A Social Multi-agent Cooperation System based on Planning and Distributed Task Allocation: Real Case Study

Dhouha Ben Noureddine[1,2], Atef Gharbi[1] and Samir Ben Ahmed[2]
[1]*LISI, INSAT, National Institute of Applied Science and Technology, University of Carthage, Tunis, Tunisia*
[2]*FST, University of El Manar, Tunis, Tunisia*

Keywords: Multi-agent System, Software Architecture, Distributed Task Allocation, Planning, Fuzzy Logic.

Abstract: In multi-agent systems, agents are socially cooperated with their neighboring agents to accomplish their goals. In this paper, we propose an agent-based architecture to handle different services and tasks; in particular, we focus on individual planning and distributed task allocation. We introduce the multi-agent planning in which each agent uses the fuzzy logic technique to select the alternative plans. We also propose an effective task allocation algorithm able to manage loosely coupled distributed environments where agents and tasks are heterogeneous. We illustrate our line of thought with a Benchmark Production System used as a running example in order to explain better our contribution. A set of experiments show the efficiency of our planning approach and the performance of our distributed task allocation method.

1 INTRODUCTION

In multi-agent systems (MAS), agents are assumed to be conscious of each other and need to cooperate with their neighboring agents to process tasks and achieve their goals. That is why accomplishing social cooperation is a crucial and important challenge in the software engineering fields, especially in the distributed artificial intelligence and MAS (Jennings et al., 1998). This challenge evolved with the progress of several applications, for example in wireless ad-hoc networks (Mejia et al., 2012), service-oriented MAS (Del Val et al., 2013), multi-robot system in healthcare facilities (Das et al., 2014), file sharing in P2P systems (Sun et al., 2004), social networks (Wei et al., 2013), etc. So, cooperation can provide appreciable convenience for these applications by promoting joint goals.

One of the elements that represents a real cooperation challenge is: the multi-agent task allocation problem (where multiple agents are used for task allocation). Using distributed task allocation methods for cooperating MAS is becoming increasingly interesting. Early researches used centralized approaches to generate a plan for cooperating all the agents by using a central server able to gather the whole system information. Other researches pointed out the distributed task allocation methods as a solution for interactive MAS, semantic web and grid technologies.

In this paper, we propose an agent-based architecture to manage tasks and control embedded systems at run-time. We firstly, introduce multi-agent planning in which each agent uses the fuzzy logic technique to select plans. The originality in this approach is that our agents evaluate plans based on their goal achievement satisfaction, which is represented as degrees of membership for each individual agent, their aggregate then represents the satisfaction of the overall goal. Proving that our approach performs better than the central planning processes in other systems. We then propose the distributed task allocation solution which is allowing agents to request help from neighbors, this would be done by allocating tasks to different agents who may be able each, to perform different subsets of those tasks. We use to highlight the performance of our solution using the provision of a benchmarking scenario.

The rest of the paper is organized as follows: Section 2 introduces the benchmark production system used in our approach. After that, a software architecture of MAS will be depicted in detail in Section 3. Section 4 defines our planning method and demonstrates the simulation and analysis about the quality and performance of our method. Then, a distributed task allocation approach is illustrated as well as its related experiments in Section 5. Finally, we discuss and conclude our work in Section 6.

Noureddine, D., Gharbi, A. and Ahmed, S.
A Social Multi-agent Cooperation System based on Planning and Distributed Task Allocation: Real Case Study.
DOI: 10.5220/0006831304490459
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 449-459
ISBN: 978-989-758-320-9

2 BENCHMARK PRODUCTION SYSTEM

We illustrate our contribution with a simple current example called *RARM* (Hruz and Zhou, 2007) which is implemented in our previous work ((Gharbi et al., 2015);(Ben Noureddine et al., 2016)). We begin informally with a description, but it will serve as an example for various formalism presented in this article. The *RARM* represented in the figure1 is composed of two inputs and one output conveyors, a servicing robotic agent and a processing-assembling center. Workpieces to be treated come irregularly one by one. The workpieces of type A are delivered via conveyor $C1$ and workpieces of the type B via the conveyor $C2$. Only one workpiece can be on the input conveyor. A robotic agent R transfers workpieces one after the other to the processing center. The next workpiece can be put on the input conveyor when it has been emptied by the robotic agent. The technology of production requires that firstly an A-workpiece is inserted into the center M and treated, then a B-workpiece is added to the center, and finally the two workpieces are assembled. Afterwards, 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 produced one.

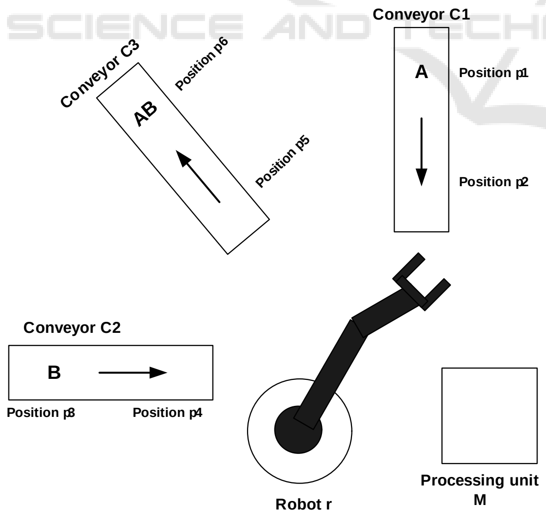

Figure 1: The benchmark production system RARM.

The system can be controlled using the following actuators:

1. move the conveyor $C1$ (act1);
2. move the conveyor $C2$ (act2);
3. move the conveyor $C3$ (act3);
4. rotate robotic agent (act4);
5. move elevating the robotic agent arm vertically (act5);
6. pick up and drop a piece with the robotic agent arm (act6);
7. treat the workpiece (act7);
8. assembly two pieces (act8).

