information

Article

A Social Multi-Agent Cooperation System Based on Planning and Distributed Task Allocation

Atef Gharbi [1,2]

[1] Faculty of Computing and Information Technology, Northern Border University, Rafha 91911, Saudi Arabia; atef.gharbi@nbu.edu.sa

[2] LISI, INSAT, Université de Carthage, Carthage 1054, Tunisia

Received: 3 March 2020; Accepted: 14 May 2020; Published: 18 May 2020

Abstract: Planning and distributed task allocation are considered challenging problems. To address them, autonomous agents called planning agents situated in a multi-agent system should cooperate to achieve planning and complete distributed tasks. We propose a solution for distributed task allocation where agents dynamically allocate the tasks while they are building the plans. We model and verify some properties using computation tree logic (CTL) with the model checker its-ctl. Lastly, simulations are performed to verify the effectiveness of our proposed solution. The result proves that it is very efficient as it requires little message exchange and computational time. A benchmark production system is used as a running example to explain our contribution.

Keywords: distributed task allocation; multi-agent system; planning agent; computation tree logic (CTL); model checker

1. Introduction

Distributed artificial intelligence (DAI) is classified under two umbrellas: the distributed system and the multi-agent system (MAS). In the distributed system, the focus is on independent intelligent components working together to resolve a problem [1–4]. In the multi-agent system, the autonomous components are called agents [5–8]. Indeed, the multi-agent system is considered very important as it combines artificial intelligence with the distributed system [9–11]. In the MAS, it is evident that there is no central control; each agent has its own independence as well as its own resources to achieve a goal [12–14]. This paper deals with distributed agents, where each one has to interact with the other to maximize the number of successfully distributed tasks [15,16]. We are interested in the agents having the following characteristics: (i) cooperative (no opponents), (ii) likely to be self-interested, (iii) have reliable communication, (iv) heterogeneous, (v) have the possibility to join and leave unpredictably, and (vi) must complete tasks as soon as possible [17–20]. In many of the existing studies, one or more of the above assumptions are violated.

To ensure cooperation between agents in a multi-agent system, planning is needed, which leads to more complications. We distinguish the following five phases to establish multi-agent planning: (i) goal allocation to agents, (ii) goal refinement into subtasks, (iii) subtask scheduling, (iv) planning communication, and (v) plan execution. Agent-based planning is considered as a single agent task [21–23].

In this paper, we consider a multi-agent cooperation system based on planning and distributed task allocation, which is based on the following steps: (1) agent-based planning, (2) formal specification and verification, and (3) evaluation.

Our main contribution in this paper is as follows. Firstly, we define the planning agents based on planning by making action decisions that combines deliberation and reactivity. Secondly, we determine the distributed task allocation applied by planning agents. To ensure the distributed task allocation's correctness, we specify it with a Petri net and we verify some properties using computation tree logic.

The formal verification of some properties expressed in computation Tree logic (CTL) are proven to be true by the appropriate model checker tool.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 introduces a simple benchmark production system to be used throughout this article to demonstrate our contribution. Section 4 describes the agent-based planning in detail. In Section 5, we present the distributed task allocation. Section 6 demonstrates the obtained performance gain with our solution in comparison with the others. Finally, the conclusions and future work are summarized in Section 7.

2. Related Work

Task allocation is considered as a problem of allocating autonomous agents to a set of tasks having some specific properties such as priority, execution time, constraints, etc. When an agent $\mathrm{Ag_i}$ has a task and cannot execute it properly, then the agent $\mathrm{Ag_i}$ tries to find other agents that can execute this task and assigns the task to a particular one. According to [24], the tasks in the multi-agent system can be categorized into the following classes:

(i) Spatially organizing behavior: in this system, the agents interact among themselves to execute a spatial configuration and have minimal interaction with the environment;
(ii) Collective explorations: in this system, the agents focus rather on interacting with the environment than among themselves;
(iii) Cooperative decision making: in this system, the agents interact among themselves as well as with the environment to complete a complicated task.

In this paper, we are interested in cooperative decision making, which can be divided into three categories: centralized, distributed or combined task allocation approaches.

The centralized task allocation approach is characterized by a top-down hierarchical topology where a central server plays a particular role as a coordinator between all agents (like in the blackboard model) [25]. This solution achieves the best possible task allocation solution, but it is not suitable for a large scale of tasks due to the huge computation in the server.

The distributed task allocation approach means that all the agents must cooperate together to find a convenient task allocation solution in the whole system [26–28]. It is considered as a solution to avoid the risk of central server deficiency (i.e., each agent is completely autonomous from the other agents). This solution is more scalable and robust but raises the communication overhead of the agents.

A combined task allocation approach benefits from the two precedent solutions, which means it uses centralized information and does not have any point of failure. However, this solution is not easy to implement and can only be applied to a very specific system (such as a manufacturing system).

In this work, the distributed task allocation approach is adopted. However, our method is still valid for a centralized solution. Many solutions have been employed in distributed task allocation such as the market-based approach, the contract net protocol (CNP), threshold-based algorithms, coalition formation and teamwork.

The market-based approach [29] uses a market-like mechanism where the consumers get the resources needed after negotiation with the providers. This solution is good, but it differs from our solution as it leads to a huge number of exchanged messages due to a large number of providers.

The contract net protocol [30,31] is well used, especially in a multi-agent system. The principle is imitating the economic behavior of the "announce—bidding—award" mechanism. It is based on four steps: (1) task announcement—when an agent needs a task, it announces it to the others (called auction agent); (2) task bidding—the other agents send their proposal to the auction agent; (3) evaluation and selection—the auction agent selects the optimal proposal; and (4) task execution—the auction agent sends a notification to the selected agent. Though this solution is easy to implement, our solution is better as the agents can join or leave the system during the negotiation (the CNP does not allow this modification during the running of the algorithm; that is why the algorithm should be re-executed and would never give results if the modification rate was high). The second advantage of our solution is reducing the number of involved agents.

The threshold protocol [32] was inspired by the biological behavior model of insect colonies. An agent accepts the execution of a new task when it compares an environmental stimulus for the task with some internal preference for the task type and finds it similar. In comparison with our solution, this method suffers from a major inconvenience; it is impossible to deal with changes such as adding a new task or removing an existing task while running the threshold protocol. Therefore, we cannot profit from the previously obtained solution whenever the task set is changed.

The coalition formation and teamwork approach [33] is a complex solution consisting of task splitting, task allocation, coordination and communication. It is assumed that the agent's organization is known.

Our solution is more flexible, as the coalition formation and teamwork approach is more restricted regarding the design and behavior of the members that constitute a group.

3. Benchmark Production System

As much as possible, we will illustrate our contribution with a simple current example called RARM [34]. We describe it informally, but it will be used as an example of the various formalisms presented in this article. The benchmark production system RARM, depicted in Figure 1, is constituted by two input conveyors and one output conveyor, a servicing robot and a processing–assembling unit. Workpieces to be treated as they come irregularly one by one. The workpieces of Type A are carried via the conveyor C1, and the workpieces of Type B, via the conveyor C2. Only one workpiece can be on the input conveyor. A robot R transfers workpieces one after another to the processing unit. The next workpiece can be put on the input conveyor when it has been emptied by the robot.

