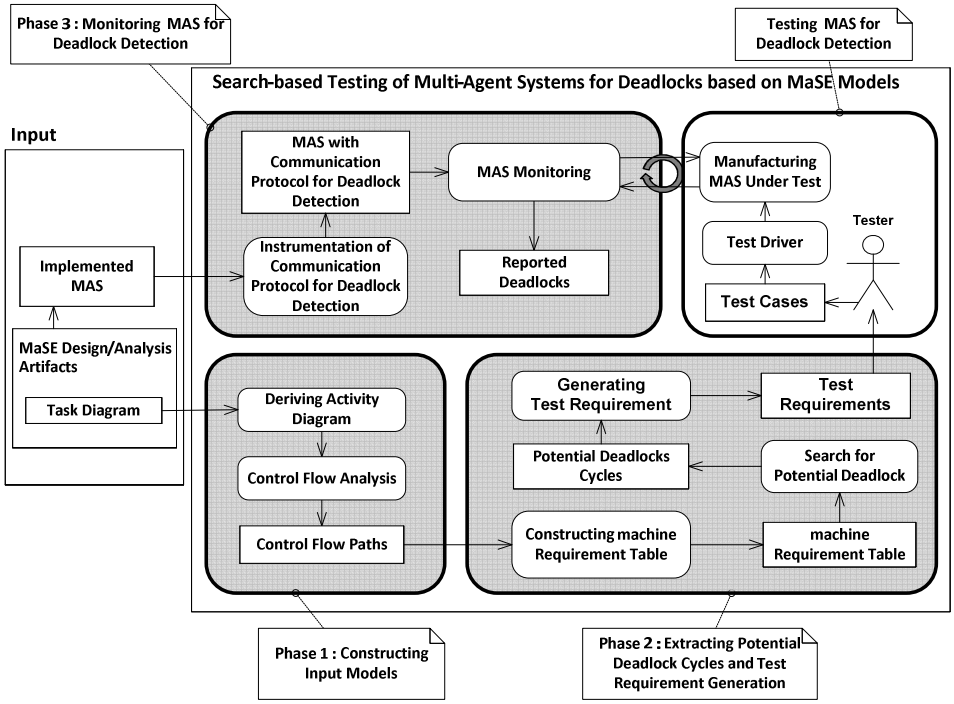

Fig. 2. Technique overview.

During the first two phase of the proposed technique (phase 1 and phase 2), the situations that can potentially lead to a deadlock at runtime (potential deadlocks) are extracted from the design models and formatted into test requirement data. The system tester uses this test data to create test cases and execute them using a test driver. But the deadlock situations forced on the system under test by means of test cases must be detected for verifying their existence. As a solution for this issue, in phase 3 we use a simplified version of our monitoring technique published in Ref. 5 which instruments the

implemented MAS with a communication protocol capable of detecting deadlocks occurred on system under test.

The input to the proposed deadlock monitoring technique is the agents' task diagrams built during the analysis and design of the MaSE methodology. These diagrams illustrate the activities performed by several cells (e.g. machines and robots) inside the multi-agent manufacturing system. During the steps in the first phase, each MaSE task diagram, a UML-like statechart diagram is converted to a UML activity diagram, from which Control Flow Paths (CFP) are derived afterwards (Section 4). Those CFPs are used in phase 2 for extracting the potential deadlock information by means of a dedicated search algorithm (Section 6). This extraction is performed by the "search for potential deadlock" unit. Then the potential deadlock information is formatted into test requirement data used by the tester for test case creation (Section 7). In phase 3, the implemented manufacturing MAS designed by MaSE methodology is instrumented with a deadlock detection communication protocol (Section 8). While the test driver is executing the test cases on the system under test, the communication protocol is able to monitor the entire MAS to verify the occurrence of the deadlocks.

4. The Input Task Model

In MaSE, a task is a structured set of communications and activities, represented by a UML-like state diagram which consists of states and transitions.[6] UML's state machine diagram is commonly used to describe the behaviour of an object by specifying its response to the events triggered by the object itself or its external environment.[10] However in MaSE,[6] this diagram is used to represent the behavior of a task associated to an agent. In a MaSE task diagram, states contain activities that represent internal reasoning, reading a percept from sensors, or performing actions via actuators. Multiple activities can be in a single state and are performed in an un-interruptible sequence. Once in a state, the task remains there until the activities' sequence is completed.[6] In a multi-agent manufacturing system, the task diagram is used for representing the flow of manufacturing activities. An example of a MaSE task diagram created for a specific task in a multi-agent manufacturing system is shown in Figure 3.

As the task diagram in Figure 3 shows, this task is able to perform its job through two alternative production lines (flow paths). When the task is initiated by the controller, the task finds the next available production line and afterwards makes the transportation robot to load the parts into that line. After performing each set of activities in each state, the task communicates with the controller agent through the "send" and "receive" messages and informs it about the task status and its current state.

In addition to the information regarding the flow of activities inside each task diagram, the task diagram has the information regarding the sequence of resources (i.e. machines and robots) that are required by the task during the execution of its activities. On the other hand, in this paper we tackle the problem of deadlock on the resources (i.e. machines and robots) shared by agents' tasks. Therefore, in the proposed technique

