A methodology to solve optimisation problems with MAS application to the graph colouring problem

Gaële Simon, Marianne Flouret, Bruno Mermet

LIH, Université du Havre, 76058 Le Havre Cedex, France
{Gaele.Simon, Marianne.Flouret, Bruno.Mermet}@univ-lehavre.fr

Abstract. Developing multi-agent systems may be a rather difficult task. Having confidence in the result is still more difficult. In this article, we describe a methodology that helps in this task. This methodology is dedicated to global optimization problems that can be solved combining local constraints. We developed CASE tools to support this methodology which are also presented. Finally, we show how this methodology has been successfully used to develop a multi-agent system for the graph colouring problem.

Keywords: multi-agent system, methodology, graph colouring.

1 Introduction

Many methods already exist to built multi-agent systems (MAS) [12, 15]. Theses methods are composed of a set of models, but there is often a lack of methodology to help to transform real system specifications into a modelization provided by the method. [18, 1]. In this paper, we introduce the bases of a methodology associated to a method to develop multiagent systems that are dedicated to optimization problems. The first part of the article presents essentially the *methodology*. In the second part, we introduce a *system* made of java classes we developed to help us to write agents derived from our methodology. Finally, in the third part, we present an *application* of our methodology to a particular problem : the graph colouring problem. Using the methodology and the system presented before, we developed a MAS for the graph colouring problem.

2 Methodology

The goal of our research is to have a method, a methodology and also tools to help the analysis and design of distributed problem solving by MAS.

An important aspect of our methodology is that :

- at *any time*, the system can be stopped ;
- from time to time, the solution proposed by our system gets better and better.

As presented in the sequel, the methodology is based on a top-down approach which guaranty the progress of our system towards a *good* solution.

2.1 Usage conditions

The methodolgy defined here must be used to solve global problems which can be specified by a set of more local constraints (LC). A more restrictive usage condition relies in the fact that this methodology is dedicated to optimisation problems for which a trivial (but bad) solution exists.

Of course, for non NP-hard and non distributed problems for which a sequential algorithm is known, using agents (and so our methodology) is rarely a good solution because communications and synchronisations introduced by MAS make the program less efficient [17].

An example of a target problem for our methodology is the graph colouring problem which consists in colouring a graph with a minimum number of colors in such a way that two connected nodes do not have the same color. This application is presented in section 5.

2.2 The methodology

Global variant The first thing to do is to define a variant : a notion often used to prove termination of algorithms. A variant is a variable defined on a totally ordered structure that must decrease at each iteration and that has a lower bound. These two properties imply the termination of the iterations.

Local decomposition The second step is perhaps the hardest one : the global problem has to be expressed in terms of local sub-problems. This consists in dividing the solution of the problem into several parts. These parts are not necessarily disjunctive. Each part is associated to a local sub-problem. The resolution of each of these sub-problems must help to solve the global probem. The ideal case is a sub-problem whose resolution is a necessary condition for solving the global problem. However, this is not always the case. An other possibility is a sub-problem whose resolution makes the global variant decrease.

Agentification Once the global problem has been decomposed, we still do not have agents. Of course, a first idea could be to assign each local problem to an agent, but this is not always possible for the following reason.

To agentify a problem, two types of constraints must be considered :

- each sub-problem must be assigned to an agent ;
- each property (piece of data) must be assigned to an agent.

Each agent perceives only a local part of the environment. Moreover, an agent being autonomous, no other agent can modify directly its properties. These two constraints are called *the locality principle*. So, if the resolution of two sub-problems rely in modifying the same property, assigning two problems to two different agents is impossible. A first solution could be to assign properties to the environment. This is an easy solution, but this makes the environment a central resource for our MAS, limiting the benefit of the distribution.

A better solution is to change the structure of the local-problems so that modifying a property can occur only in the resolution of one sub-problem. So, each property modification is controlled by one and only one agent. Other agents that need to get the value of this property must have the agent owning the property in their accointance set and can know its value by message passing. Sub-problems resulting from this restructuring are called Property Oriented Sub-Problems (POSP) in the sequel.