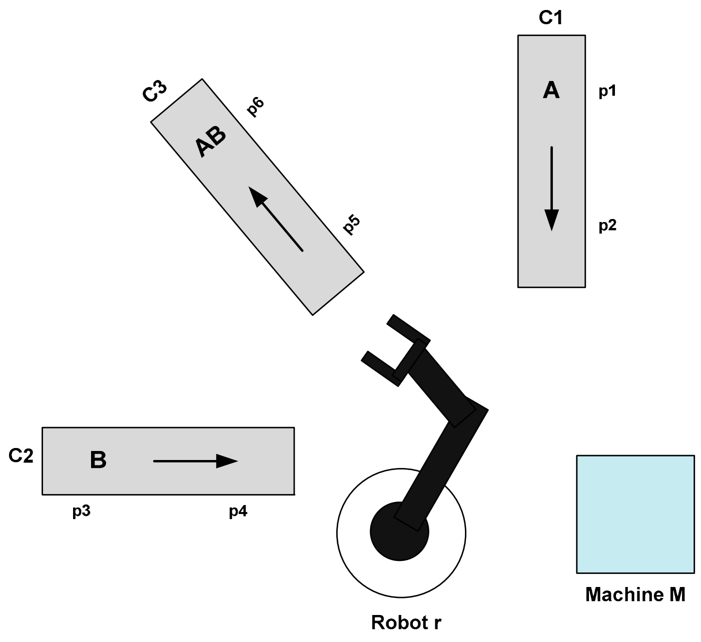

Figure 1. The benchmark production system RARM.

Three production strategies can be defined (i) the first production policy consists of inserting a Type A workpiece (through the conveyor C1) into the processing center M to be treated, then it is evacuated by the robot to the output conveyor C3; (ii) the second production policy consists of inserting a Type B workpiece (through the conveyor C2) into the processing center M to be treated, then it is evacuated by the robot to the output conveyor C3; (iii) the third production policy consists of inserting a Type A workpiece into the processing center M to be treated, then a Type B workpiece is added in the center and the two workpieces are finally assembled. Afterward, the assembled product is taken by the robot and put above the C3 conveyer of output. The assembled product can be transferred on C3 only when the output conveyor is empty and ready to receive the next one produced.

Running Example

RARM is composed of autonomous conveyors, robots, and machines that we call agents. RARM is constructed in a distributed manner since each agent can perform detailed tasks by itself. The agents differ in the types of action that they can perform. Thus, agents should cooperate to fulfil tasks.

4. The Agent-Based Planning

In general, agents can be classified based on their decision making: (i) reactive agents, whose decisions are taken according to sensor inputs; and (ii) planning agents, whose decisions are more intelligent as they consider many constraints such as sensing inputs, goal, intention and/or utility.

We propose the conceptual planning model of an agent, which is composed of the following components: (i) the sensing input, (ii) the decision making and (iii) the actuator output (Figure 2). Thus, each planning agent is autonomous in sensing, acting and making a decision.

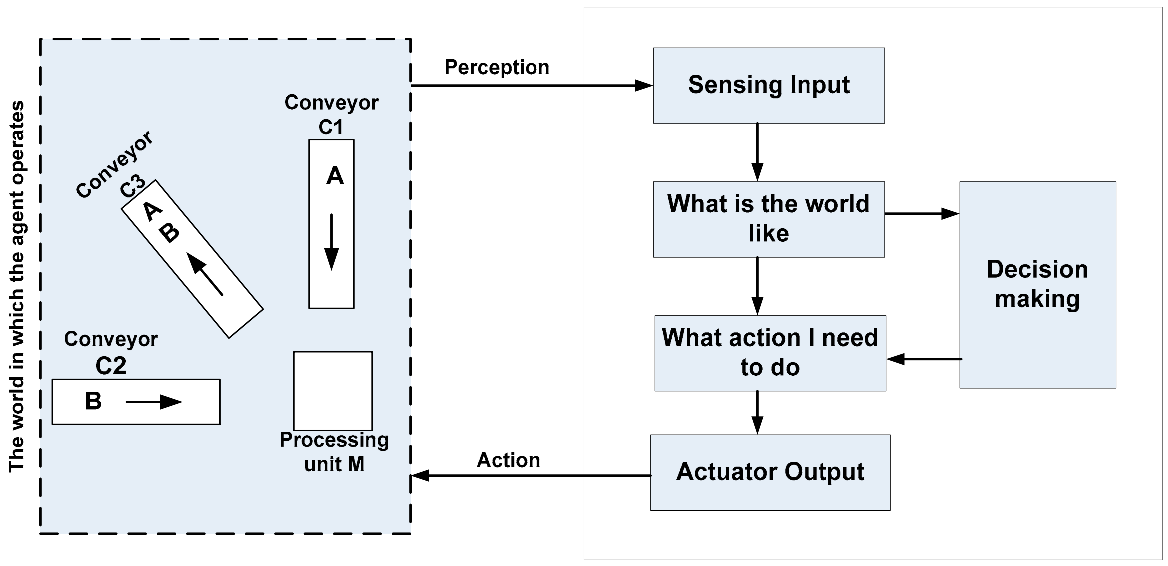

Figure 2. The conceptual planning model.

Given a description of the environment in which the planning agent exists, the goal to achieve, the set of operations to be performed, and the current state of the agent, the planning agent determines a plan, which is an arrangement of actions to be performed to attain the goal. The planning agent is needed either to reach a goal that satisfies some conditions or to perform several tasks in a specific order.

The most important step in the planning model is decision making. We propose the decision making as a state machine that includes a description of different states, the initial state(s), and the transitions from one state to another state according to some conditions defined in advance. Therefore, a classical planning problem is represented as a state-transition model. The planning agent starts from the current state, which constitutes the relevant part of the world, and tries to reach another state, which represents the goal state (see Algorithm 1).

To define the internal behavior of the agent, we use the following algorithm where:

- The queue, Q, is used to save every arriving event;
- The event, ev, represents any event input;
- The state, Si,j, is an element of the state machine.

Algorithm 1 GenericBehavior.

```
begin
while (Q.length() > 0) do
        ev←Q.Head()
        $S_{i,j}$←$currentState_i$
        If ev ∈ I($S_{i,j}$) then
            For each state $S_{i,k}$ ∈ next($S_{i,j}$)      do
              1. Find the plan list that can satisfy $S_{i,k}$
              2. Assess every possible plan's pre-conditions and retain only those whose pre-conditions are satisfied.
              3. For every remaining plan, estimate its required resources.
              4. Sort by its priority
                If execute($S_{i,k}$) then
                      $currentState_i$←$S_{i,k}$
                      break
                end if
            end for
        end if
end while
end.
```

Running Example

If we consider RARM, we define the following parts:

(i) The sensing input: the planning agent knows its environment through sensors. Thus, the data provided by the sensors present the agent's vision of its environment. It is defined as follows:

✓ The sensor sens1 (respectively sens2) is used to verify if there is a workpiece at the position *p*1 (respectively the position *p*2) on the conveyor C1;
✓ The sensor sens3 (respectively sens4) checks for the existence of a workpiece at the position *p*3 (respectively the position *p*4) on the conveyor C2;
✓ The sensor sens5 (respectively sens6) verifies if there is a workpiece at the position *p*5 (respectively the position *p*6) on the conveyor C3;
✓ The sensor sens7 observes if there is a Type A workpiece at the unit *M;*
✓ The sensor sens8 enables the checking of if there is a Type B workpiece at the unit *M;*
✓ The sensor sens9 (respectively sens10) perceives if the conveyor *C*1 is in its extreme left (respectively right) position;
✓ The sensor sens11 (respectively sens12) determines if the conveyor *C*2 is in its extreme left (respectively right) position;
✓ The sensor sens13 (respectively sens14) detects if the conveyor *C*3 is in its extreme left (respectively right) position;
✓ The sensor sens15 (respectively sens16) is used to decide if the robotic agent arm is in its lower (respectively higher) position.

