FUZZY LOGIC MULTI-AGENT SYSTEM

Atef GHARBI$^1$ and Samir BEN AHMED$^2$

$^1$Department of Computer Engineering, INSAT, Tunis, Tunisia
atef.elgharbi@gmail.com

$^2$Department of Computer Engineering, FST, Tunis, Tunisia
samir.benahmed@fst.rnu.tn

ABSTRACT

The paper deals with distributed planning in a Multi-Agent System (MAS) constituted by several intelligent agents each one has to interact with the other autonomous agents. The problem faced is how to ensure a distributed planning through the cooperation in our multi-agent system.

To do so, we propose the use of fuzzy logic to represent the response of the agent in case of interaction with the other. Finally, we use JADE platform to create agents and ensure the communication between them.

A Benchmark Production System is used as a running example to explain our contribution.

KEYWORDS

Multi-Agent System, Distributed Planning, Fuzzy Logic, JADE

1. INTRODUCTION

While Multi-Agent System (MAS) is a concept mainly used in research [23], by adapting it we must face various problems, some of which are serious enough to place the utility of MAS in the doubt. Since we wish to use the MAS in large scales, concurrent systems, and since we wish to address not very frequent, but demanding problems [24], MAS can become arbitrarily complex if MAS can not provide guarantees which help to order the system and ensure the progression of the total application.

We can not pretend the unicity nor the exactitude of an agent definition, however the most adapted one presented by [1] where an agent is defined as a physical or virtual entity (i) which is capable of acting in an environment; (ii) which can communicate directly with other agents; (iii) which is driven by a set of tendencies (in the form of individual objectives or of a satisfaction/survival function which it tries to optimize); (iv) which possesses resources of its own; (v) which is capable of perceiving its environment (but to a limited extent); (vi) which has only a partial representation of its environment (and perhaps none at all); (vii) which possesses skills and can offer services; (ix) which may be able to reproduce itself; (ix) whose behaviour tends towards satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representation and the communications it receives.

In MAS, distributed planning is considered as a very complex task [3], [18]. In fact, distributed planning ensures how the agents should plan to work together, to decompose the problems into
subproblems, to assign these subproblems, to exchange the solutions of subproblem, and to synthesize the whole solution which itself is a problem that the agents must solve [19, 20, 4]. The actions of the other agents can induce a combinatorial explosion in the number of possibilities which the planner will have to consider, returning the space of research and the size of solution exponentially larger.

There are several techniques to reduce data-processing complexity of planning interactions with other agents including [22]: (i) dividing states in the classes of equivalence, (ii) reducing search space into states which are really required. (iii) planning on line, i.e., eliminating the possibilities which do not emerge during the execution of plan.

Our contribution in this research work is the use of another solution what is Fuzzy Logic Control. The Fuzzy Logic Control is a methodology considered as a bridge on the artificial intelligence and the traditional control theory [17]. This methodology is usually applied in the only cases when exactitude is not of the need or high importance [16]. Fuzzy Logic is a methodology for expressing operational laws of a system in linguistic terms instead of mathematical equations. Wide spread of the fuzzy control and high effectiveness of its applications in a great extend is determined by formalization opportunities of necessary behavior of a controller as a "fuzzy" (flexible) representation [14]. This representation usually is formulated in the form of logical (fuzzy) rules under linguistic variables of a type "If A then B" [12]. The Fuzzy Logic methodology comprises three phases: Fuzzyfication, Rule engine, Defuzzyfication [13].

This article is concerned with two important matters: how to define the MAS in a manner such that it has more utility to deploy it, and how to use such a MAS for the advanced software. The MAS must discover the action to be taken by supervising the application and its environment and analyzing the data obtained.

With MAS, we face two important matters: (i) the detection of a need for action. The need for action must be discovered by supervising the application and its environment and analyzing data obtained. (ii) the planning of the action. It consists to envisage the action (by proposing which modifications need to be made) and by programming it. In practice, the opposite dependency also requires consideration: Only those situations which can be repaired by an action taken which can really be planned should be considered during the analysis.