The robot-like agent receives the information from the sensors as follows:

1. Is there an A-workpiece at the extreme end of the position $p1$? (sens1)
2. Is $C1$ in its extreme left position? (sens2)
3. Is $C1$ in its extreme right position? (sens3)
4. Is there an A-workpiece at the unit M? (sens4)
5. Is $C2$ in its extreme left position? (sens5)
6. Is $C2$ in its extreme right position? (sens6)
7. Is there a B-workpiece at the extreme end of the position $p3$? (sens7)
8. Is there a B-workpiece at the unit M? (sens8)
9. Is $C3$ in its extreme left position? (sens9)
10. Is $C3$ in its extreme right position? (sens10)
11. Is there a AB-workpiece at the unit M? (sens11)
12. Is the robotic agent arm in its lower position? (sens12)
13. Is the robotic agent arm in its higher position? (sens13)

3 SOFTWARE AGENT ARCHITECTURE

We propose an agent-based architecture to control embedded systems at run-time. The agent checks the environment's evolution and reacts when new events occur.

3.1 Formal Specification

To describe the dynamic behavior of an intelligent agent that dynamically controls the plant, we use the state machine which can be defined as a state machine whose states, inputs and outputs are enumerated. The state machine is a graph of states and transitions. It treats the several events that may occur by detecting them and responding to each one appropriately. We define a state machine SM_i as the following:

$SM_i = (S_i, S_{i0}, I_i, O_i, Precond_i, Postcond_i, t_i)$

- $S_i = \{s_{i1}, .., s_{ip}\}$: the set of states;
- S_{i0} the initial state;
- $I_i = \{I_{i1}, .., I_{im}\}$: the input events;
- $O_i = \{O_{i1}, .., O_{ik}\}$: the output events;
- $Precond_i$: the set of conditions to be verified before the activation of a state;
- $Postcond_i$: the set of conditions to be verified once a state is activated;
- $t_i : S_i \times I_i \rightarrow S_i$: the transition function.

We propose a conceptual model for a state machine in Figure 2 where we define the classes *State machine*, *State*, *Transition*, *Event* and *Condition*. The *State Machine* class contains a certain number of *State* and *Transition* classes. This relation is represented by a composition. The *Transition* class is double linked to the *State* class because a transition is considered as an association between two states. Each transition has an event that is considered as a trigger to fire it and a set of conditions to be verified. This association between the *Transition* class and the two classes *Event* and *Condition* exists and it's modeled by the aggregation relation.

3.2 Conceptual Architecture for MAS

We propose a generic architecture for MAS depicted in Figure 3. This architecture consists of the following parts: *(i)* the Event Queue to save different input events that may take place in the system, *(ii)* the software agent that reads an input event from the Event Queue and reacts as soon as possible, *(iii)* the set of state machines such that each one is composed of a set of states, *(iv)* each state represents a specific information about the system. The agent, based on state machines, determines the new system's state to execute according to event inputs and also conditions to be satisfied. This solution has the following characteristics: *(i)* The agent design is general enough to cope with various kinds of embedded-software based application. Therefore, the agent is uncoupled from the application and from its components. *(ii)* The agent is independent of the state machines: it permits to change the structure of the state machine (add state machines, change connections, change input events, and so on) without having to change the implementation of the agent. This ensures that the agent continues to work correctly even in case of modification of state machines.

In the following algorithm, the symbol Q is an event queue which holds incoming event instances, *ev* refers to an event input, S_i represents a State Machine, and $s_{i,j}$ a state related to a State Machine S_i. The internal behavior of the agent is defined as follow:

1. the agent reads the first event *ev* from the queue Q;
2. searches from the top to the bottom in the different state machines;
3. within the state machine SM_i, the agent verifies if *ev* is considered as an event input to the current state $s_{i,j}$ (i.e. ev $\in$ I related to $s_{i,j}$). In this case, the agent searches the states considered as successor for the state $s_{i,j}$ (states in the same state machine SM_i or in another state machine SM_l);
4. the agent executes the operations related to the different states;
5. repeats the same steps (1-4) until no more event exists in the queue to be treated.

Algorithm 1: GenericBehavior.

```
1: while $Q.length() > 0$ do
2:     ev ← $Q.Head()$
3:     for each state machine $SM_i$ do
4:         $s_{i,j} \leftarrow currentState_i$
5:         if ev $\in I(s_{i,j})$ then
6:             for each state $s_{i,k} \in next(s_{i,j})$
7:                 such that $s_{i,k}$ related to $S_i$ do
8:                 if execute($s_{i,k}$) then
9:                     $currentState_i \leftarrow s_{i,k}$
10:                    break
11:            for each state $s_{l,k} \in next(s_{i,j})$
12:                such that $s_{l,k}$ related to $S_l$ do
13:                if execute($s_{l,k}$) then
14:                    $currentState_l \leftarrow s_{l,k}$
15:                    break
```

First of all, the agent evaluates the pre-condition of the state $s_{i,j}$. If it is false, then the agent exits, else the agent determines the list of tasks to be executed. Finally, it evaluates the post-condition of the state $s_{i,j}$ and generates errors whenever it is false.