(ii) The actuator output: the planning agent can control the environment using the following actuators:

✓ The actuator act1 ensures the movement of the conveyor *C*1;
✓ The actuator act2 moves the conveyor *C*2;
✓ The actuator act3 enables the movement of the conveyor *C*3;
✓ The actuator act4 rotates a robotic agent;
✓ The actuator act5 elevates the robotic agent arm vertically;
✓ The actuator act6 picks up and drops a piece with the robotic agent arm;
✓ The actuator act7 treats the workpiece;

✓ The actuator act8 assemblies two pieces.

(iii) The decision making: the planning agent decides the actions to perform to achieve the goal. The set of actions is {*Conveyor*1*_left*, *Rotate*1*_left*, *Rotate*1*_right*, *Conveyor*2*_left*, *Rotate*2*_left*, *Rotate*2*_right*, *Conveyor*3*_left*, *Rotate*3*_left*, *Rotate*3*_right*, *take*1, *take*2, *take*3, *load*1, *load*2, *load*3, *put*1, *put*2, *put*3, *process*1, *process*2}. These are as follows:

✓ *Conveyor*1*_left* (respectively *Conveyor*2*_left, Conveyor*3*_left*) means a workpiece of type *A* (respectively *B, AB*) is moving to the left of conveyor *C*1 (respectively *C*2, C3) from position *p*1 (respectively *p3, p5*) to position *p*2 (respectively *p4, p6*);
✓ *Rotate*1*_left* (respectively *Rotate*2*_left, Rotate*3*_left*) means the robotic agent taking a workpiece of type *A* (respectively *B, AB*) is moving to the left from the position *p*2 (respectively *p4, p6*) of conveyor *C*1 (respectively *C2, C3*) to the processing unit *M* (respectively the position *p*2 of conveyor *C*1);
✓ *Rotate*1*_right* (respectively *Rotate*2*_right, Rotate*3*_right*) means the robotic agent taking a workpiece of type *A* (respectively *B, AB*) is moving to the right from the processing unit *M to* the position *p*2 (respectively *p4, p6*) of conveyor *C*1 (respectively *C2, C3*);
✓ take1 (respectively take2, take3) means the operation of taking a workpiece of type A (respectively B, AB);
✓ load1 (respectively load2, load3) means loading a workpiece of type A (respectively B, AB);
✓ put1 (respectively put2, put3) means the operation of putting a workpiece of type A (respectively B, AB);
✓ process1 (respectively process2) means processing a workpiece of type A (respectively B).

5. Distributed Task Allocation

Distributed task allocation is necessary when a task cannot be executed by only one planning agent. Planning agents situated in the multi-agent system should cooperate to achieve this task.

The formal definition of distribution task allocation in a multi-planning agent is as follows:

- a set of planning agent $A = \{A_1, \dots, A_n\}$;
- each planning agent Ai has a set of resources $R_i = \{R_{i1}, \dots, R_{im}\}$;
- a set of tasks to be executed in the multi-planning agent $T = \{T_1, \dots, T_p\}$;
- each task T_k may require a set of resources $\{R_{k1}, \dots, R_{kl}\}$.

A distributed task allocation is accomplished if, and only if, every task T_k is executed on an agent A_i having all the required resources for this task.

5.1. The Principle of Distributed Task Allocation

To ensure distributed task allocation, we define two kinds of planning agent:

✓ Initiator: the planning agent that requests help to perform its task;
✓ Participant: the planning agent having the requested resources that receives the announced task and sends back its response.

To ensure the coherent behavior of the planning agent, we propose two possible states, i.e., States = {BUSY, IDLE}. Whenever a planning agent is an initiator, the planning agent state should be BUSY. We consider a planning agent in an IDLE state when it is free, i.e., when it has no task to do. We suppose that only an IDLE planning agent can be assigned to a new task as an initiator or as a participant.

The idea of the algorithm is illustrated as follows.

✓ The initiator agent, denoted as *InitA*: the planning agent that starts the distributed task allocation process. There are many reasons to apply a distributed task allocation in the whole system such as the fact that the planning agent is unable to execute a task by itself, it is too busy, or it does not have all the necessary resources. Each initiator agent InitA has a list of planning agents. It sends a request to the planning agents: Resource_Announce_Message = <AgentID, TaskID, Resource(*number*)>, containing the initiator agent identity, the task identity and the number of requested resources.

The initiator agent waits for the response from the other planning agents.

- ✓ The planning agent (called the participant agent) receives the message *Resource_Announce_Message* sent by the initiator agent *InitA*; it checks the possibility of applying distributed task allocation based on its state. If it is feasible (i.e., the participant agent has the requested resources and is idle), it accepts the collaboration with the initiator agent. If the participant agent is busy, it refuses the request.

    - ➢ If the participant agent *Aj* is IDLE then it provides data regarding its identity, the types of resources it owns and the executing time, explicitly, Propose_Message = <AgentID, Resource, Execute>.
    - ➢ If the participant agent *Aj* is BUSY then it sends a negative response containing the following message Refuse_Message = <AgentID>.
- ✓ After collecting the responses from all the participant agents or the timeout is over, the initiator agent InitA then compares the available resources from these participant agents with the resources required for its task *tInitA*. This leads to two cases:

If the initiator agent InitA obtains all the requested resources then the initiator agent sends a notification message to the selected participant agent Aj. The initiator planning agent waits for a confirmation.

- ✓ Two cases can happen depending on the selected participant agent Aj state

    - ➢ If the selected participant agent Aj is still free, it sends a confirmation message to the initiator agent InitA and its state will be changed to BUSY: Confirm_Message = <AgentID>. Consequently, the distributed task allocation protocol finishes well.
    - ➢ If the selected participant agent Aj becomes busy (which means it receives a notification message from another initiator), it sends a decline message to the initiator agent InitA: Decline_Message = <AgentID>. Therefore, the initiator agent InitA repeats the same steps from the beginning.

The different distributed task allocation steps are illustrated in Figure 3, where nbR represents the number of participant agents related to the initiator agent InitA.

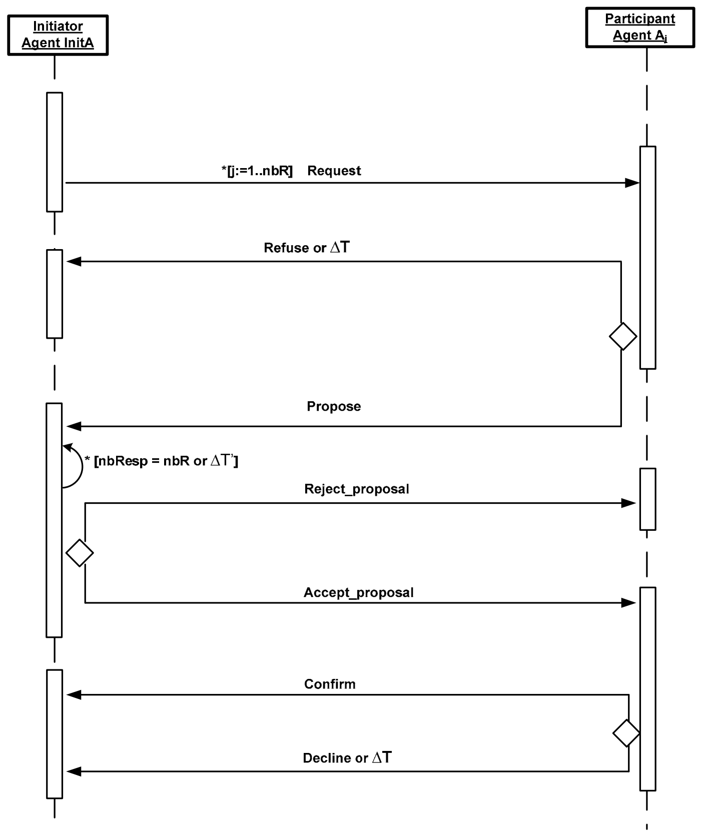

Figure 3. Distributed task allocation approach.