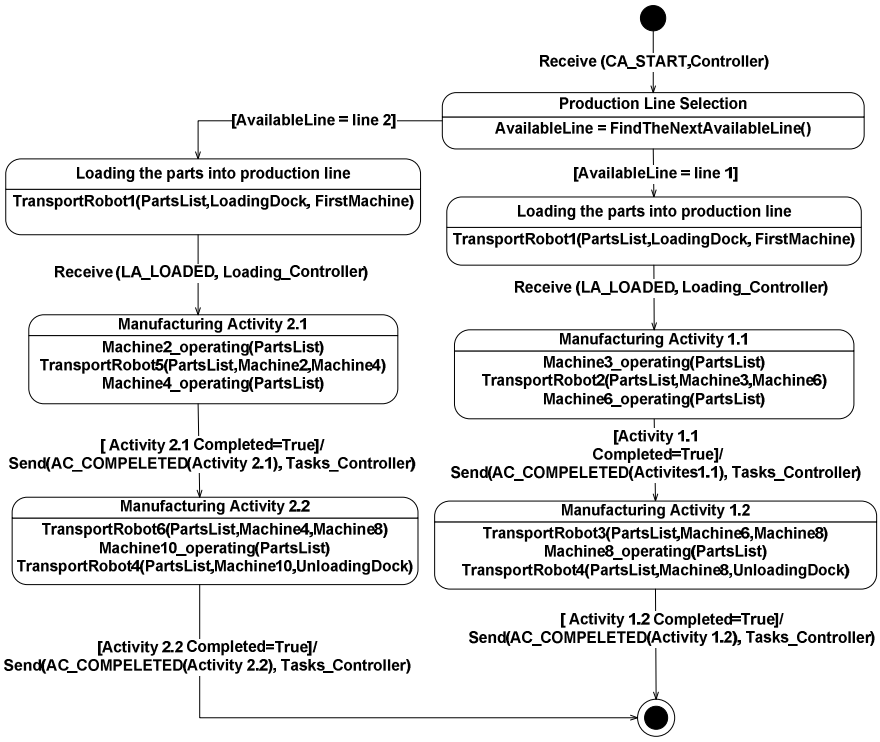

Fig. 3. An example input model: A MaSE task diagram for a manufacturing task.

in this paper, we use MaSE task diagrams for extracting the information regarding the required resources by each agent's task. This information is obtained during the steps in phase 1 of the proposed technique in this paper (See Figure 2) by converting the task diagrams into UML activity diagrams. Figure 4(a) shows the constructed activity diagram based on the task model provided in Figure 3. Activity diagrams have been in UML since its early 1.x versions and they are used to describe both sequential and concurrent control flow and data flow.[15] As it is mentioned in UML 2.0 (Section 12.1 of Ref. 10), the UML activity diagrams are commonly called Control Flow Graph (CFG). By analysis of the control flow paths (CFP) in the constructed CFG or activity diagram the resource requirement flow of the task can be extracted. The procedure is described in more details in the reminder of this section.

Each state in MaSE task diagram contains the activities representing internal behavior of the task when it is on that state. Since the order of task's states is presented in the MaSE task diagram by the directed transition arrows, the activities within the task, the sequence of their execution and their execution conditions can be easily driven from the states and the transition arrows and then they are imported into a new activity diagram.

Furthermore, the transition protocol (i.e. "send" and "receive" message communications) in MaSE task diagram uses the syntax of trigger [guard]/transmission and the trigger and

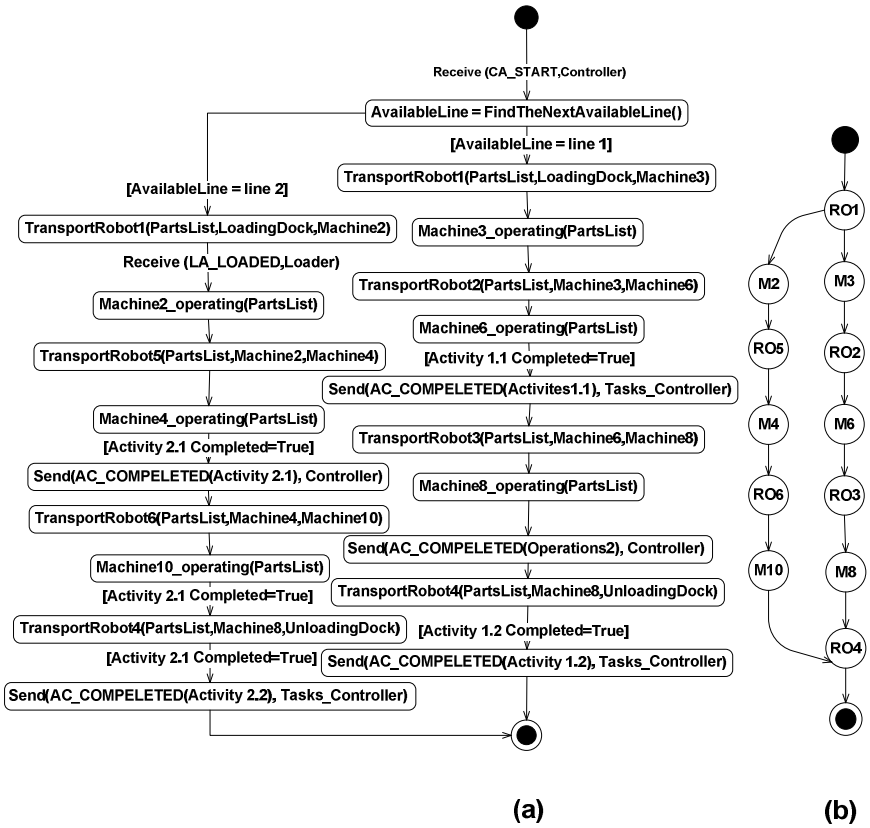

Fig. 4. (a) Constructed activity diagram based on the Task Diagram (b) Resource Requirement Flow Model.

transmission are limited to send and receive messages[6] (See Figure 3). Therefore, in the newly constructed activity diagram, each trigger and [guard] form the MaSE task diagram should be considered as condition of the transition from the activities in source state to the activities in destination state. Also, transmission should be considered as a new activity executed after all activities listed in the source state (See Figure 4(a)). As a result, using the derived activities from the states and the transitions among them, the corresponding activity diagram for a MaSE task diagram is created in Figure 4(a). Using the sequence of activities in the constructed activity diagram (i.e. CFPs), the resources used by each tasks activity can be extracted and imported into a Resource Requirement Flow Model (Figure 4(b)). In the Resource Requirement Flow Model presented in Figure 4(b), the resources (i.e. machines and transport robots) used during the execution of each CFP are presented. Each manufacturing machine is shown by Mi and each transport robot by ROi. Between the manufacturing machines and the transport robots resource types, only the manufacturing machines are selected and considered for tackling the deadlock problem in this paper. This selection is based on the assumption that the transport robots are only used for carrying the parts and they are not directly involved in

the manufacturing jobs such as assembly or modification. Therefore, they can be forcibly taken from the tasks by the others agents' requests and be involved in other tasks by providing them the storages that they can move the carrying parts into them temporarily. In this case, they do not satisfy the Coffman *et al.*[8] conditions for a deadlock situation and they must be ignored for the deadlock detection technique proposed in this paper.