Algorithm 2: Function execute($s_{i,j}$): boolean.

```
1: if $\neg s_{i,j}.PreCondition$ then
2:     return false
3: else
4:     listTask ← getInfo($s_{i,j}$.info)
5:     for each task $t \in$ listTask do
6:         t.execute()
7:     if $\neg s_{i,j}.PostCondition$ then
8:         Generate error
9:     return true
```

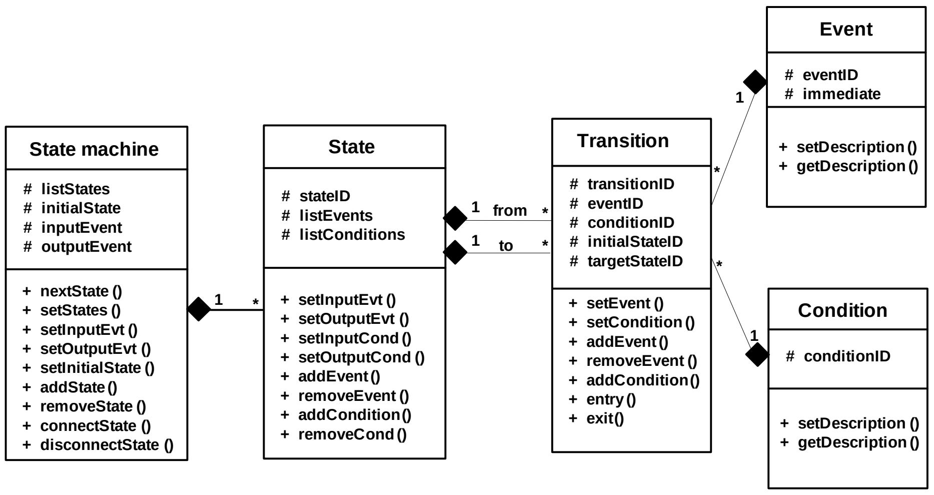

Figure 2: The Meta-model state machine.

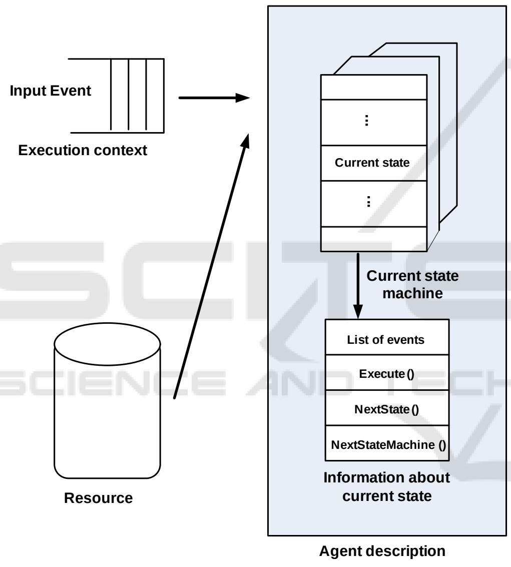

Figure 3: The internal agent behavior.

4 MULTI-AGENT PLANNING

4.1 Policy

To deal with the uncertainty and hesitancy problems, we use conjunction operators for *fuzzy relationship* as we mentioned before in (Ben Noureddine et al., 2016). Here, it is very natural and reasonable to apply a membership function in fuzzy mathematics to evaluate the *satisfaction degree* of the plan (list of events). G_s is the problem goal, it is the union of individual goals of all robot-like agents denoted by Sg_i, which are flexible propositions.

$$G_s = \cup_{i=1..n} Sg\_i \tag{1}$$

These goals can be achieved with a certain satisfaction degree. The form of a flexible proposition is (ρ ϕ_1, ϕ_2,..., ϕ_j κ_i), where $\phi_i \in \phi$ and κ_i are elements of totally ordered set, K, which represents the truth degree of the proposition. K is composed of a finite number of membership degrees, $k_\uparrow$, k_1,..., $k_\downarrow$, where $k_\uparrow \in$ K and $k_\downarrow \in$ K, representing respectively total falsehood and total truth. When dealing with a flexible proposition with a truth value of $k_\uparrow$ or $k_\downarrow$, the boolean style $\neg$ (ρ ϕ_1, ϕ_2,..., ϕ_j) or (ρ ϕ_1, ϕ_2,..., ϕ_j) is adopted. The flexible proposition (Miguel et al., 2000) is described by a *fuzzy relation*, R, which is defined as a membership function $\mu_R(.)$: $\Phi_1 * \Phi_2 * ... * \Phi_j \rightarrow$K, where $\Phi_1 * \Phi_2 * ... * \Phi_j$ is the Cartesian product of the subsets of Φ in the current proposition state. In other words, if every agent achieves its individual goals with a certain satisfaction degree, the public goals of the problem are achieved. The satisfaction degree of a multi-agent flexible planning problem is defined as the conjunction of the satisfaction degrees of each action and goal.

$$\mu_G = \wedge_{i=1..n} \mu_R(i) \tag{2}$$

The function μ_G indicates how well a given plan is satisfying and can be considered as a value between 0 and 1, 1 stands for completely satisfied and 0 stands for not satisfied at all. In our approach, each plan alternative is associated with a satisfaction degree. That means each value is the metric that provides the means to select a plan among different alternatives. Having the ameliorated mean values calculated, the plan alternative along with these values are sent to the current state machine. The plan alternative, the need, the goal and corresponding values reach the decision-making mechanism first. The decision-making mechanism uses these values to compare the satisfaction degrees for each plan alternative to find the most satisfactory one. The one with the highest satisfaction degree is considered as the most satisfactory plan alternative.