5.2. Dynamic Planning Agent Discovery with JADE

We implement the different planning agents with the JADE (JavaTM Agent DEvelopment) framework. JADE is a platform to develop MAS in compliance with the Foundation for Intelligent Physical Agents (FIPA) specifications.

With the platform JADE, we find the directory facilitator (DF) providing a Yellow Pages service where the initiator agent can find other planning agents providing the resources it needs to accomplish its tasks. The jade.domain.DFService class provides static utility methods that facilitate the interactions with the DF: register(), modify(), deregister(), and search() [35]. The first step is that the planning agent publishes its provided resources. The second step is that the initiator agent searches in the directory facilitator to discover the planning agents providing the required resources.

5.2.1. Publish a Provided Resource

The planning agent can publish a provided resource by creating a proper description, which is an instance of the DFAgentDescription class, and call the register() method of the DFService class. The planning agent must provide a description of:

➢ The agent AID (unique identifier);
➢ A collection of resource descriptions (class ServiceDescription):

- The resource type (e.g., "Resource1");
- The resource name (e.g., "Resource");
- The languages, ontologies and interaction protocols that must be known to exploit the service.

Running Example

```
/// Register the Planning Agent in DFService
      DFAgentDescription reg = new DFAgentDescription();
      reg.setName(getAID());
      ServiceDescription desc = new ServiceDescription();
      desc.setType("Resource1");
      desc.setName("Resource");
      reg.addServices(desc);
      try {
            DFService.register(this, reg);
      }
      catch (FIPAException fe) {
            fe.printStackTrace();
      }
```

5.2.2. Search a Required Resource

The initiator agent can search for some required resources by providing the directory facilitator with a template description. The result of the research is a list of all the descriptions matching the template. The search() method of the DFService class provides the result. The final result will be the set of planning agents providing the requested resource.

Running Example

```
DFAgentDescription template = new DFAgentDescription();
          ServiceDescription desc = new ServiceDescription();
            desc.setType("Resource1");
            template.addServices(desc);
              DFAgentDescription[[] result;
            try {
              do
              {
                        result = DFService.search(myAgent, template);
                        planningAgents = new AID[result.length];
                               for (int i = 0; i < result.length; i++)
                                        planningAgents[i] = result[i].getName();
                        }
                while (result.length <= 0);
              }
              catch (FIPAException fe) {
                 fe.printStackTrace();
              }
nbR = planningAgents.length;
```

5.3. Message Exchanged between Initiator and Planning Agents with JADE

A multi-agent system interacts through exchanging asynchronous messages. All participant agents communicate using messages that correspond to the FIPA ACL specification. This format comprises a number of fields and in particular (1) the communicative intention (also called "performative"), indicating what the sender intends to achieve by sending the message (for instance the performative can be REQUEST, INFORM, QUERY_IF, CFP, PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL, etc.); (2) the content, i.e., the actual information included in the message, which may be a string in simple cases—otherwise, we need a content language, a corresponding ontology and a protocol; and (3) the list of receivers.

5.3.1. Creating a Message According to FIPA ACL Specification

Sending a message to another participant agent is as simple as filling the fields of an ACLMessage object and then calling the send() method of the agent class.

Running Example

```
// The code below creates a new message with the following content:
ACLMessage msg = new ACLMessage(ACLMessage.INFORM); // (1) the performative: INFORM,
msg.addReceiver(new AID("Agent1", AID.ISLOCALNAME)); // (3) the receiver: Agent1
msg.setContent("I need the help from others"); // (2) the content: requesting help from other agents
send(msg);    // sending the message
```

5.3.2. Receiving a Message with a Matching Method

When a template is specified, the receive() method returns the first message (if any) matching it, while ignoring all non-matching messages. Such templates are implemented as instances of the jade.lang.acl.MessageTemplate class, which provides a number of factory methods to create templates in a very simple and flexible way.

Running Example

The action() method is modified so that the call to myAgent.receive() ignores all messages except those whose performative is PROPOSE:

```
  public void action() {
MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage. PROPOSE);
ACLMessage msg = myAgent.receive(mt);
if (msg != null) {
// REQUEST Message received. Process it
...
}
else {
block();
}
}
```

5.4. Agent Behavior in JADE

A behavior represents a task that a participant agent can carry out. A behavior is a kind of control thread for the agent where the method action() is similar to Thread.run(). New behaviors can be added at any time during the agent's life (Figure 4).

- **Behaviour class:** it is implemented as an object of a class that extends jade.core.behaviours.Behaviour. The Behaviour class is an abstract class having several abstract methods. The ***action*** method defines the operation to be performed when the behavior is in execution. The ***done*** method returns a boolean value to indicate whether or not a behavior has completed. The Behaviour class also provides two methods, named ***onStart*** and ***onEnd***. These methods can be overridden by user defined subclasses when some actions are to be executed before and after running behavior execution.
- **SimpleBehaviour class:** it is an abstract class representing simple atomic behaviors (which means it is composed of only one behavior). It can be overridden by user defined subclasses which are OneShotBehaviour and CyclicBehaviour classes.

- **OneShotBehaviour class:** it represents atomic behaviors that must be executed only once and cannot be blocked. Two subclasses inherit from the OneShotBehaviour class, which are the WakerBehaviour and the TickerBehaviour classes.

  ✓ The **WakerBehaviour** class implements a one-shot task that must be executed only once just after a given timeout has elapsed, which is specified in the ***handleElapsedTimeout*** method.

  ✓ The **ReceiverBehaviour** class, which triggers when a given type of message is received or a timeout expires.

- **CyclicBehaviour class:** it models atomic behaviors that must be executed forever. Thus, its ***done*** method always returns false. The CyclicBehaviour class represents "Cyclic" behaviors that never complete and whose ***action*** method executes the same operations each time it is called.

  ✓ The **TickerBehaviour** class implements a cyclic task that must be executed recurrently. The periodic actions are determined in the ***onTick*** method.

- **CompositeBehaviour class:** This abstract class models behaviors that are made up of a number of other behaviors (children). Thus, the actual operations performed by executing this behavior are not defined in the behavior itself but inside its children, while the composite behavior only takes care of child scheduling according to a given policy (sequentially for the SequentialBehaviour class, concurrently for the ParallelBehaviour class and as a finite state machine for the FSMBehaviour class).

  ✓ **ParallelBehaviour** class: this behavior controls a set of sub-behaviors that execute in parallel. The most important thing is the termination condition; it is possible to specify the termination of ParallelBehaviour when **all** of its sub-behaviors terminate or **any** sub-behavior is ended.

  ✓ **SequentialBehaviour** class: this behavior is composed of a set of sub-behaviors where each one is executed after the other. The SequentialBehaviour terminates when all its sub-behaviors have terminated.

- **ReceiverBehaviour class:** a behavior with a timeout that waits until the reception of a message or the elapsing of a given timeout.