This step is necessary (it provides the agents and the accointance relations of the MAS) and not so difficult to realize as it is shown in this article for the graph-colouring problem.

Agents behaviour We consider the agents as reactive and social ones, that is they can react to changes of their environment and communicate with other agents.

General behaviour Each POSP is assigned to an agent. So, the general behaviour of each agent is very simple :

- if its problem is solved, it does nothing (it could also help other agents). The agent is *satisfied* ;
- otherwise, it tries to find a solution to its problem.

Solving a problem For the global problem, we introduced a variant. We have to do the same for each POSP in such a way that each time a local variant decrease, the global variant does not increase.

Each POSP can be divided into sub-goals whose resolution makes the local variant decrease. Then, a not satisfied agent chooses a sub-goal and must solve it.

When a sub-goal has to be solved, there are two cases :

- either it can be solved by the agent : the agent can then choose a new goal ;
- or the agent cannot solve it.

There are two reasons making a subgoal unable to be reached :

- either there is a blocking situation : an other agent prevent the acting agent to apply one of its stategies ;
- or the agent doesn't know what to do to solve the subgoal in the given situation.

In the second case, the agent chooses an other goal or waits for a modification of the situation. In the first case, the agent attacks the obstructing agent. This behaviour follows the eco-agent's one [7]. The attack mecanism is simulated by sending an agression message. An agent under attack has to flee so that the blocking situation disappears, but preserving the local constraints LC. Note that the fleeing behaviour can increase the local variant. If the agent cannot flee, it ignores the attack.

In order to help us specifying problems agents behaviours, we used the formalism of automata with multiplicities [2]. This formalism can also be used to specify behaviours of other kinds of agents [13]. Thus, we defined the general behaviour of an eco-agent by the automaton shown figure 1.

$\Sigma = \{Init, S, NS, A, IC, E\}$ is the set of the following perceptions :

$Init$: Synchronization signal,
S : goal satisfied,
NS : goal unsatisfied,
A : attacked by an other agent,
IC : main action impossible,

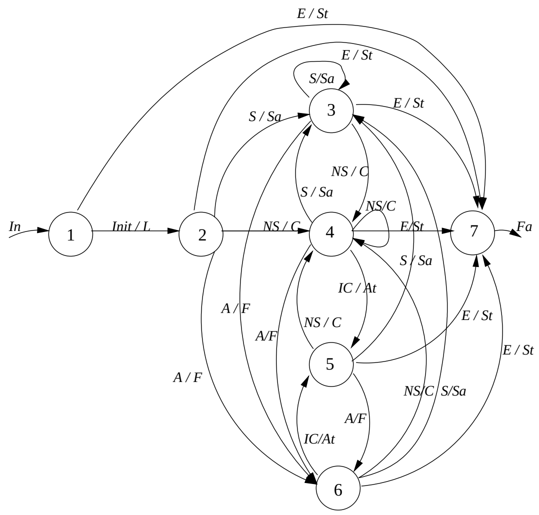

Fig. 1. eco-agent : general behaviour

E : stop signal.

The set of elementary actions (behaviour primitives), performed when a state is reached, is defined by {In,L,C,Sa,At,St,F,Fa} with

In : initialization of agent parameters,
L : lauching agent functionalities,
C : main action itself,
Sa : satisfied message,
At : agression,
St : stop process,
F : flee,
Fa : final message.

To help us developping MAS whose agents are specified by automata with multiplicities, we developped Java classes that are described in the next section.

3 System : CASE Tools dedicated to our agents

3.1 Presentation of the system

In order to help developers in coding agents specified by automata with multiplicities, Java classes have been developed by our team. These classes give to the programmer an upper level to the MAS platform Madkit on which they rely [14]. These classes can however be easily adapted to other contexts because their concepts are general.

The Madkit plateform is based on a multiagent model (agent, group, role) proposed in the Aalaadin method [8].