5. Machine Requirements Table

We use the resource requirement information obtained from each CFP in the activity diagram to find the potential deadlocks inside the design models. But before using this information for extracting the deadlock information, we organize the gathered the resource requirement information from the activity diagram into a table called Machine Requirement Table. As discussed in Section 4, among all the resources types extracted from the input model, only the manufacturing machine are considered for the process of deadlock detection in this paper. Therefore, each column in machine requirements table represents the Sequence of Required Machines (SRM_i) by one CFP_i . Formally, The SRM is defined as below:

$$SRM_i = \langle M_j \mid M_j \text{ is the } jth \text{ required Machine of the } CFP_i \rangle \quad (1)$$

Figure 5 shows an example of a Machine Requirement table created for the case study in this paper. The first two columns shows the SRMs for the two CFPs extracted from the constructed activity diagram in Figure 4(a).

Agent #	$Agent_1$		$Agent_2$	$Agent_3$		$Agent_4$	
Task #	$Task_1$		$Task_3$	$Task_4$	$Task_5$	$Task_6$	$Task_7$
CFP #	CFP_1	CFP_2	CFP_3	CFP_4	CFP_5	CFP_6	CFP_7
Sequence of Required Machines (SRM)	M_3	M_2	M_1	M_{11}	M_2	M_5	M_{11}
	M_6	M_4	M_4	M_9	M_4	M_4	M_8
	M_8	M_{10}	M_7	M_7	M_3	M_3	
			M_{10}	M_5			

Fig. 5. Machine requirement table for manufacturing tasks.

6. Extracting the Potential Deadlocks

The constructed machine requirements table in previous section is used in this section for extracting potential deadlocks in the system. The deadlocks found from the design models in this step are called potential deadlock cycles since they can potentially lead to a deadlock situation at runtime. In this technique, a $Task_1$ is said to be dependent on another $Task_k$ if there exists a sequence of tasks such as $Task_1, Task_2, \ldots, Task_k$ where each task in sequence is idle and waiting for a resource held by the next task in the sequence. If $Task_1$ is dependent on $Task_k$, then $Task_1$ has to remain in idle status as

long as $Task_k$ is idle. $Task_1$ is deadlocked if it is dependent on itself or on a task which is dependent on itself. The deadlock can be extended to a cycle of idle tasks, each dependent on the next in the cycle. This is called a potential deadlock.

Our search technique uses the machine requirements table and searches for the tasks' combinations that can potentially lead to a deadlock cycle. These cycles and the participant agents in it are candidates for potential deadlock cycles and are used for forming the rest requirements (i.e. Section 7). For describing the procedure of the proposed search technique in this paper, an illustrated example is provided and shown in Figure 6.

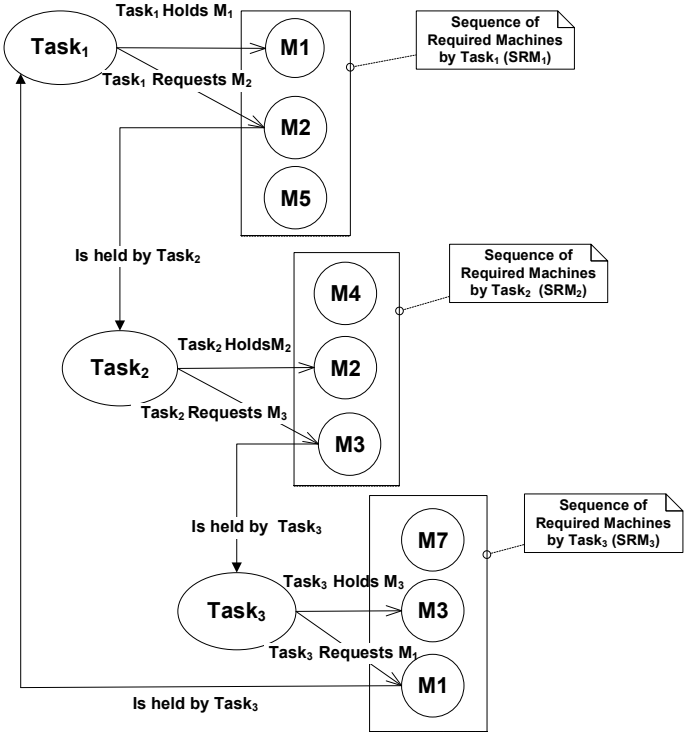

Fig. 6. An example of potential deadlock of tasks.