This paper introduces a simple Benchmark Production System that will be used throughout this article to illustrate our contribution which is developped as agent-based application. We implement the Benchmark Production System in a free platform which is JADE (JavaTM Agent DEvelopment) Framework. JADE is a platform to develop multi-agent systems in compliance with the FIPA specifications [5, 6, 2].

In the next section, we present the Benchmark Production System. The third section introduces the Fuzzy Multi-Agent System. We present in section 4 the creation of JADE agents.

2. BENCHMARK PRODUCTION SYSTEM

As much as possible, we will illustrate our contribution with a simple current example called RARM [11]. We begin with the description of it informally, but it will serve as an example for various formalism presented in this article. The benchmark production system RARM represented in the figure 1 is composed of two input and one output conveyors, a servicing robot 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 robot R transfers workpieces
one after another to the processing center. The next workpiece can be put on the input conveyor when it has been emptied by the robot. The technology of production requires that first one A-workpiece is inserted into the center M and treated, then a B-workpiece is added in the center, and last 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 one produced.

![Diagram of the benchmark production system RARM](image)

Figure 1. The benchmark production system RARM

Traditionally, the RARM systems are directly controlled by a central server. The server proposes the schedule for the system as a whole and dispatches commands to the robots. This results in reliable and predictable solutions. The central point of control also allows an easier diagnosis of the errors. However, a variation in user's needs leads to change the centralized architecture. Customers ask more and more for self-management systems, i.e., systems that can adapt their behavior with changing circumstances in an autonomous way. Self-management with regard to the dynamics of system needs two specific quality requirements: flexibility and openness.

Flexibility refers to the capacity of the system to treat dynamic operating conditions. The openness refers to the capacity of the system to treat robots leaving and entering system. To treat these new quality requirements, a radically new architecture was conceived based on multi-agent systems (Figure 2).

Applying a situated multi-agent system opens perspective to improve the flexibility and the openness from the system: the robots can adapt to the current situation in their vicinity, order assignment is dynamic, the system can therefore treat in an autonomous way the robots leaving and reentering the system, etc.

However, a decentralized architecture can lead to a certain number of implications, in particular distributed planning can have an impact on the total efficiency of the system. In fact, this critical topic must be considered during the design and development of multi-agent system.
3. FUZZY MULTI-AGENT SYSTEM

Multi-agent planning problems can sometimes be translated into non deterministic single-agent planning problems by modifying the plan-execution agent's actions to incorporate the effects of the other agents' possible responses to those actions. For example, suppose an agent RARM₁ is going to reduce the production.

The another agent RARM₂ may either decrease the production (in which case the agents can cooperate together) or increase the production (in which case neither agent can cooperate). As shown in Figure 3, this two possible actions can be modeled as nondeterministic outcomes.

The basic form of a fuzzy logic agent consists of: Input fuzzification, Fuzzy rule base, Inference engine and Output defuzzification (Figure 4).
3.1 Fuzzification

In the classical logic set, its characteristic function assigns a value of either 1 or 0 to each individual in the universal set, thereby discriminating between members and non-members of the crisp set under consideration. However, a fuzzy set is a set containing elements that have varied degrees of membership in the set. The fuzzification can be defined as a conversion of a precise quantity to a fuzzy quantity.

Running example

The number of defected pieces is measured through a sensor related to the system. The range of number of defected pieces varies between 0 to 40, where zero indicates the rate of defected pieces of A that is null (each piece is well) and 40 indicates the rate of defected pieces of A is very high. Now assume that the following domain meta-data values for these variable, \( VF = \) very few, \( F = \) few, \( Md = \) medium, \( Mc = \) much, \( VMc = \) very much. Assume that the linguistic terms describing the meta-data for the attributes of entities are: \( VF = [0,...,10], F = [5,...,15], Md = [10,...,20], Mc = [15,...,25] \) and \( VMc = [20,...,40] \).