Running Example

Giving (S, A, G_s) where $S = \{s_i|$i=1...n $\}$ is a set of states, $A = \{Ci_left, Ci_right, Ri_left, Ri_right, take_i, load_i, put_i, process_i|$i=1...n $\}$ is a set of actions, and G_s is the problem goal. *if* s_0 *and g = {workpiece in the processing unit}. Let:*

- $\pi_0 : (C2\_left, take_2, load_2, process_2)$
- $\pi_1 : (load_1, put_1, process_1, C1\_right)$
- $\pi_2 : (C1\_left, take_1, load_1, put_1, process_1, C1\_right)$

We solve multi-agent planning problems by distributed flexible constraint satisfaction problem (CSP) technique (Miguel and Giret, 2008) and make a trade-off between plan length and the compromise decisions made. The quality of a plan is measured by its satisfaction degree and its length, where the shorter of two plans is better under the same satisfaction degrees. In this example, the definitions of K and L are: $K = \{k_\uparrow, k_1, k_2, k_\downarrow\}$, $L = \{l_\uparrow, l_1, l_2, l_\downarrow\}$. The multi-agent planning problem is helpful to robot-like agents like in this example. If any actions $\in \{Ci_left, Ci_right, Ri_left, Ri_right, take_i, load_i, put_i, process_i|$i=2...n $\}$ will damage the plan, leading to a satisfaction degree l_2, any plan not beginning with $C1\_left$ will result in a satisfaction degree l_2 because it is not applicable to s_0, and when any action is applicable to s_0 and the resulting state is a goal state then the result will be a satisfaction l_1. We may obtain more than one plan with different satisfactions by different compromises as shown in Table 1 and 2.

Table 1: A plan of 4 steps with satisfaction l_2.

Action No	Action	Satisfaction
1	$C2\_left$	l_2
2	$take_2$	l_2
3	$load_2$	l_2
4	$process_2$	l_2

Then π_0 is not a solution because although it is applicable to s_0, the resulting state is not a goal state; π_1 is not a solution because it's not applicable to s_0; π_2 is the most appropriate solution.

4.2 Evaluation

In this paper, we propose an architecture that integrates conjunction operators for fuzzy relationship to

Table 2: A plan of 6 steps with satisfaction $l_\uparrow$.

Action No	Action	Satisfaction
1	$C1\_left$	$l_\uparrow$
2	$take_1$	$l_\uparrow$
3	$load_1$	$l_\uparrow$
4	put_1	$l_\uparrow$
5	$process_1$	$l_\uparrow$
6	$C1\_right$	$l_\uparrow$

evaluate the satisfaction degree of the plan, this reliable technique can offers robust and reliable solutions for the planning problem. The aim of the experiment we have done was to find out how our software architecture having different facilities and abilities can perform, simulate robot-like agent's behavior, and how the performance of the robotic agents was influenced by varying their satisfaction degree and the plan length. In order to show the feasibility of our approach, we present experimental results on preliminary tests focusing on the analysis of the planning performance using the satisfaction degree by simulating *RARM*. Figure 4 shows the results obtained when running our architecture. Therefore, we compare their performance on a set of plans for the *RARM* state-transitions. Since the second plan of 2 steps with maximum satisfaction degree l_2, the fifth plan of 15 steps with satisfaction $l_\uparrow$. So, it is often possible to find short, satisfactory plans quickly during the decision-making mechanism. The quality of a plan is its satisfaction degree combined with its length, where the shorter of two plans with equivalent satisfaction degrees is better.

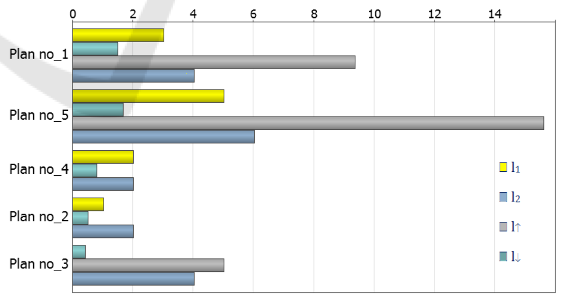

Figure 4: Experimental results collected plan length and the satisfaction degree.

These results are indicative of the ability to dynamically treat operating conditions among different conveyors, a service robot and a treating-assembling center over time, plays a critical role in the actions selection during the planning. Further, making decisions also affect the choice of process flexibility. According to the software architecture, breakdown of an individual robotic agent will have little effect on the whole team, because of the existence of the technique satisfaction degree of the plan which simplify the choice of the most satisfactory plan alternative. When the architecture is applied to multi-robot system *RARM*, there are several important functions that have been performed. The associated issues are planning and intelligent decision making.

5 DISTRIBUTED TASK ALLOCATION APPROACH

5.1 Problem Definition

We describe in this section the social task allocation problem which can be defined as an agent not adequate to complete a task by itself and it needs the cooperation from other agents to achieve an action or service. We denote $A = \{a_1,..., a_m\}$ a set of agents, that require resources to achieve tasks; and we denote $R = \{r_1,..., r_k\}$ a set of resources types available to A. Each agent $a \in A$ controls a fixed quantity of resources for each resource type in R, which is defined by a resource function: $rsc : A \times R \rightarrow N$. Moreover, we assume agents are connected by a *social network* as discussed before in (Ben Noureddine et al., 2017).

Definition 1 (Social Network). *An agent social network $SN = (A,AE)$ is an undirected graph, where A is a set of agents and AE is a set of edges connecting two agents a_i and a_j significant that it exists a social connection between these two agents.*

We define $T = \{t_1, t_2,..., t_n\}$ a set of needed tasks at such an agent social network. Each task $t \in T$ is then defined by a 3-tuple $\{u(t), rsc(t), loc(t)\}$, where $u(t)$ is the utility gained if task t is accomplished, rsc: $T \times R \rightarrow N$ is the resource function that specifies the amount of resources required for the accomplishment of task t and loc: $T \rightarrow A$ is the location function that defines the locations (i.e., agents) at which the tasks arrive in the social network. An agent a is the location of a task t, i.e. $loc(t) = a$, is called this task *manager*. Each task $t \in T$ needs some specific resources from the agents to complete the task. A task allocation is defined as the exact assignment of tasks to agents.