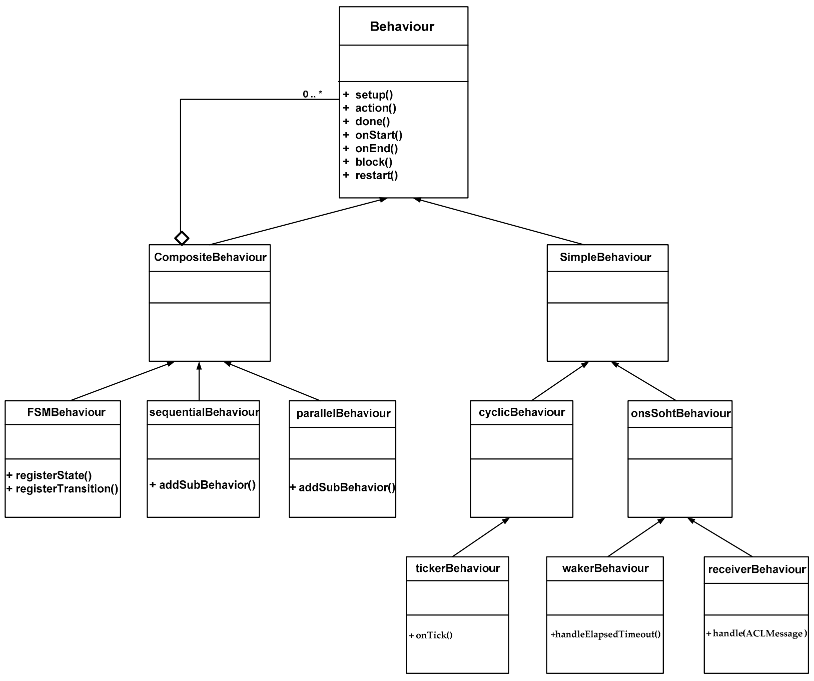

Figure 4. Behavior class hierarchy in JADE.

Running Example

We can design the dynamic initiator agent behavior with JADE. To make the initiator agent execute a task implemented by a behavior object, the behavior should be added to the agent by means of the addBehavior() method of the agent class in the setup() method or inside another behavior. First of all, the initiator agent broadcasts a message with the performative Query_REF to all the participant agents. The behaviors include:

- ✓ A ParallelBehaviour, which is composed of a sub-behavior set. Each sub-behavior is a ReceiverBehaviour class, which means it waits until the reception of a message, an answer from a particular participant agent or a given timeout is elapsed. The termination of this ParallelBehaviour is ensured when **all** of its sub-behaviors (i.e., ReceiverBehaviour) terminate. The ReceiverBehaviour have 0.5 s as a timeout period.
- ✓ After that, there is a delay (1 s) before sending a REQUEST to the best answer from a specific participant agent.
- ✓ The final behavior is a receiver behavior, which waits for an AGREE/REFUSE message from this specific participant agent with a common conversation ID.

In the case of failure, our initiator agent launches another round of negotiation by simply calling setup again.

```
public class Initiator extends Agent
{
    private Initiator myAgent;
    private int requested;
    private int resourceType;
    int      bestExecute;
   ACLMessage message, bestProposal;
    static protected Random genAleatoire = new Random();

protected void setup()
    {
Object[[] args = getArguments();
if (args != null && args.length > 0) {
    resourceType=((Integer)args[0]).intValue();
    }
bestExecute = 1000;
bestProposal = null;
// number of resources requested: generated rondom
    requested=(int) Math.floor(genAleatoire.nextDouble() *10;
  myAgent=this;
         // Register the resource offered by the Initiator agent in DFService
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());
    ServiceDescription sd = new ServiceDescription();
    sd.setType(String.valueOf(resourceType));
    sd.setName("Resource");
    dfd.addServices(sd);
    try {
      DFService.register(this, dfd);
    }
    catch (FIPAException fe) {
      fe.printStackTrace();
    }

// Add a TickerBehaviour that searches Planning agents every 2 seconds in the DF
addBehaviour(new TickerBehaviour(this, 2000)
{
protected void onTick() {
    // Update the list of Planning agents
    DFAgentDescription template = new DFAgentDescription();
      ServiceDescription desc = new ServiceDescription();
          desc.setType(resourceType);
          template.addServices(desc);
           DFAgentDescription[[] result;
          try {
           do
           {
               result = DFService.search(myAgent, template);
               planningAgents = new AID[result.length];
               for (int i = 0; i < result.length; i++)
                                  planningAgents[i] = result[i].getName();
                    }
             while (result.length <= 0);
            }
          catch (FIPAException fe) {
             fe.printStackTrace();
          }
    nbR= planningAgents.length;
    }
};
          message = newMsg( ACLMessage.REQUEST );

          MessageTemplate template = MessageTemplate.and(
              MessageTemplate.MatchPerformative( ACLMessage.PROPOSE ),
              MessageTemplate.MatchConversationId(message.getConversationId() ));
          SequentialBehaviour seq = new SequentialBehaviour();
          addBehaviour( seq );

          ParallelBehaviour par = new ParallelBehaviour( ParallelBehaviour.WHEN_ALL );
          seq.addSubBehaviour( par );

          for (int i = 0; i < planningAgents.length; i++)
          {
               message.addReceiver( new AID(planningAgents[i], AID.ISLOCALNAME ));
               par.addSubBehaviour( new ReceiverBehaviour( this, 500, template)
                      {
                          public void handle( ACLMessage message)
                          {
                              if (message != null) {
                                  int execute = Integer.parseInt( message.getContent());
                                  if (execute < bestExecute) {
```

```
                                        bestExecute = execute;
                                        bestProposal = message;
                                  }
                              }
                          }
                     });
          }

          seq.addSubBehaviour(new DelayBehaviour(this, 1000)
          {
             public void handleElapsedTimeout()
             {
                        if (bestProposal != null) {
                             //The best proposal is obtained through bestProposal.getSender()
                             ACLMessage reply = bestProposal.createReply();
                             reply.setPerformative( ACLMessage.ACCEPT );
                             send ( reply );
                        }
             }
          });

      seq.addSubBehaviour( new ReceiverBehaviour( this, 500,
                   MessageTemplate.and(
                   MessageTemplate.MatchConversationId(message.getConversationId()),
                        MessageTemplate.or(
                             MessageTemplate.MatchPerformative( ACLMessage.CONFIRM ),
                             MessageTemplate.MatchPerformative( ACLMessage.REFUSE ))) )
                   {
                        public void handle( ACLMessage message)
                        {
                             if (message != null ) {
                                   if( message.getPerformative() == ACLMessage. CONFIRM)
                                   System.out.println("Distributed Task Allocation is Finished");
                                   else
                                   // repeat the same steps in case of failure
                                   setup();
                              }
                              else {
                                   //The time is elapsed without response so repeat the same steps
                                   setup();
                              }
                        }
                   });

          send ( message );
}
```

5.5. The Model Checking of Distributed Task Allocation

The model checking of distributed task allocation consists of two steps. The first step is to represent the distributed task allocation with the Petri net, which permits the representation of a concurrent process. The second step is the verification of some properties using computation tree logic (CTL) with the model checker ***its-ctl*** [36].

As explained in the previous section, we distinguish two kinds of participating agents, which are the initiator and the participant agent (Figure 5). For further explanation of the Petri net diagram (especially the transition meanings), refer to Table 1.