Based on the metadata value for each attribute the membership of that attribute to each data classification can be calculated. In the Figure 5 and 6, triangular and trapezoidal fuzzy set was used to represent the state of defected pieces from A classifications (i.e. state of defected pieces from A classification levels: \( VF, F, Md, Mc, VMc \) whereas state of defected pieces from B classification levels: \( F, Md, Mc \)).

In the figure 7, state of production system classification levels: Null, Low, Medium and High.
The membership value based on its meta-data can be calculated for all these classification using the formulas:

Formulas for calculation triangular fuzzy memberships

\[
\begin{align*}
    m_A(x) &= 0 \text{ if } x < a_1, \\
    m_A(x) &= \frac{x-a_1}{a_2-a_1} \text{ if } a_1 \leq x < a_2, \\
    m_A(x) &= \frac{a_3-x}{a_3-a_2} \text{ if } a_2 \leq x \leq a_3, \\
    m_A(x) &= 0 \text{ if } x > a_3.
\end{align*}
\]
Formulas for calculation trapezoidal fuzzy memberships

\[
\begin{aligned}
    m_A(x) &= 0 \text{ if } x < a_1, \\
    m_A(x) &= \frac{x-a_1}{a_2-a_1} \text{ if } a_1 \leq x \leq a_2, \\
    m_A(x) &= 1 \text{ if } a_2 \leq x \leq a_3, \\
    m_A(x) &= \frac{a_4-x}{a_4-a_3} \text{ if } a_3 \leq x \leq a_4, \\
    m_A(x) &= 0 \text{ if } x > a_4
\end{aligned}
\]

(2)

**Running example**

As an example, we consider the membership functions for the fuzzy variable defected pieces from A. Figure 5 shows various shapes on the universe of defected pieces from A. Each curve is a membership function corresponding to various fuzzy variables, such as very few, few, medium, much and very much (Figure 8).

![Figure 8. Membership function representing imprecision in number of defected pieces from A](image_url)

**3.2 Rule Engine**

In the inference method we use knowledge to perform deductive reasoning. That is, we wish to deduce or infer a conclusion, given a body of facts and knowledge. Now that the data can be classified and categorized into fuzzy sets (with membership value), a process for determining precise actions to be applied must be developed. This task involves writing a rule set that provides an action for any data classification that could possibly exist. The formation of the rule set is comparable to that of an expert system. Thus, behaviors is synthesized as fuzzy rule base i.e. a collection of fuzzy if-then rules.

Each behavior is encoded with a distinct control policy governed by fuzzy inference. We write fuzzy rules as antecedent-consequent pairs of If-Then statements (Figure 9).
Running example

We take as example, the first column from the Table 1:

IF number of defected pieces from A is Very Few and number of defected pieces from B is Few Then Production is High.

IF number of defected pieces from A is Few and number of defected pieces from B is Few Then Production is High.

IF number of defected pieces from A is Medium and number of defected pieces from B is Few Then Production is High.

IF number of defected pieces from A is Much and number of defected pieces from B is Few Then Production is Medium.

IF number of defected pieces from A is Very Much and number of defected pieces from B is Few Then Production is Medium.

Table 1. Fuzzy Control rules for the Agent

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
<th>Md</th>
<th>Mc</th>
</tr>
</thead>
<tbody>
<tr>
<td>VF</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Md</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Mc</td>
<td>M</td>
<td>L</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>VMc</td>
<td>M</td>
<td>L</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>
Table 2. Selection-based rules for the Agent

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
<th>Md</th>
<th>Mc</th>
</tr>
</thead>
<tbody>
<tr>
<td>VF</td>
<td>H</td>
<td>H (0.2)</td>
<td>H (0)</td>
<td>M (0)</td>
</tr>
<tr>
<td>F</td>
<td>H</td>
<td>H (0.8)</td>
<td>H (0.4)</td>
<td>M (0.3)</td>
</tr>
<tr>
<td>Md</td>
<td>H</td>
<td>H (0.1)</td>
<td>M (0)</td>
<td>L (0.2)</td>
</tr>
<tr>
<td>Mc</td>
<td>M</td>
<td>M (0.6)</td>
<td>L (0.4)</td>
<td>N (0.2)</td>
</tr>
<tr>
<td>VMC</td>
<td>M</td>
<td>M (0.1)</td>
<td>L (0)</td>
<td>N (0)</td>
</tr>
</tbody>
</table>