The provided example in Figure 6 shows three tasks of a multi-agent manufacturing system. Each task runs a CFP from its activity diagram (i.e. CFG). Associated to each CFP is a SRM (SRM_1, SRM_2, and SRM_3) representing the sequence of required resource by task on that CFP. The proposed search technique works as follow: starting from $Task_1$, the search technique assumes that it holds its first requested machine in the SRM_1 which is M_1 and adds M_1 to a SofarTraversed list (i.e. SofarTraversed = $\langle M_1 \rangle$). Therefore, the next machine that the $Task_1$ will request for it after holding M_1 is M_2. In this stage, the technique searches inside the other SRMs (SRM_1, SRM_2) to find out if there is any other

request for M_2 from the other tasks as well. It finds that $Task_2$ has M_2 as a requested machine in its SRM (SRM_2) and therefore it may compete with $Task_1$ for acquiring M_2. In this stage, the technique assumes that in the worst case scenario, the M_2 has been held by $Task_2$ and the Task1 has to wait for it and then add the M_2 to the SofarTraversed list (i.e. SofarTraversed = $\langle M_1, M_2 \rangle$). Therefore, if the $Task_2$ is holding M_2 then the next requested machine by $Task_2$ from SRM_2 will be M_3. In this stage, the technique searches the SRM_1 and SRM_2 to find any match for M_3. A match is found in SRM_3 and again the technique assumes that M_3 is held by $Task_3$ and $Task_2$ has to wait for it. M_3 is added to the SofarTraversed list (i.e. SofarTraversed = $\langle M_1, M_2, M_3 \rangle$). After M_3, the next requested machine by $Task_3$ is M_1. The technique finds a match for M_1 in SRM_1. The technique has to assume that the M_1 is held by $Task_1$ and $Task_3$ has to wait for it. But, the technique finds out that M_1 has been already added in to SofarTraversed and this means that the M_1 has been already traversed by our proposed technique. In this stage, technique finds a cycle of holds and requests which can potentially lead to a deadlock at runtime. The technique reports this cycle as a potential deadlock. This procedure is repeated by initiating the search from all other the SRMs and the resources inside them till the entire potential deadlocks are found. A possible pseudo-code for the technique discussed in this section is presented in Figure 7.

In Figure 7 pseudo-code, first the traversed SRM (SRM_i), held machine (M_i), and requested machine (M_{i+1}) are added into a thus far traversed list called *SofarTraversedList* (lines 3-5). Then, if the newly added items into the *SofarTraversedList* make a cycle with the existing items in it, the created *SofarTraversedList* is printed as a poetical deadlock cycle (lines 8-10). Otherwise, the requested machine (M_{i+1}) is searched in the all other SRMs except the existing SRMs in

```
1  private void Search_Potential_Deadlocks( List  SofarTraveresed , List SRM_i , Int  M_i ,Int  M_i+1)
2  {
3    List SofarTraversedList = SofarTraveresed ;
4     Next  =  M_i+1;
5     Add  (SRM_i , M_i , M_i+1 )  to SoFarTraversedList
6     if (M_i == Last Requested Set  in SRM_i)
7                    return;
8     for (int k = 0; k< SoFarTraversedList.Length-1 ; k++)
9          if (SoFarTraversedList [k]  == Next )
10               Print all the items in SoFarTraversedList as a Potential Deadlocks Cycle ;
11
12    for ( int p=0 ;All other SRMs (SRM_p) in the MAS except the SRM_i  ; p++)
13                if ( SRM_p does not exist in SofarTraversedList )
14                   if ( Next  exists in the SRM_p)
15                        {
16                        NextRequiredMachine = Find the required set next set to Next  in SRM_p;
17                        Found= true;
18                        Search_Potential_Deadlocks (SofarTraveresed , SRM_p , Next ,NextRequiredMachine  );
19                        }
20     if ( Found == false)
21                return;
22  }
```

Fig. 7. Pseudo-code for finding the potential deadlocks.

SofarTraversedList to find a match (lines 12-19). If any match found, the requested machine (M_{i+1} or *Next*) is assumed as a held machine and the next required machine in the SRM that the match is found in it is assumed as the requested machine (*NextRequiredMachine*) and the same procedure is called again till a cycle is found or all the SRM are traversed without any found match.

7. Test Requirement Metamodel

The extracted potential deadlock cycle in Section 6 are used for generating the test requirements. In Section 6 we described a deadlock cycle based on tasks which are running the different CFPs inside the system. As it's discussed in Section 5 (Figure 5), there are tasks associated to each agent in the system. Also, each task can have different CFPs associated to it. An example of deadlock cycle in agent point of view can be represented as below:

$$\begin{aligned}\text{Deadlock Cycle} &= \langle(\text{Agent}_1, \text{M}_4, \text{M}_7), (\text{Agent}_3, \text{M}_7, \text{M}_5), (\text{Agent}_4, \text{M}_5, \text{M}_4)\rangle \\ &= (\text{Agent}_3, \text{M}_7, \text{M}_5), (\text{Agent}_4, \text{M}_5, \text{M}_4), \langle(\text{Agent}_1, \text{M}_4, \text{M}_7)\rangle \\ &= \langle(\text{Agent}_4, \text{M}_5, \text{M}_4), (\text{Agent}_3, \text{M}_7, \text{M}_5), (\text{Agent}_1, \text{M}_4, \text{M}_7)\rangle\end{aligned}$$

For specifying the order of the items in the cycle, we identified the elements in $\langle\ \rangle$ notation. (Agent_1, M_4, M_7) means that Agent_1 is holding R_4 and the next resource (i.e. manufacturing machine) that it plans to take is R_7. Since a cycle can be started from any element inside it, the equivalent version of each cycle is indicated in the above example too. But presenting the deadlock cycles in this format does not give us the information about the CFPs and Tasks participated in the deadlock situation (i.e. as its discussed associated with each agent are different tasks and associated to each task are different CFPs). Therefore, in this section, we define a test requirement metamodel based on the involved CFPs information and since the relationship between agents, tasks, and CFPs are indicated in the machine requirement table (e.g. Figure 5) different presentations can be easily interpreted to each other. The test requirements contain the required information for the tester to create the test cases.

Each test requirement describes a potential deadlock situation by indicating the following information: (1) the resources held by each CFP at the time of deadlock, (2) the amount of time that each resource is supposed to be used, (3) resources requested (as the next required resource) by each CFP at the time of deadlock, (4) the start time of operation of each CFP based on the time that the whole system started working, and (5) the time that the deadlock occurs based on the amount of time passed from the start time of the CFP operation until deadlock happens. Based on the required information for describing a deadlock situation, we proposed the metamodel for a test requirement for deadlock in Figure 8.