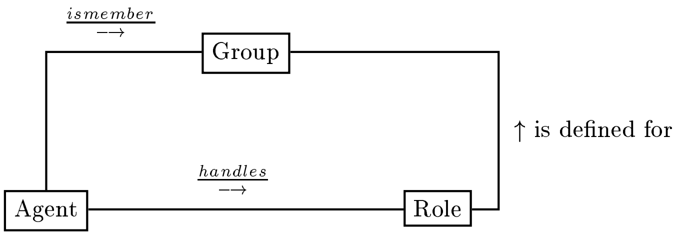

Fig. 2. AALAADIN concepts

In this model, an agent is defined as an autonomous and communicating entity that can play a given number of roles in one or more groups. A group is a set of agents. A semantic is given to groups at the design step, depending of the application. Roles are functionnalities (or services) performed by the agents in a given group. There is no other constraint in this model. Moreover, the inner structure of each agent is unspecified.

Madkit provides an `Agent` class based on three main methods :

- **activate** : describes actions to perfom when the agent is initialized ;
- **live** : specifies the general behaviour of the agent ;
- **end** : lists actions to be executed when the agent dies.

To develop its application, the developer must create its own classes extending the `Agent` class. The three methods described above must be written using primitives provided by Madkit, helping to managing groups, roles, communications, etc.

The classes, we developed and we present in this section, specialize the `Agent` class of Madkit to help to describe the agent's behaviour. According to the first part of this article, here are the concepts our system helps to manage :

- description of the general behaviour of the agent by an automaton with multiplicities ;
- description of the set of perceptions of the agent
- description of actions associated to perceptions ;
- description of the behaviour of the agent according to its inner state.

These four concepts are available thanks to two main classes : the `Automate` and `AgentAuto` classes.

The *Automate* (automaton) class The goal of this class is to provide functionnalities to built and use automata with multiplicities. In this class, each state is caracterized by :

- a number n ;
- a method $etat_n$ describing the behaviour of this agent in state n. This method must be written in the class representing the agent (presented below).

Following the definition given in [2], each transition of the automaton is specified by a 4-uple (initial state, final state, perception, action). Initial and final states are represented by their number. The *perception* is identified by a method that is executed to determine wether the perception is valid or not. If it is valid, the transition may be fired. The *action* is identified by a method that is executed if the corresponding transition is activated. Both methods representing the perception and the action must be defined in the class describing the agent.

The Automaton class provides to the user the two following main methods :

- *setTransition* : a method to add a transition to the automaton. The four parameters correspond to the four characteristics of a transition ;
- *transiter* : this method determines valid transitions in the current state by dynamic invocation. If many perceptions are valid, a method *choix_perception* (choose perception), defined in the class associated to the agent, is executed to determine the transition to fire. The associated action is executed (by dynamic invocation). Finally, the method associated to the final state is executed.

AgentAuto **class** This class inheritates from the *Agent* class of Madkit. It allows to implement an agent whose behaviour is described by an automaton with multiplicities. The caracteristics of this class are the following :

- definition of the *live* method : the standard algorithm looks like this :

```
current_state = initial_state;
execute the initial_state method
while (current_state != final_state)
     current_state = transiter(current_state);
```

- definition of the *choix_perception* method : this method tells to the *Automate* class which perception must be chosen when several are valid at the same time. A default version, written in the *AgentAuto* class, chooses a perception randomly.

Using classes described above To define an agent, a developer has to write a class *MyAgent* inheritating from the *AgentAuto* class defining at least the *activate, etat*$_i$*, perception and action* methods as shown in the following example.

3.2 A toy example

We want to write an agent with the behaviour described figure 3.