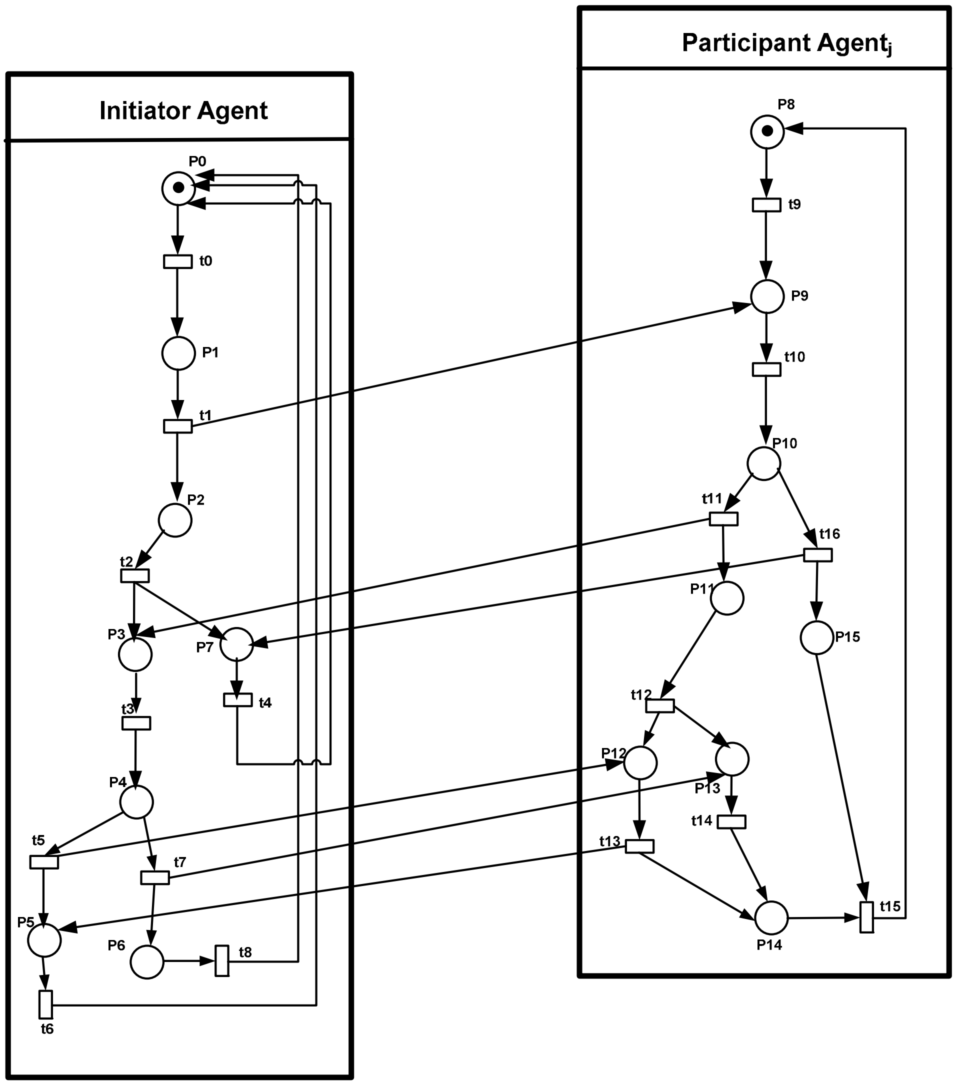

Figure 5. Formal distributed task allocation with Petri net.

Table 1. The meaning of the main transitions related to Figure 5.

Transition	Meaning
t0	The initiator agent needs to apply a distributed task allocation
t1	The initiator agent sends a request to a participant agent$_i$
t3	The initiator agent receives a positive answer from the participant agent$_i$
t4	The initiator agent receives a negative answer from the participant agent$_i$
t5	The initiator agent accepts the proposition from the participant agent$_i$ to apply the new distributed task allocation
t7	The initiator agent refuses the proposition from the participant agent$_i$ to apply the new distributed task allocation
t9	A *participant agent*$_j$ receives the request from the initiator agent for distributed task allocation
t10	The *participant agent*$_j$ receives a disapproval message from the initiator agent
t11	The *participant agent*$_j$ sends a proposition to the initiator agent
t12	The *participant agent*$_j$ receives an approval message from the initiator agent
t13	The *participant agent*$_j$ send a confirmation to the initiator agent
t16	The *participant agent*$_j$ refuses to help the initiator agent

CTL properties: To verify that our Petri net model is correct, we present some properties expressed with the CTL formalism. These properties are checked by the ITS-CTL model checker [36]

- Property 1: Whenever an initiator agent needs to ensure a task but does not have all the necessary resources, the initiator agent informs the participant agent$_j$.

  AG (P1 = > EF P9).

- Property 2: During the negotiation, the initiator agent receives a positive response from the participant agent$_j$ or receives a disapproval message.

  EF P1 AND EF (P11 OR P15).

- Property 3: The participant agent$_i$ could not receive two different decisions from the initiator agent at the same time (i.e., either the initiator agent accepts or refuses the new distributed allocation).

  NOT EF (P5 AND P6).

All these properties are proven to be true with the ITS-CTL model checker.

eCTL properties: The following eCTL property is proven to be true:

Property 4: Whenever the participant agent$_j$ accepts the new distributed task allocation, it awaits the final decision from the initiator agent, which can be a confirmation to apply a new distributed allocation or to cancel the new distributed allocation.

AGA t11 XAGA (t5 OR t7) X p14.

6. Evaluation

To evaluate our proposed solution for distributed task allocation in a multi-agent system, we test its efficiency compared with a second approach [37] and the greedy distributed allocation protocol (GDAP) [38]. We choose these two approaches because they are closed and similar to our work.

With the second approach [37], we find three kinds of agents: the manager (the same as the initiator agent in this paper); the participant (the same as the planning agent in this paper), which offers to perform the announced task; and the mediator, which helps the manager to find resources by requesting its own neighbors. This approach is good but gives the priority to direct linked neighboring agents (for the manager), which could affect the solution. The main difference between the second approach [37] and our solution is as follows: in the second approach [37], the neighbors can be related to the manager as well as the mediator, but in our solution, the neighbors are only related to the initiator agent having the requested resources that can be involved in the solution.

The greedy distributed allocation protocol (GDAP) [38] can be described briefly as follows: there are many manager agents (each manager is responsible for a task) and many contractor agents (each contractor controls a resource). All manager agents (the same as the initiator agent in this paper) try to find contractor agents (the same as the participants in this paper) who can help them with their tasks. Contractor agents offer help to the manager agent by sending a bid (consisting of all the available resources to be allocated to the task). In the end, there is a possibility that the manager agent collects all the required resources and so informs the corresponding contractors. The GDAP can lead to many unallocated tasks due to static neighbors. In our approach, we try to resolve this problem by dynamically discovering the neighboring agents providing the requested resources.

We implemented the three approaches for the distributed task allocation algorithm in the JADE Framework, and we tested them. The performance of our solution is evaluated through a different number of neighboring agents. The only changing variable in this setting is the average number of neighbors. This setting, as shown in Table 2, intends to represent the influence of neighbors' numbers on the performance of three approaches.

Table 2. The details of the settings.

Setting	Quantity
Number of agents	10
Average number of neighbors	[1..10]
Tasks	5
Available resources	6

The algorithms have been evaluated according to three criteria in this experiment: the communication cost, the execution time and the utility ratio.

Firstly, we compare the three approaches by counting the maximum number of exchanged messages between agents after each distributed task allocation scenario. For our approach, we denote by numberMsg messages such the number when we use only two kinds of agents, which are initiator and participant agents. For the second approach [7], we denote by numberMsgMd messages such the number when we use manager, mediator and participant agents in coordination.

If we apply our approach, in this case, the initiator agent has to inform all the planning agents (i.e., N) to apply any distributed task before waiting for their responses and, after that, decide to accept or to refuse the offer; then, numberMsg = 2 * N.

If we apply the second approach [37], in this case, the manager agent has to inform all its neighboring agents (i.e., N) to apply any distributed task. The worst case is needing the help of another agent (mediator), which also in turn needs another mediator, and so forth. If we consider that each mediator agent sends the request to its neighboring agents, P represents the number of mediator agents existing in the distributed system (where P << N). Then, numberMsgMd = (2 * N) * P.