As it is illustrated in Figure 8, each test requirement consists of: (1) a Held Set (HS) contains the resources held by each CFP at the time of deadlock and the amount of that

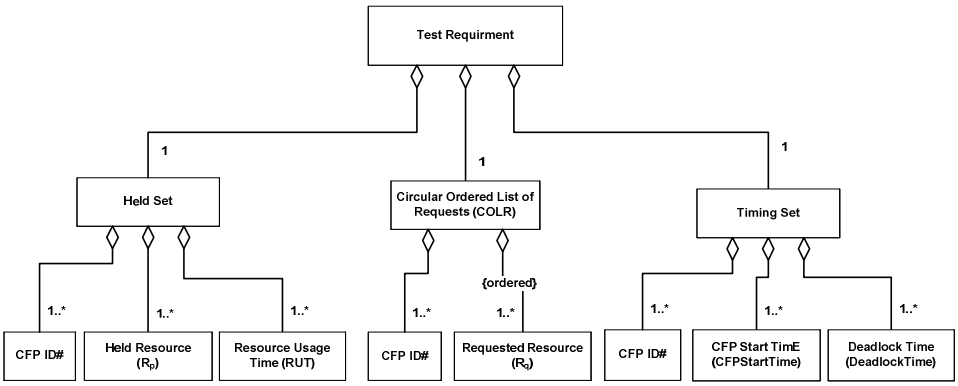

Fig. 8. Test requirement metamodel.

the resources were supposed to be used, (2) a Timing Set (TS) contains each CFP's start time of operation and the time that deadlock occurs, and (3) a Circular Ordered list of Requests (COLR) which contains cycle of the resource requests by each CFP at the time of deadlock in system.

The information for the Held Set (HS) and Circular Request Sequence (COLR) are extracted by the search algorithm discussed in Section 6. But the information in the Timing Set (TS) has to be calculated based on the existing data. The amount of time that a resource is supposed to be used by a specific CFP is indicated by the system designer in the MAS design and analysis specification. Therefore, in the timing set, the CFP start time of operation (i.e. the time that each CFP should start its operation to be participated in the deadlock situation) and the time that deadlock occurs can be calculated using following information: (1) the sequence of required resources by each CFP in the resource requirement table when the deadlock occurs, and (2) the amount of time that resource (machine) is supposed to be used by the CFP. The required data for the former is provided by the search algorithm (i.e. discussed in Section 6) and the latter one is provided by the system designer in the specification. The formal definition of each item in the test requirement is provided as below:

Held Set: HS= $\{(CFP_i, M_p, RUT_p) | CFP_i$ is holding required machine M_p for RUT_p (*Resource Usage Time*) Seconds$\}$

Circular Ordered List of Requests: COLR = $\langle Req_1, Req_2, \ldots, Req_n \rangle$
While $Req_i = (CFP_i, M\text{q}) | CFP_i$ requests for acquiring required machine set M_q)

Timing Set: TS = $\{(CFP_i, CFPStartTime_i, DeadlockTime_i) | CFP_i$ starts its operation after $CFPStartTime_i$, seconds from the start time of MAS and deadlock occurs after $DeadlockTime_i$ seconds from the start time of CFP operation$\}$

The test requirement should be able to present a deadlock situation by presenting each deadlock cycle. In this paper we specify this cycle by presenting the resource request in a circular ordered list format. As it is shown in above definitions, we use the $\langle Req_1, Req_2, \ldots, Req_n \rangle$ notation for presenting the circular ordered list which each *Req* is a

2-tuple. Two elements inside the *Req* respectively represent the CFP and the resource that it request. As an example, a test requirement generated for a deadlock cycle and its interpretation to the participated agents is shown below:

Formatted test requirement (indicating participated CFPs):

$$HS = \{(CFP_3, M_4, 2), (CFP_4, M_7, 2), (CFP_6, M_4, 2)\}$$
$$COLR = \langle(CFP_3, M_7), (CFP_4, M_5), \{(CFP_6, M_4)\rangle =$$
$$\langle(CFP_4, M_5), (CFP_6, M_4), (CFP_3, M_7),\rangle = \langle(CFP_6, M_4), (CFP_3, M_7), (CFP_4, M_5),\rangle$$
$$TimingSet = \{(CFP_3, 2,4), (CFP_4, 0, 6), (CFP_6, 4, 2)\}$$

Deadlock cycle (indicating participated agents):

$$\langle(Agent_1, M_4, M_7), (Agent_3, M_7, M_5), (Agent_4, M_5, M_4)\rangle$$
$$(Agent_3, M_7, M_5), (Agent_4, M_5, M_4), \langle(Agent_1, M_4, M_7)\rangle =$$
$$\langle(Agent_4, M_5, M_4), (Agent_3, M_7, M_5), (Agent_1, M_4, M_7)\rangle$$

In above example, HS shows the combination of the CFPs and held resources at the time of deadlock. For example, a triple such as (CFP_3, M_4, 2) in HS indicates that CFP_3 was holding M_4 and M_4 was supposed to be used by CFP_3 for 2 seconds. In this paper we indicate the CFP start time of operation based on the time that MAS starts its operation. Also, the time that the deadlock occurs shows the amount of time that passed from the start of CFP operation until deadlock happens. For example in the timing set (i.e."TimingSet"), (CFP_3, 2, 4) indicates that for being participated in a deadlock situation, CFP_3 has to start its operation 2 seconds after the MAS starts its operation and the deadlock occurs 4 seconds after CFP_3 is run by its agent.