A task plan of agent consists of a list of actions to be taken in order. Each action is an attempt to acquire a particular resource, by asking the agent associated with that resource for permission to use the resource. A task agent builds a plan by maximizing the satisfaction degree described in the section 4. At each timestep, a task agent performs the action presently prescribed by its plan. It does this by contacting the agent associated with the targeted resource, and asking it whether it may take the resource.

Definition 2 (Multi-agent Planning Problem). *We denote π a plan which is described by a 5-tuple $\{T, P(t), E(t), G, \mu_t\}$, where T is a set of tasks as mentioned above, $P(t)$ is the set of action (task) preconditions, $E(t)$ is the set of task effects, G is the problem goal and μ_t the satisfaction degree of a multi-agent flexible planning problem introduced in section 4.*

Definition 3. *Each agent $a \in A$ is composed of 4-tuple $\{AgentID(a), Neig(a), Resource(a), State(a)\}$, where $AgentID(a)$ is the identity of agent a, $Neig(a)$ is a set indicating the neighbors of agent a, $Resource(a)$ is the resource which agent a contains, and $State(a)$ demonstrates the state of agent which will be described in the following subsection.*

Definition 4 (Task Allocation). *We consider a set of tasks $T = \{t_1, t_2,..., t_n\}$, a set of agents $A = \{a_1,..., a_m\}$, a set of plans $\pi = \{\pi_1,..., \pi_m\}$, and a set of resources $R = \{r_1,..., r_k\}$ in a social network SN, a task allocation is a mapping $\phi : T \times A \times R \times \pi \rightarrow SN$.*

5.2 The Principle of Distributed Task Allocation

To guarantee a coherent behavior of the whole distributed system, we define the following idea: we suppose that $Neig(a_i)$ stores only directly linked neighboring agents of agent a_i where at each timestep, these task neighboring agents perform the action presently prescribed by their most satisfying tasks. The task neighboring agents do this by contacting the agent associated with the targeted resource, and asking it whether it may take the resource.

To control system in our multi-agent architecture, we introduce three types of agents like in (Ben Noureddine et al., 2017): *Manager* is the agent which requests help for its task, the agent which accepts and performs the announced task is called *Participant* and *Mediator* is the agent that receives another agent's commitments for assistance to find participants.

We propose a software multi-agent architecture to handle distributed task allocation. To guarantee a coherent behavior of the whole distributed system, we define the following idea: we suppose that $Neig(a_i)$ stores only directly linked neighboring agents of agent a_i.

We define as in (Ben Noureddine et al., 2017) three states *States = {Busy, Committed, Idle}* in a complex adaptive system and an agent can be only in one of the three states at any timestep. When an agent is a *Manager* or *Participant*, the state of that agent is *Busy*. When an agent is a *Mediator*, the agent is in *Committed* state. An agent in *Idle* state is available and not assigned or committed to any task.

For efficient task allocation, it is supposed that only an *Idle* agent can be assigned to a new task as a *Manager* or a partial fulfilled task as a *Participant*, or *Committed* to a partial fulfilled task as a *Mediator*. A partial fulfilled task is a task, for which a full group is in formation procedure and has not yet formed.

We present our approach which describes an interactive model between agents detailed as follows:

- When a *Manager* denoted by A_{Mn} ought to apply distributed task allocation, it then sends resource announce messages described formally as *ResAnnounceMess* = <AgentID(A_{Mn}), TaskID(t_{Mn}), Resource(t_{Mn})>, to all its neighbors;
- These neighboring agents receiving the *ResAnnounceMess* sent by A_{Mn},
  - **If** (*state(neighboring agent) = Idle*) **Then** the neighboring agent A_j applies the single-agent planning to select the most appropriate tasks and then proposes with information about the types of resources it contains, the execution time, the utility and the identities of them, namely *ProposeMess* = <AgentID(A_j), Resource(A_j), Execute(A_j), Utility(A_j)>.
  - **Else** (*state(neighboring agent) = Busy*) the neighboring agent A_j refuses and sends the following message *RefuseMess* = <AgentID(A_j)>.
- After answering the resource announce messages sent by A_{Mn}
  - **If** (A_{Mn} is satisfied with many resource proposals of the neighbor) **Then** A_{Mn} will pick the agent having the highest utility, denoted by A_j, and the state of A_j will be changed to *Busy*. In case the A_{Mn} finds many agents having the highest utility then it chooses the agent A_j proposing the least execution time with a most appropriate task.
  - **Else** the A_{Mn} is satisfied with only one resource of the neighbor, then the A_{Mn} will choose this agent without any utility consideration.