If we apply the GDAP, the manager agent has to request only its neighbors (in the best case). In the worst case, there is a failure in task allocation and the same process has to be repeated several times by another managers. The worst case is to have the number of manager agents existing in the distributed system being equal to all the agents. Then, numberMsgMd = (2 * N) * N.

The algorithms have been evaluated according to the *number of exchanged messages* as a criterion in this experiment. All the results are obtained by applying the logarithm function.

From Figure 6, it is clear that whenever the number of neighboring agents is reduced, the three approaches have approximatively the same performance. Whenever the number of neighboring agents is elevated, it is evident that our approach is the best as it reduces the communication cost. In conclusion, our proposed solution is very efficient (especially when the neighboring agent number is high) as it requests little message exchange. The approach in [37] suffers from a very high message exchange rate in the case of the neighboring agent number being important.

The second criterion is the execution time. The unit of execution time is the millisecond. For simplicity, we suppose that once a task has been allocated to an agent, the agent would successfully finish this task without failure. We would like to test, in this experiment, the impact of different average numbers of neighbors on the execution time applied for these three approaches.

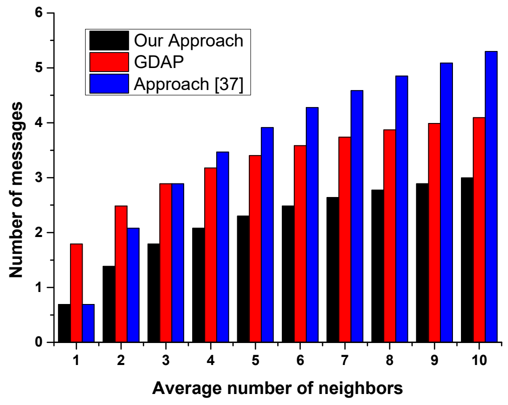

Figure 6. Comparison between the three approaches based on exchanged messages.

We notice in Figure 7 that our approach tracks the lowest execution time amongst the other ones.

The third criterion is the utility ratio, which is defined as the sum of completed tasks divided by the total number of tasks.

$$\text{Utility Ratio} = \frac{\sum successful - completed - task}{Total - of - Tasks}$$

We would like to test, in this experiment, the influence of different average numbers of neighbors on the three algorithms. We notice in Figure 8 that the utility ratio of our algorithm in different networks is more reliable than that of the GDAP algorithm. We notice also that the GDAP solution utility is the worst in the majority of cases.

Whenever the number of neighboring agents is reduced, our solution performs moderately in terms of utility. This is can be explained by the following:

- Not all the planning agents having the requested resources are registered in the directory facilitator (some of them are selfish and do not like sharing their resources).
- Even when the initiator agent searches the neighboring agent providing the requested resources (the number is reduced), some of the planning agents refuse to participate in the distributed task allocation for the simple reason that the resource can be used by itself later.

Therefore, the probability of finding a solution with a reduced number of neighboring agents is low. However, when the number of neighboring agents is elevated, there is more chance of finding a solution, which means among all these neighboring agents, there are at least some that are free and willing to help the initiator agent. Whenever the number of neighboring agents is elevated (it is expected that there are many resources), our approach works fine and gives a result similar to that of the second one [37].

For more details about the implementation as well as the data used in this paper, a useful link is shared with the public [39].

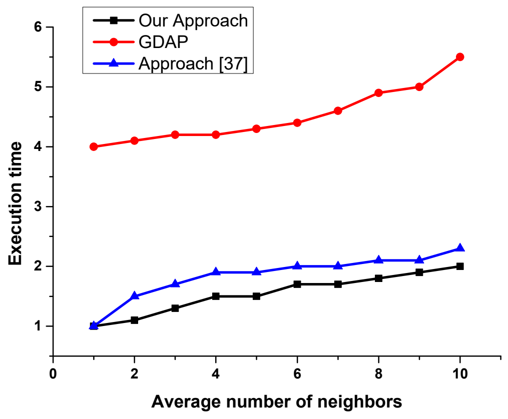

Figure 7. Comparison between the three approaches based on execution time.

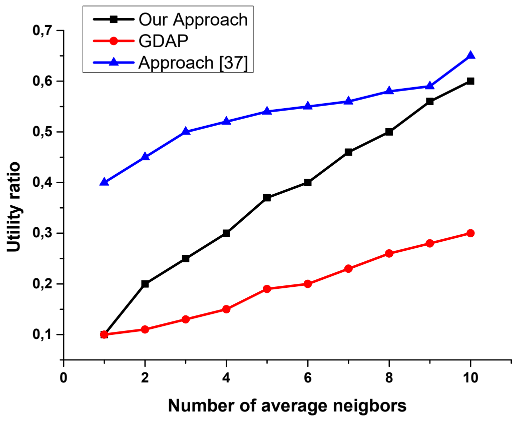

Figure 8. Comparison between the three approaches based on the utility ratio.

7. Conclusions

In this paper, we consider the problem of multi-agent cooperation systems based on planning and distributed task allocation. The problem is divided into two parts: agent-based planning and distributed task allocation. There are many contributions as follows. (1) In the first part, we define the planning agents based on planning by making action decisions that combine deliberation and reactivity. (2) In the second part, we determine the distributed task allocation applied by planning agents. To ensure the communication protocol's correctness, we specify it with the Petri net, and we verify some properties using computation tree logic. Our method can be used for solving the multi-agent planning problem, in which agents can plan cooperatively more flexibly. Planning agents behave more as though they are thinking, by making decisions about task allocation and predicting the effects of their decisions.

The experimental results prove that the proposed approach has good performance. The main goals of our solution are to ensure robustness (the formal verification of certain properties that demonstrate the correct behaviour of the system based on computation tree logic), reliability (even if an agent is unable to execute a task, it can be done by another), scalability (there no limit to the possible agents or tasks that can be added), openness (the agents can join or leave the system at any time), communication reduction (minimal exchanged messages), and structure flexibility (there is no need to be organized in groups).

Furthermore, several issues are worth future research. Fault tolerance is a crucial point to study; it can be considered in future work.