8. A Communication Protocol for Deadlock Detection

The potential deadlock cycles extracted by the search technique represents the situation that can lead to a deadlock at runtime. They are formatted into test requirement data and used for creating test cases by system tester. The system tester executes the test cases which each forces a deadlock situation into the system. While the system is under test, a technique is required to detect the deadlocks occurs on system under the test. We proposed a runtime monitoring technique for deadlock detection in Ref. 5. The technique in Ref. 5 is a deadlock detection protocol which is able to use the deadlock knowledge extracted from the design models and propagate a deadlock detection query to the possibly participated agents in a deadlock situation to detect deadlocks at runtime.

In this paper, we use a simplified version of that technique which is basically just a deadlock detection query without any knowledge of the potential deadlocks and agents. The query is propagated to the all the agents in the systems by an agent whenever it is suspected to be involved in a deadlock situation. In this deadlock detection protocol, each agent is associated with a set of dependent agents called the "dependency set". Each agent identifies its dependency set based on the agents that it was recently in conversation with and it's expecting to receive replies for the requests that it has made to them. Whenever an agent is suspected to be involved in a deadlock situation (after spending a defined amount of time in an idle mode), it propagates a deadlock detection query to its

dependency set. Assuming each CFP is running by an agent in multi-agent manufacturing system, an agent in a dependency set can change its status from idle to active upon receiving any message from one of the other members of its dependency set. A nonempty set of agents are considered as deadlocked if all agents in that set are permanently idle. An agent is called permanently idle, if it never receive a message from its dependency set to change its status. An agent can determine if it is deadlocked by initiating a deadlock detection query messages to its dependency set when it enters the idle state. If it figures out that it won't receive any message from its dependency set to change its status, it declares itself as deadlocked (permanently idle). Upon receiving a query by an idle agent in dependency set, it forwards the query to its own dependency set too if it has not done already. Figure 9 shows an example of deadlock detection conversations among agents. The query conversation messages are shown by continues arrows and the replies are shown by dashed arrows. In this example, Agent 1 initiates two deadlock detection queries to its dependency set: Agent 2, Agent 3, and Agent 4. Agent 2 can reply to the query immediately by informing Agent 1 about its state since it is not involved in any other dependency set. Agent 3 and Agent 4 are involved in other dependency sets and they pass the query to their own dependency set before informing Agent 1 about their status. Agent 3 and Agent 4 answer the Agent 1's query upon receiving the replies from their dependency set. Based on the replies received, Agent 1 can identify whether its status can be changed from idle to active or not.

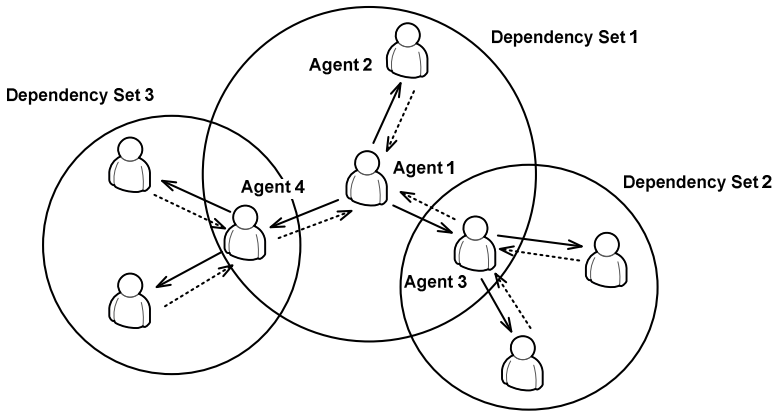

Fig. 9. Deadlock detection conversation.

9. Case Study and Experimental Results

We performed an experiment for showing the effectiveness of our technique based on a multi-agent manufacturing system with capability of running four different CFPs concurrently by four agents at each time. Figure 10 shows a snapshot of the tool implemented based on the search algorithm discussed in Section 6 and test requirement metamodel in Section 7. The tool user interface is divided to three different panes. The

machine requirement table constructed for the case study in this paper (i.e. discussed in Section 5) is used as an input for this tool (Pane 1 in Figure 10). The system tester is able to assign CFPs to each of dedicated four agents in the system and presses search for potential deadlock button (Pane 2 in Figure 10). Then the tool applies the proposed algorithm in Section 6 to the assigned CFPs and finds the potential deadlocks. The tool also interprets the deadlock cycles into the test requirements format proposed in Section 7 and shows them with timing information in data grid (Pane 3 in Figure 10).