Table 3. Final fuzzy values for the Agent

<table>
<thead>
<tr>
<th>Consequent</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0.8</td>
</tr>
<tr>
<td>M</td>
<td>0.6</td>
</tr>
<tr>
<td>L</td>
<td>0.4</td>
</tr>
<tr>
<td>N</td>
<td>0.2</td>
</tr>
</tbody>
</table>

FzSet AddLeftShoulderSet(std::string name,
                      double    minBound,
                      double    peak,
                      double    maxBound);

FzSet AddRightShoulderSet(std::string name,
                         double    minBound,
                         double    peak,
                         double    maxBound);

FzSet AddTriangularSet(std::string name,
                      double    minBound,
                      double    peak,
                      double    maxBound);

FzSet AddSingletonSet(std::string name,
                      double    minBound,
                      double    peak,
                      double    maxBound);

//fuzzify a value by calculating its DOM in each of this variable's subsets
void Fuzzify(double val);

//defuzzify the variable using the MaxAv method
double DeFuzzifyMaxAv()const;

//defuzzify the variable using the centroid method
double DeFuzzifyCentroid(int NumSamples)const;
Running example

/* Add the rule set */
fm.AddRule(FzAND(A_VF, B_F), High);
  fm.AddRule(FzAND(A_VF, B_Md), High);
  fm.AddRule(FzAND(A_VF, B_Mc), Medium);
  fm.AddRule(FzAND(A_F, B_F), High);
  fm.AddRule(FzAND(A_F, B_Md), Medium);
  fm.AddRule(FzAND(A_F, B_Mc), Medium);
  fm.AddRule(FzAND(A_Md, B_F), High);
  fm.AddRule(FzAND(A_Md, B_Md), Medium);
  fm.AddRule(FzAND(A_Md, B_Mc), Low);
  fm.AddRule(FzAND(A_Mc, B_F), Medium);
  fm.AddRule(FzAND(A_Mc, B_Md), Low);
  fm.AddRule(FzAND(A_Mc, B_Mc), Null);
  fm.AddRule(FzAND(A_VMc, B_VF), Medium);
  fm.AddRule(FzAND(A_VMc, B_VF), Low);
  fm.AddRule(FzAND(A_VMc, B_VF), Null);

3.3 Defuzzification

Fuzzy set is mapped to a real membered value in the interval 0 to 1.

If an element of universe, say x, is a member of fuzzy set A, then the mapping is given by $\mu A \in [0,1]$

The output of a fuzzy process needs to be a single scalar quantity as opposed to a fuzzy set. Defuzzification is the conversion of a fuzzy quantity to a precise quantity. There are many methods to calculate it such as Max membership, Centroid method, Weighted average method, Mean max membership, Center of sums, Center of largest area and First (or last) of maxima. Obviously, the best defuzzification method is context-dependant [13].

4. CREATING JADE AGENTS

JADE is a Java tool and therefore creating a JADE-based multi-agent system requires creating Java classes. For more details, we refer to [7, 8, 9, 10].

Creating a JADE agent is very easy through defining a class that extends the jade.core.Agent class and implementing the setup() method. Each class introduced in the Figure 10 will be presented in the following paragraphs.
import jade.core.Agent;

public class Robot extends Agent {
    protected void setup() {
        System.out.println("Hello everybody! I am an agent");
    }
}

### 4.1 Agent Identifier

Each agent is identified by an “agent identifier” represented as an instance of the jade.core.AID class. The getAID() method of the Agent class allows retrieving the agent identifier. An AID object includes a globally unique name plus a number of addresses. The name in JADE has the
form \texttt{<nickname>@<platform-name>} so that an agent called Robot1 living on a platform called RARM will have \texttt{Robot1@RARM} as globally unique name. The addresses included in the AID are the addresses of the platform the agent lives in. These addresses are only used when an agent needs to communicate with another agent living on a different platform.