Acknowledgments: The authors gratefully acknowledge the approval and the support of this research study by the grant no-7142-CIT-2017-1-7-F-from the Deanship of Scientific Research at Northern Border University, Arar, K.S.A.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Aramrattana, M.; Andersson, A.; Reichenberg, F.; Mellegård, N.; Burden, H. Testing cooperative intelligent transport systems in distributed simulators. *Transp. Res. Part F Traffic Psychol. Behav.* **2019**, *65*, 206–216. [CrossRef]
2. Viloria, A.; Rodado, D.; Lezama, O. Recovery of scientific data using Intelligent Distributed Data Warehouse. *Procedia Comput. Sci.* **2019**, *151*, 1249–1254. [CrossRef]
3. Zhen, Z.; Zhu, P.; Xue, Y.; Ji, Y. Distributed intelligent self-organized mission planning of multi-UAV for dynamic targets cooperative search-attack. *Chin. J. Aeronaut.* **2019**, *32*, 2706–2716. [CrossRef]
4. Jacobson, D.; Dickerman, L. Distributed intelligence: A critical piece of the microgrid puzzle. *Electr. J.* **2019**, *32*, 10–13. [CrossRef]
5. Jafari, M.; Xu, H. A biologically-inspired distributed fault tolerant flocking control for multi-agent system in presence of uncertain dynamics and unknown disturbance. *Eng. Appl. Artif. Intell.* **2019**, *79*, 1–12. [CrossRef]
6. Briones, A.; Mezquita, Y.; Garzón, J.; Prieto, J.; Corchado, J. Intelligent multi-agent system for water reduction in automotive irrigation processes. *Procedia Comput. Sci.* **2019**, *151*, 971–976. [CrossRef]
7. Kovalenko, I.; Tilbury, D.; Barton, K. The model-based product agent: A control oriented architecture for intelligent products in multi-agent manufacturing systems. *Control Eng. Pract.* **2019**, *86*, 105–117. [CrossRef]
8. Panteleev, M.G. Advanced Iterative Action Planning for Intelligent Real-Time Agents. *Procedia Comput. Sci.* **2019**, *150*, 244–252. [CrossRef]
9. Fu, Q.; Du, L.; Xu, G.; Wu, J.; Yu, P. Consensus control for multi-agent systems with distributed parameter models. *Neurocomputing* **2018**, *308*, 58–64. [CrossRef]
10. Pupkov, K.A.; Fadi, I. Collective Opinion Formation as a Set of Intelligent Agents to Achieve the Goal. *Procedia Comput. Sci.* **2019**, *150*, 216–222. [CrossRef]
11. Ghadimi, P.; Wang, C.; Lim, M.; Heavey, C. Intelligent sustainable supplier selection using multi-agent technology: Theory and application for Industry 4.0 supply chains. *Comput. Ind. Eng.* **2019**, *127*, 588–600. [CrossRef]
12. Salazar, L.; Mayer, F.; Schütz, D.; Heuser, B. Platform Independent Multi-Agent System for Robust Networks of Production Systems. *IFAC-PapersOnLine* **2018**, *51*, 1261–1268. [CrossRef]
13. Komenda, A.; Novák, P.; Pěchouček, M. Domain-independent multi-agent plan repair. *J. Netw. Comput. Appl.* **2014**, *37*, 76–88. [CrossRef]
14. Geanakoplos, J.; Karatzas, I.; Shubik, M.; Sudderth, W. Inflationary equilibrium in a stochastic economy with independent agents. *J. Math. Econ.* **2014**, *52*, 1–11. [CrossRef]
15. Qu, G.; Brown, D.; Li, N. Distributed greedy algorithm for multi-agent task assignment problem with submodular utility functions. *Automatica* **2019**, *105*, 206–215. [CrossRef]
16. Trujillo, M.A.; Becerra, H.M.; Gómez-Gutiérrez, D.; Ruiz-León, J.; Ramírez-Treviño, A. Priority Task-Based Formation Control and Obstacle Avoidance of Holonomic Agents with Continuous Control Inputs. *IFAC-PapersOnLine* **2018**, *51*, 216–222. [CrossRef]
17. Vorobiev, V. Inference algorithm for teams of robots using local interaction. *Procedia Comput. Sci.* **2018**, *123*, 507–511. [CrossRef]
18. An, B.; Liu, G.; Tan, C. Group consensus control for networked multi-agent systems with communication delays. *ISA Trans.* **2018**, *76*, 78–87. [CrossRef]
19. Wang, D.; Wang, Z.; Chen, M.; Wang, W. Distributed optimization for multi-agent systems with constraints set and communication time-delay over a directed graph. *Inf. Sci.* **2018**, *438*, 1–14. [CrossRef]
20. Zhao, L.; Yang, G. End to end communication rate-based adaptive fault tolerant control of multi-agent systems under unreliable interconnections. *Inf. Sci.* **2018**, *460–461*, 331–345. [CrossRef]
21. Torreño, A.; Sapena, Ó.; Onaindia, E. FMAP: A platform for the development of distributed multi-agent planning systems. *Knowl.-Based Syst.* **2018**, *145*, 166–168.

22. Tucnik, P.; Nachazel, T.; Cech, P.; Bures, V. Comparative analysis of selected path-planning approaches in large-scale multi-agent-based environments. *Expert Syst. Appl.* **2018**, *113*, 415–427. [CrossRef]
23. Zhen, Z.; Xing, D.; Gao, C. Cooperative search-attack mission planning for multi-UAV based on intelligent self-organized algorithm. *Aerosp. Sci. Technol.* **2018**, *76*, 402–411. [CrossRef]
24. Brambilla, M.; Ferrante, E.; Birattari, M.; Dorigo, M. Swarm Robotics: A Review from the Swarm Engineering Perspective. *Swarm Intell.* **2013**, *7*, 1–41. [CrossRef]
25. Liu, Y.; Song, R.; Bucknall, R.; Zhang, X. Intelligent multi-task allocation and planning for multiple unmanned surface vehicles (USVs) using self-organising maps and fast marching method. *Inf. Sci.* **2019**, *496*, 180–197. [CrossRef]
26. Yao, F.; Li, J.; Chen, Y.; Chu, X.; Zhao, B. Task allocation strategies for cooperative task planning of multi-autonomous satellite constellation. *Adv. Space Res.* **2019**, *63*, 1073–1084. [CrossRef]
27. Chen, X.; Zhang, P.; Du, G.; Li, F. A distributed method for dynamic multi-robot task allocation problems with critical time constraints. *Robot. Auton. Syst.* **2019**, *118*, 31–46. [CrossRef]
28. Mahi, M.; Baykan, O.; Kodaz, H. A new approach based on particle swarm optimization algorithm for solving data allocation problem. *Appl. Soft Comput.* **2018**, *62*, 571–578. [CrossRef]
29. Saxena, K.; Abhyankar, A. Agent based bilateral transactive market for emerging distribution system considering imbalances. *Sustain. Energy Grids Netw.* **2019**, *18*, 100203. [CrossRef]
30. Liang, H.; Kang, F. A novel task optimal allocation approach based on Contract Net Protocol for Agent-oriented UUV swarm system modeling. *Optik* **2016**, *127*, 3928–3933. [CrossRef]
31. Panescu, D.; Pascal, C. Holonic coordination obtained by joining the contract net protocol with constraint satisfaction. *Comput. Ind.* **2016**, *81*, 36–46. [CrossRef]
32. Momen, S. Ant-inspired Decentralized Task Allocation Strategy in Groups of Mobile Agents. *Procedia Comput. Sci.* **2013**, *20*, 169–176. [CrossRef]
33. Shehory, O.; Kraus, S. Methods for task allocation via agent coalition formation. *Artif. Intell.* **1998**, *101*, 165–200. [CrossRef]
34. Hruz, B.; Zhou, M. *Modeling and Control of Discrete-Event Dynamic Systems with Petri Nets and Other Tools*; Springer: Berlin/Heidelberg, Germany, 2007.
35. Gharbi, A.; Gharsellaoui, H.; Ben Ahmed, S. Multi-Agent control system. In Proceedings of the 9th International Conference on Software Engineering and Applications (ICSOFT-EA), Vienna, Austria, 29–31 August 2014; pp. 117–124.
36. Homepage of ITS-Tools. Available online: https://lip6.github.io/ITSTools-web (accessed on 8 November 2019).
37. Ben Noureddine, D.; Gharbi, A.; Ben Ahmed, S. A Social Multi-Agent Cooperation System based on Planning and Distributed Task Allocation: Real Case Study. In Proceedings of the ICSOFT, Porto, Portugal, 26–28 July 2018; pp. 483–493.
38. Weerdt, M.; Zhang, Y.; Klos, T. Distributed task allocation in social networks. In Proceedings of the 6th Automous Agents and Multiagent Systems (AAMAS 2007), Honolulu, HI, USA, 14–18 May 2007; pp. 500–507.
39. Available online: https://drive.google.com/open?id=1SYKk2CyxpAgq6o-n2VqZ7gOuwRPj3qJw (accessed on 10 May 2020).