  A_{Mn} sends a contract to the chosen agent A_j composed of 4-tuple, *Contract* = <AgentID(A_{Mn}), AgentID(A_j), TaskID(t_{Mn}), Resource(A_{Mn})>.
- After obtaining the answer from its different cooperative neighbors, A_{Mn} then compares the available resources from its neighbors, i.e. Resoneig(A_{Mn}), with the resources required for its task t_{Mn}, namely rsc(t_{Mn}). (Here, Resoneig(A_{Mn}) = $\bigcup_{A_j \in Neig(A_{Mn})} Resource(A_j)$). This comparison would result in one of the following two cases:
  1. **If** (rsc(t_{Mn}) $\subseteq$ Resoneig(A)) **Then** A_{Mn} can form a full group for task t_{Mn} directly with its neighboring agents which they apply the policy of single-agent planning.
  2. **Else** (Resoneig(A) $\subset$ rsc(t_{Mn})), in this condition, A_{Mn} can only form a partial group for task t_{Mn}. It then commits the task t_{Mn} to one of its neighbors. The commitment selection is based on the number of neighbors each neighbor of A_{Mn} maintaining. The more neighbors an agent has, the higher probability that agent could be selected as a $Mediator$ agent to commit the task t_{Mn}.
- After selection, A_{Mn} commits its partial fulfilled task t_{Mn} to the $Mediator$ agent, denoted as A_{Md}. A commitment consists of 4-tuple, *Commitment* = <AgentID(A_{Mn}), AgentID(A_{Md}), TaskID(t_{Mn}), rsc(t_{Mn})^1 >, where rsc(t_{Mn})^1 is a subset of rsc(t_{Mn}), which contains the unfulfilled required resources. Afterwards, A_{Md} subtracts 1 from N_{max} and attempts to discover the agents with available resources from its neighbors. If any agents satisfy resource requirement, A_{Md} will send a response message, *RespMess*, back to A_{Mn}. The agent A_{Mn} then directly makes contract with the agents which satisfy the resource requirement and have an appropriate plan of tasks. If the neighboring agents of A_{Md} cannot satisfy the resource requirement either, A_{Md} will commit the partial fulfilled task t_{Mn} to one of its neighbors again.
- This process will continue until all of the resource requirements of task t_{Mn} are satisfied, or the N_{max} reaches 0, or there is no more $Idle$ agent among the neighbors. Both of the last two conditions, i.e. N_{max} = 0 and no more $Idle$ agent, demonstrates the failure of task allocation. In these two conditions, A_{Mn} disables the assigned contracts with the $Participant$ s, and the states of these $Participant$ are reset to $Idle$.
- When finishing an allocation for one task, the system is restored to its original status and each agent's state is reset to $Idle$.

```
    Algorithm Communicate()
begin
switch (role)
case Manager:
    switch (step)
   case 0: // send a request to all neighbors Agents
        for j = 1 to NbA do
            send(ResAnnounceMes(Agents[j]));
        step++;
        break;
   case 1: // Receive accept/refusal from neighbors Agents
        reply ← receive();
        if (reply = ProposeMess(Agents[j]))
```

```
            send(Contract(Agents[j]));
            Res=Res+Res(Agents[j]);
        Nb++;
        **if** (Nb = NbA )
            **if** (Res = Resource)
                step ← 4; (execute step)
            **else**
                step ++;
        **break**;
    **case** 2: // choose the Mediator Agent
        Max ← Neig(Agents[1])
        Mediator ← 1
        **for** j = 2 **to** NbA **do**
            if Neig(Agents[j]) > Max ;
                Mediator ← j
        send(Commitment(Agents[Mediator]));
        step++;
    **case** 3: // wait the response from the Mediator Agent
        reply ← receive();
        **if** (reply = ResMess(Agents[Mediator]))
            **for** j = 1 **to** list(Agents[Mediator]) **do**
                send(Contract(Agents[j]));
            Res += Res(list(Agents[Mediator]))
            **if** (Res = Resource)
                step ← 4; (execute step)
            **else**
                step ← 5; (cancel step)
        **break**;
    **case** 4:
     **for** j = 1 **to** length(list(Agents)) **do**
        send(Execute(list(Agents[j]));
     step ← 0;
     role ← participant;
     **break**;
    **case** 5:
     step ← 0;
     role ← participant;
     **break**;
     **End switch**
**case** Mediator:
     **switch** (step)
    **case** 0: // wait a message from the Manager Agent
     reply ← receive();
     **if** (reply = Commitment)
        step++;
     **break**;
    **case** 1: // send a request to all neighbors Agents
        **for** j = 1 **to** NbA **do**
            send(ResAnnounceMes(Agents[j]));
        step++;
        **break**;
    **case** 2: // Receive accept/refusal from neighbors Agents
     reply ← receive();
     **if** (reply = ProposeMess(Agents[j]))
        Res=Res+Res(Agents[j]);
     Nb++;
     **if** (Nb = NbA )
        step ← 3; (inform the manager)
     **break**;
    **case** 3: // inform the manager Agent
     send(ResMess(Manager));
        **break**;
     **End switch**
**case** Participant:
     **switch** (step)
    **case** 0: // wait a message from the Manager Agent
     reply ← receive();
     **if** (reply = ResAnnounceMes(Manager))
        **if** (state = IDLE )
            send(ProposeMess(Manager));
            step++;
        **else**
            step ← 0;
     **break**;
    **case** 1: // wait a CONTRACT from the Manager Agent
     reply ← receive();
     **if** (reply = CONTRACT(Manager))
        state = BUSY
        step++;
     **break**;
    **case** 2: // Receive accept/refusal from neighbors Agents
     reply ← receive();
     **if** (reply = Execute(Manager))
        ExcuteTask();
     state = IDLE
     step← 0;
     **break**;
     **End switch**
**end**
```

5.3 Experiments

In order to strengthen the validity and to demonstrate the quality of our approach, we have simulated our distributed task allocation algorithm in different networks. To test the efficiency of our algorithm, we compare it with the Greedy Distributed Allocation Protocol (*GDAP*) (Weerdt et al., 2007). In this subsection, we briefly define GDAP. Then, we introduce the experiment environment' settings. And we depict in the last sub-subsection the results and the relevant analysis.