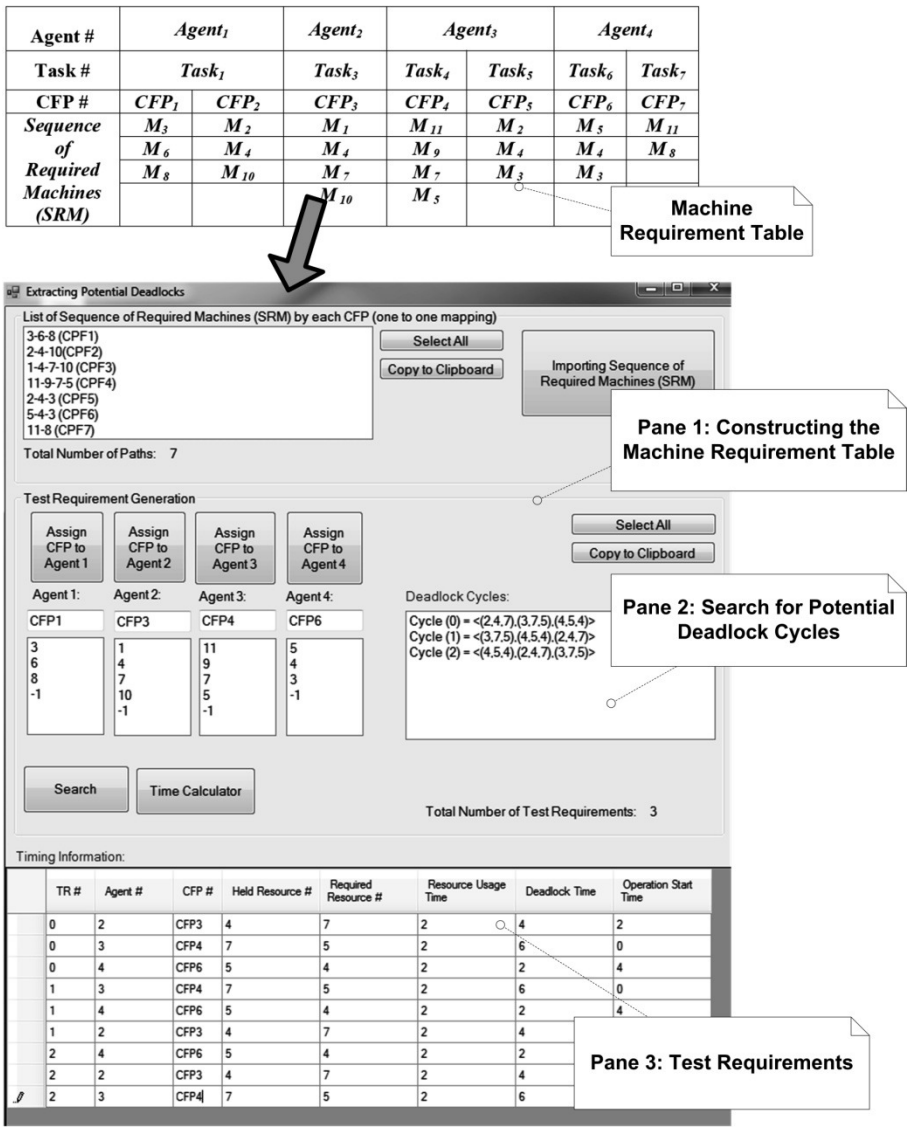

Fig.10. Tool for extracting potential deadlock cycles and generating the test requirements.

For verifying the fact that the extracted potential deadlocks can lead to an actual deadlock at runtime, we developed a simulator which imitates the MAS behaviour in a deadlock condition. A snapshot of the implemented simulator is shown in Figure 11. Using the simulator, a tester is able to verify the behaviour of MAS in the potential deadlocks situations extracted by our proposed algorithm. Figure 11 snapshot shows the behaviour of MAS based on the one of test requirements generated by the potential deadlock extractor tool shown in Figure 10 (TR# 0). The tester defines the specification of a test requirement for the simulator (i.e. information such as start time of each CFP and the sequence of required machines (SRM) for each one) and presses "Run Agents" button (Pane 1 in Figure 11). The simulator imitates that situation and verifies if a deadlock occurs. The monitoring unit is also embedded in the simulator (Pane 2 in Figure 11). So when a deadlock happens the monitoring unit detect it and report to the user. As Figure 11 shows that the monitoring unit detect a deadlock in the system after agents 2, 3, and 4 getting involved in a deadlock cycle. The status of each machine (i.e. locked or free and also if it's locked the locker agent) and each agent is also reported on the real time windows on the tool user interface with proper messages (Pane 3 and Pane 4 in Figure 11).

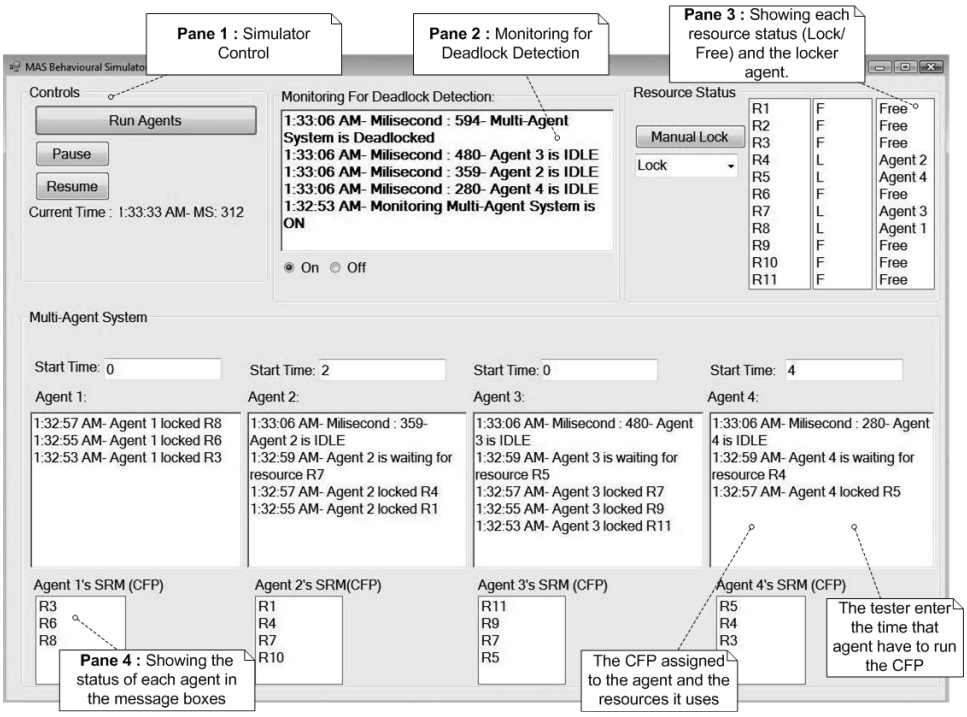

Fig. 11. MAS behavior simulator.

For showing the effectiveness of the system, we performed another experiment in which random input are generated for the discussed simulator to imitate a running MAS till a deadlock happens. Random tasks and CFPs are assigned to the agents and monitoring unit is watching the MAS for deadlocks. This experiment shows that time to a deadlock situation in a running system can be very long and unexpected. This verifies the importance of exiting the testing techniques such as the one proposed in this paper. The results are shown in Table 2.