### 4.2 Agent discovery

The JADE platform allows the possibility to discover dynamically the available agents. To do so, a yellow pages service permits agents to describe one or more services they provide. An agent can register (publish) services and search to discover services.

**Running example**

In order to publish a service, an agent must create a proper description which is an instance of DFAgentDescription class and call the register() method of DFService class.

```java
/// Register the Robot in DFService
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
ServiceDescription sd = new ServiceDescription();
sd.setType("Robot");
sd.setName("Robot-executing");
dfd.addServices(sd);
try {
    DFService.register(this, dfd);
} catch (FIPAException fe) {
    fe.printStackTrace();
}
```

It is possible to search some agents, if the agent provides the DF with a template description. The result of the research is a list of all the descriptions matching the template.

**Running example**

The search() method of the DFService class ensures the result.

```java
DFAgentDescription template = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
sd.setType("Robot");
template.addServices(sd);
DFAgentDescription[] result ;
try {
    do {
        result = DFService.search(myAgent, template);
        robotAgents = new AID[result.length];
        for (int i = 0; i < result.length; i++) {
            robotAgents[i] = result[i].getName();
        }
    } while (result.length <= 0);
```
4.3 Message exchanged between JADE Agents

Agents never interact through method calls but by exchanging asynchronous messages. Obviously, inter-agent interaction will be very difficult until all agents adopt the same communication language, and fortunately ACL standards ensure this requirement. All JADE agents communicate using messages that obey the FIPA ACL specification, which is described in: http://www.fipa.org.

This format comprises a number of fields and in particular: (1) the sender of the message, (2) the list of receivers, (3) the communicative intention (also called “performative”) indicating what the sender intends to achieve by sending the message (for example the performative can be REQUEST, INFORM, QUERY_IF, CFP (call for proposal), PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL, and so on). (4) The content i.e. the actual information included in the message which may be string in simple cases; otherwise we need a content language, a corresponding ontology, and a protocol. (5) The ontology i.e. the vocabulary of the symbols used in the content and their meaning (both the sender and the receiver must be able to encode expressions using the same symbols to be sure that the communication is effective).

4.3.1. Sending a message

Sending a message to another agent is as simple as filling the fields of an ACLMessage object and then call the send() method of the Agent class. The code below informs an agent whose nickname is Robot1 that the production must be decreased.

Running example

```java
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Robot1", AID.ISLOCALNAME));
msg.setOntology("Production");
msg.setContent("We must decrease in the production");
send(msg);
```

4.3.2. Receiving a message

As mentioned above the JADE runtime automatically posts messages in the receiver’s private message queue as soon as they arrive. An agent can pick up messages from its message queue by means of the receive() method.

This method returns the first message in the message queue (removing it) or null if the message queue is empty and immediately returns.

Running example

```java
ACLMessage msg = receive();
if (msg != null) {
    // Process the message
}
```
4.3.3. Blocking behavior waiting a message

Some behaviors must be continuously running and at each execution of their action() method, must check if a message is received and perform some action.

Running example

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

4.3.4. Selecting a message

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

Running example

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

4.4 Agent Behavior in JADE

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 life. A behavior represents a task that an agent can carry out and is implemented as an object of a class that extends jade.core.behavioursBehaviour. To make an agent execute the 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 other behavior (Figure 11).

![Diagram of Behaviour class hierarchy in JADE](image)

Figure 11. Behaviour class hierarchy in JADE

- class Behaviour: Each class extending the abstract class Behavior must implement two 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 behaviour execution. onEnd() returns an integer that represents a termination value for the behaviour. It should be noted that onEnd() is called after the behaviour has completed and has been removed from the pool of agent behaviours.