5.3.1 Greedy Distributed Allocation Protocol

GDAP is selected to handle task allocation problem in agent social networks. It's described briefly in (Weerdt et al., 2007) as follows: All *Manager* agents $a \in A$ try to find neighboring contractors (the same as *Participant* in this paper) to help them do their tasks $T_a = \{t_i \in T | loc(t_i) = a\}$. They start offering the most efficient task. Among all tasks offered, contractors select the one having the highest efficiency and send a bid to the related manager. A bid consists of all the resources the agent is able to supply for this task. If sufficient resources have been offered, the manager selects the required resources and informs all contractors of its choice. When a task is allocated, or when a manager has received offers from all neighbors but still cannot satisfy its task, the task is removed from its task list. And this is the main disadvantage of GDAP that it only relies on neighbors which may cause several unallocated tasks due to limited resources, that is exactly what our approach tries to solve.

5.3.2 Experimental Settings

We have been implementing our distributed task allocation algorithm and (GDAP) in JAVA and we have been testing them. There are two different settings used in our experiment. The first setup has been done in the *Small-world networks* in which most neighbors of an agent are also connected to each other. The second setup has been done in the *Scale free networks*.
Setting 1: we consider the number of agents 40, the number of tasks 20, the number of different resource's types 5, the average number of resources required by each task 30 and the average number of resources needed by each tasks 30. We assume that tasks are distributed uniformly on each *IDLE* agent and resources are normally allocated to agents. The only changing variable in this setting is the average number of neighbors. This setting intends to represent the influence of neighbors' number on the performance of both our algorithm and GDAP.
Setting 2: we fix the average number of neighbors at 10. We consider that the number of agents increases and varies from 100 to 2000. We fix the ratio between the number of agents and tasks at 5/3 and the resource ratio at 1.2. The number of different resource types is 20 and the average resource requirement of tasks is 100. The tasks are uniformly distributed. This setting is defined to demonstrate the scalability of both our algorithm and GDAP in a large scale networks with a fixed average number of neighbors.
The algorithms have been evaluated according to two criteria in this experiment; the *Utility Ratio* and the *Execution Time*, where:

$$UtilityRatio = \frac{\sum Successful - completed - tasks}{Total - of - tasks} \quad (3)$$

The unit of Execution Time is millisecond. For simplicity, we suppose that once a task has been allocated to a Participant, the Participant would successfully finish this task without failure.

5.3.3 Experiment Results and Analysis

Experiment Results and Analysis from Setting 1: we would like to test in this experiment the influence of different average number of neighbors on both algorithms. We notice in Figure 5 that the Utility Ratio of our algorithm in different networks is more reliable than the GDAP algorithm. For the reason that the distribution of tasks in GDAP is only depending on the Manager neighbors, contrary to ours, in the case of need, other agents are allocated (i.e. not only the neighbors).

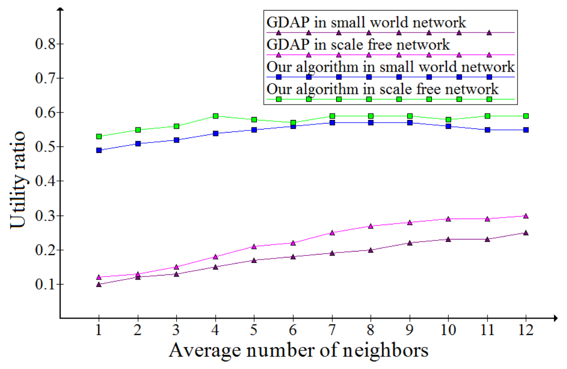

Figure 5: The Utility ratio of the GDAP and our algorithm depending on the average number of neighbors in different type of networks.

We can mention another factor to compare both approaches which is the network type. The results of GDAP in a small world network is higher than in a scale free network, and this could be explained by the fact that the most agents have a very few neighbors in the small network. Opposingly to that , in the scale free network when the average number of neighbors increases, the GDAP performance decreases. Which leads to say that this factor does not affect the performance of our algorithm as we take into consideration enough neighbors to obtain satisfactory resources for processing its tasks without reallocating tasks further.

Figure 6 presents the Execution Time of two algorithms in different networks depending on the average number of neighbors. The Execution Time of our algorithm is higher than that of GDAP since during execution, the agents in our algorithm reallocate tasks when resources from neighbors are unsatisfying. Furthermore, we note that the results of GDAP in a small world network is higher than in a scale free network, but compared to our algorithm are still lower and this is because it considers only neighbors which could decrease the time and communication cost during task allocation process.
Experiment Results and Analysis from Setting 2: we would like to test the scalability of both GDAP and our algorithm in different large network scales like applications running on the internet. The Figure 7 presents the Utility Ratio of GDAP which is constantly descending while that of our algorithm can save the stability and it is higher than GDAP with the increase of number of agents and simultaneously the number

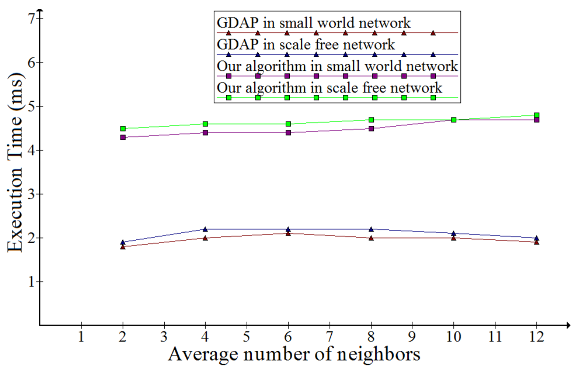

Figure 6: The Execution time in millisecond of the GDAP and our algorithm depending on the average number of neighbors in different type of networks.