The experiment is repeated for 9 times and for more accuracy with three different probability distribution functions for random input generation (i.e. equal, normal, and exponential). The second column of the table shows number of simulation runs for each experiment. In each experiment run, a CFP with resources is assigned to each agent with random start times. The third column shows the estimated simulation time based on the time for each run (i.e. in this case study 32 seconds for each run). The fourth column indicates if a deadlock occurs in a specific experiment or not. As it can be concluded from Table 2, deadlocks can appear in the system in a quite long time since their start time. Since the source of this type of behavioral fault is from the design level of the system, it would be very cost effective if these faults can be extracted in the system before the release and during the development process. Our testing technique gives the system tester the opportunity to test a multi-agent manufacturing system for deadlocks while the system is under development.

Table 2. Experiment results — Time to deadlock based on random inputs.

Experiment #	Number of Generated Inputs (Simulation Runs)	Estimated Simulation Time (≈ hrs)	Deadlock Detection	Probability Distribution Function	Parameters for Probability Distribution Function
1	47855	425	No Deadlock	Equal	Domain = [0,5]
2	36454	324	Found		
3	47820	425	No Deadlock		
4	41453	368	Found	Normal	Mu(μ) = 2.5 Sigma (σ) = 0.85 Domain ≈ [0,5]
5	43361	385	Found		
6	47893	425	No Deadlock		
7	47414	421	No Deadlock	Exponential	Lambda (λ)= 0.7 Domain ≈ [0,5]
8	47863	425	No Deadlock		
9	37351	332	Found		

10. Conclusion

Previously in Ref. 5 we proposed a runtime technique for monitoring multi-agent manufacturing systems for deadlocks based on their design model. In Ref. 5 a search algorithm was proposed for extracting the potential deadlock cycles from the design models which were used for system monitoring (Section 6). Although the monitoring technique can be helpful to warn the system user upon deadlock occurrences in the released systems, but the testing techniques can be more useful by reporting the existence

of the deadlocks while the system is under design and development. This can save lots of time and effort from the system developers. Therefore, using the same searching algorithm in Ref. 5, in this paper we format the extracted potential deadlocks into test requirements that can be used for testing a multi-agent manufacturing system under the development. A metamodel for test requirement is proposed which contains the adequate data required for testing deadlock situations (Section 7). For the case study part of this paper, a tool has been developed that is able to find the potential deadlock cycles using the design model information and creates the test requirements based on the test requirement metamodel (Section 9). The tool was also used for generating the test requirement for a case study multi-agent manufacturing system with four agents. Furthermore, a simulator has been developed which is able to imitate the behaviour of a MAS under test and shows that the extracted deadlock situations can lead to deadlocks at runtime. Another experiment has been also performed using the implemented simulator and a random input generator for imitating a running MAS and measuring the time to an unexpected deadlock. That experiment verified that the time to a deadlock in the system is unpredictable and testing techniques for deadlocks can save money and time by detecting deadlock using the design models while the system is under development.

Acknowledgment

The authors were supported by discovery grants from NSERC. Vahid Garousi was further supported by an Alberta Ingenuity New Faculty Award no. 200600673.

References

1. M. R. Genesereth and P. K. Ketchpel, "Software agents," *Commun. ACM*, vol. 37 (7), pp. 48-53, 1994.
2. K. Kumaran, W. Chang, H. Cho, and R. A. Wysk, "A structured approach to deadlock detection, avoidance and resolution in flexible manufacturing systems," *International Journal of Production Research*, vol. 32 (10), pp. 2361-2379, Oct. 1994.
3. F. Bergenti, M. P. Gleizes, and F. Zambonelli, *Methodologies and Software Engineering for Agent System* (New York: Kluwer Academic Publishers, 2004).
4. M. P. Fanti and M. Zhou, "Deadlock control methods in automated manufacturing system," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 34 (1), pp. 5-22, Jan. 2004.
5. N. Mani, V. Garousi, and B. H. Far, "Runtime Monitoring of Multi-Agent Manufacturing Systems for Deadlock Detection Based on Models" in the *21st IEEE International Conference on Tools with Artificial Intelligence* (ICTAI 09), Newark, New Jersey, USA, 2009, pp. 292-299.
6. S. A. DeLoach, "The MaSE Methodology," in *Methodologies and Software Eng. for Agent System*, F. Bergenti, M. P. Gleizes, and F. Zambonelli, Eds. (Boston: Kluwer Academic Publishers, 2004), pp. 107-147.
7. A. Tanenbaum, *Modern Operating Systems* (Englewood Cliffs: Prentice Hall Inc., 1992).
8. E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, pp. 67–78, June 1971.
9. Weiming Shen, D. H. Norrie, and Jean-Paul Barthès, *Multi-agent Systems for Concurrent Intelligent Design and Manufacturing* (London: Taylor & Francis Press, 2001).

10. Objet Management Group (OMG), "UML 2.1.2 Superstructure Specification," November 2007.
11. N. Z. Gebraeel and M. A. Lawley, "Deadlock detection, prevention, and avoidance for automated tool sharing systems," *IEEE Trans. Robot. Automat.*, vol. 17, pp. 342-356, June 2001.
12. S. A. Reveliotis and P. M. Ferreira, "Deadlock avoidance policies for automated manufacturing cells," *IEEE Trans. Robot. Automat*., vol. 12, pp. 845–857, 1996.
13. A. A. Desrochers and R. Y. Al-Jaar, *Applications of Petri Nets in Manufacturing Systems* (New York: IEEE, 1995).
14. F. Chu and X. Xie, "Deadlock analysis of Petri nets using siphons and mathematical programming," *IEEE Trans. Robot. Automat*., vol. 13, pp. 793-804, Dec. 1997.
15. V. Garousi, L. Briand, and Y. Labiche, "Control Flow Analysis of UML 2.0 Sequence Diagrams," in *Model Driven Architecture — Foundations and Application*s, vol. 3748/2005, Berlin/Heidelberg: Springer 2005, pp. 160-174.