- class SimpleBehaviour: The SimpleBehaviour class is an abstract class modeling simple atomic behaviours. Its reset() method does nothing by default, but it can be overridden by user-defined subclasses.

- class OneShotBehaviour: The OneShotBehaviour class models atomic behaviours that must be executed only once and cannot be blocked. So, its done() method always returns true. The class WakerBehaviour implements a one-shot task that must be executed only
once just after a given timeout is elapsed. The class TickerBehaviour implements a
cyclic task that must be executed periodically.

- Class CyclicBehaviour: The CyclicBehaviour class models atomic behaviours that must
  be executed forever. So its done() method always returns false. “Cyclic” behaviours that
  never complete and whose action() method executes the same operations each time it is
called.

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

**Running example**

```java
int nbPositive = 0;

protected void setup()
{
  ACLMessage msg = newMsg(ACLMessage.QUERY_REF);

  MessageTemplate template = MessageTemplate.and(
    MessageTemplate.MatchPerformative(ACLMessage.INFORM),
    MessageTemplate.MatchConversationId(msg.getConversationId()));

  SequentialBehaviour seq = new SequentialBehaviour();
  addBehaviour(seq);

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

  for (int i = 1; i <= nbRobots; i++)
  {
    msg.addReceiver(new AID( "Robot" + i, AID.ISLOCALNAME));

    par.addSubBehaviour(new myReceiver(this, 1000, template)
    {
      public void handle(ACLMessage msg)
      {
        if (msg != null) {
          if (msg.getPerformative() == ACLMessage.ACCEPT) {
            nbPositive = nbPositive + 1;
          }
        }
      }
    });
  }

  seq.addSubBehaviour(new OneShotBehaviour()
  {
    public void action()
    {
      if (nbPositive == nbRobots)
        System.out.println("All agents accept to change the production");
      else
```
System.out.println("Some agents refuse to change the production");
}
});

5. CONCLUSION

Distributed planning is narrowly interlaced with the distributed resolution of the problems, being a problem in itself and means to solve a problem. The main aim of this paper is how to ensure a distributed planning in Multi-Agent System (MAS) composed of several intelligent autonomous agents able to take the initiative instead of simply reacting in response to its environment. Our solution to this problem is the use of fuzzy logic which is based on three steps: fuzzification, rule engine and defuzzification. We create the MAS through JADE platform and show the interaction between the different agents through exchanging messages. All our contributions are applied on the benchmark production system (RARM system).

REFERENCES


[19] Sergio Pajares Ferrando, Eva Onaindia, Context-Aware Multi-Agent Planning in intelligent environments, Information Sciences, Volume 227, 1 April 2013, Pages 22-42


[22] Tsz-Chiu Au, Ugur Kuter, and Dana Nau, Planning for Interactions among Autonomous Agents


AUTHORS

Atef Gharbi received his computer engineering Diploma from the National School in Computer Science (ENSI) of Tunisia, in 2005. After that, he received the Master degree from the National Institute of Applied Sciences and Technology (INSAT) of Tunisia in 2007. He obtained his Phd in 2013. He is currently related to LISI Research Laboratory in Tunisia. His research interests include specification of model, verification of properties related to functional safety, implementation of software solutions to ensure functional safety.

Samir Ben Ahmed is a Full Professor in Computer Science at Tunis-El Manar University, President of National Institute of Applied Sciences and Technology (INSAT), and Head of MOSIC Research Unit in Tunisia. He was Founder of ISI Institute of Computer Science, and Head of IT Department of Faculty of Science at Tunis-El Manar University in Tunisia. Prof. Ben Ahmed obtained his PhD Thesis in Automation and Computer Science at Paul Sabatier University in France. The Engineering Diploma was obtained before from National School of Electrical Engineering, Electronic, Computer Science and Hydraulic in Toulouse (ENSEEIHT). Prof. Ben Ahmed is strongly active in several National and International Projects and Collaborations.