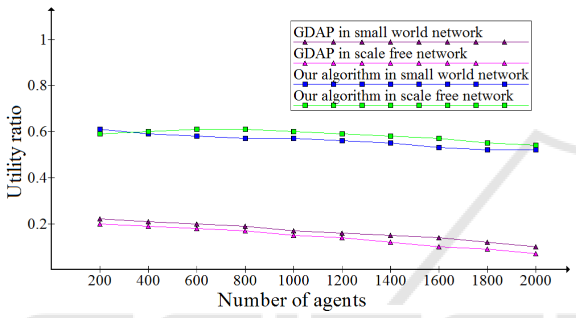

Figure 7: The Utility ratio of the GDAP and our algorithm depending on the number of agents in different type of networks.

of tasks in a large network scale. In fact, we can explain this by the proportional rise of the network scale, the tasks and the resource types.

Moreover the condition in small world network is better than that in scale free network. And this is justified by the same reason described above, that in scale free network, several agents only have a few neighbors which is not good for GDAP. Compared with GDAP, our algorithm is more competitive and it is favoured from task reallocation.

Figure 8 presents the Execution Time of our algorithm and GDAP in different network types. GDAP spends less time when there are more agents in the network. This is because there are more tasks despite the average number of neighbors is fixed. Accordingly, more reallocation steps cannot be avoided towards allocating these tasks, that leads to soaring in time and overhead communication. Furthermore, the graphs show that the GDAP and our algorithm almost behaves linearly and the time consumption of GDAP keeps a lower level than ours. This can be supposedly interpreted that GDAP only relies on neighboring agents.

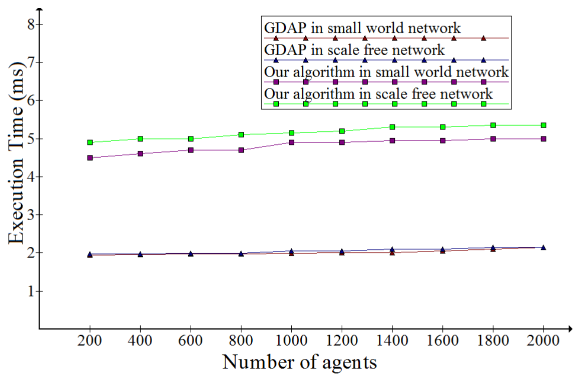

Figure 8: The Execution time in millisecond of the GDAP and our algorithm depending on the number of agents in different type of networks.

6 CONCLUSION

Cooperation is a key process for multi-agent system's research and, as such, it has received a considerable amount of attention in literature. In this paper we propose an agent-based architecture to manage services, handle and control embedded systems at runtime to perform self-adaptation. An important originality of our work is the integration of the fuzzy logic technique to select plans in the planning phase, in which a great attention is payed in this paper. All the results are applied in this phase to a particular benchmark production system. We also put forward a distributed method to solve multi-task allocation problems in the MAS. Although our approach overcomes many dilemmas, which exist in some current related works, due to its decentralization and reallocation features, it still has several deficiencies. They will be faced in near future work, that will focus on assessing the mechanism's ability to deal with larger state action spaces than the one exemplified in this paper and review the performance benefits compared to the heavier-weight alternative solutions.

REFERENCES

Ben Noureddine, D., Gharbi, A., and Ben Ahmed, S. (2016). An approach for multi-robot system based on agent layered architecture. *International Journal of Management and Applied Science (IJMAS)*, 2(12):135–143.

Ben Noureddine, D., Gharbi, A., and Ben Ahmed, S. (2017). Multi-agent deep reinforcement learning for task allocation in dynamic environment. In *In the Proceedings of the 12th International Conference on Software Technologies (ICSOFT'17)*, pages 17–26.

Das, G. P., McGinnity, T. M., Coleman, S. A., and Behera, L. (2014). A distributed task allocation algorithm for a multi-robot system in healthcare facilities. *Springer Journal of Intelligent and Robotic Systems*, 84:1–26.

Del Val, E., Rebollo, M., and Botti, V. (2013). Promoting cooperation in service-oriented mas through social plasticity and incentives. *Journal of Systems and Software*, 86:520–537.

Gharbi, A., Ben Noureddine, D., and Ben Hlima, N. (2015). Building multi-robot system based on five capabilities model. In *In the Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'15)*, pages 270–275.

Hruz, B. and Zhou, M. (2007). Modeling and control of discrete-event dynamic systems with petri nets and other tools. pages 67–78.

Jennings, N. R., Sycara, K., and Wooldridgse, M. (1998). A roadmap of agent research and development. *Autonomous agents and multi-agent systems*, 1:7–38.

Mejia, M., Peña, N., Muñoz, J. L., Esparza, O., and Alzate, M. (2012). Decade: Distributed emergent cooperation through adaptive evolution in mobile ad hoc networks. *Ad Hoc Networks*, 10:1379–1398.

Miguel, I. and Giret, A. (2008). Feasible distributed csp models for scheduling problems. *Engineering Applications of Artificial Intelligence*.

Miguel, I., Jarvis, P., and Shen, Q. (2000). Flexible graphplan. In *In the Proceedings of the 14th European Conference on Artificial Intelligence, Berlin*, pages 4506–4514.

Sun, Q., Garcia-Molina, H., and Ben Ahmed, S. (2004). Slic: A selfish linkbased incentive mechanism for unstructured peer-to-peer networks. In *In the Proceedings of the 24th International Conference in Distributed Computing Systems*, pages 506–515.

Weerdt, M., Zhang, Y., and Klos, T. (2007). Distributed task allocation in social networks. In *In the Proceedings of the 6th Autonomous Agents and Multi-agent Systems (AAMAS 2007), Honolulu, Hawaii, USA*, pages 500—-507.

Wei, G., Zhu, P., Vasilakos, A. V., Mao, Y., Luo, J., and Ling, Y. (2013). Cooperation dynamics on collaborative social networks of heterogeneous population. *IEEE Journal Selected Areas in Communications*, 31:1135–1146.