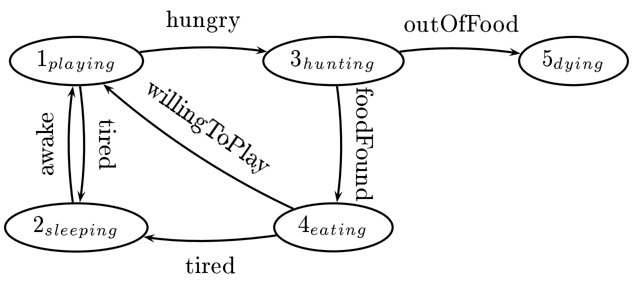

Fig. 3. behaviour of a simple agent

This behaviour can be built on 4 properties :

```
private int food_storage;
private int tiredness;
private boolean hasFood;
private boolean isNoMoreTired;
```

The automaton is defined using the *setTransition* method in the constructor. Here are three examples of methods :

– a method associated to a *state*, the *playing* state :
```
public void etat1() {// I play
    foodStorage-=2;
    tiredness++;
}
```
– a *perception* method, the *tired* perception :
```
public boolean isTired() {return (tireness > 5);}
```
– an action to perform when a state change is performed, the *tiredAction* method :
```
public void tiredAction()
   {System.out.println("I am tired");isNoMoreTired=false;}
```

4 Application to graph colouring

4.1 The graph colouring problem

We give in this part the application of the methodology presented before to a graph colouring problem. The *general problem* is to color the nodes of a graph with a minimal number of colors (*optimisation*) without two neighbour nodes having the same color (*local constraint*).

The problem of graph colouring being NP-hard, algorithms looking for optimal solutions are numerous [5] but rarely usefull for real-size problems. We can refer to [6, 9–11] for various methods trying to solve this problem in which two connected nodes never must have the same colour, and more precisely to [3, 16] for ants algorithms.

The essential characteristic with our solution is that it starts with a correctly coloured graph but not optimal as far as the number of colours is concerned. For instance, a trivial initial solution is to assign a different color for each node. As the time goes, our algorithm tries to suppress colors, keeping a correct coloration of the graph, and, at any time, we can stop our algorithm and obtain a correctly coloured graph. Obviously, the more our algorithm will work, the more pertinent the proposed solution will be.

4.2 Some coloured graph properties

For details about graph definitions and properties, it can be refered to [4] for example. Here are given the main ones used in the sequel. We will denote $G = (N, E)$ an oriented (resp. non oriented) *graph* with N and E two sets such that elements of E are ordered (resp. unordered) couples $(u, v) \in N^2$, and

$N \cap E = \emptyset$. The elements of N are *nodes*, those of E are the *edges*. Two nodes u, v of G are *neighbours* if $(u, v) \in E$. $V(u)$ will denote the set of all neighbours of u.

Let $C(u)$ the color associated to a node u, and $C(V(u))$ the set of colours of u neighbours. The k-colouring of a graph $G = (N, E)$ is the attribution, to each node, of a colour among k such that, for each edge (u, v) of E, $C(u) \neq C(v)$.
A graph is *k-colourable* if a k-colouring can be applied[1]. The smallest k such that G is k-colourable is the *chromatic number* of G and denoted $\chi(G)$.
In the sequel, we will consider a k-coloured graph.

For this application, we have to define two new specific notions concerning nodes. The *local chromatic number* of a node u is $lcn(u) = max\{|C|, \forall C$ clique of G $/u \in C\}$. The *current chromatic number* of u is $ccn(u) = |\{c(u)\} \cup \{c(v)/v \in V(u)\}|$. Then, a node u *satisfies its lcn* if and only if $lcn(u) = ccn(u)$. The following properties are used to implement our solution.

Theorem 1. *Let $G = (N, E)$ be a graph. For all node $u \in N$, if G is correctly coloured, then $ccn(u) \leq lcn(u)$.*

Theorem 2. *Let $G = (N, E)$ a graph, and let $\chi(G) = n$. For each node $u \in N$, we have $lcn(u) \leq n$.*

Remark 1. Let us notice that, despite these two theorems, even if all the nodes of a graph satisfy their *lcn*, the chromatic number of the graph can not always be reached, or some graphs cannot be coloured such that each node satisfies its *lcn*.

4.3 Application

Global variant Our goal is to make decrease the number of colours of the nodes of a graph (the chosen global variant), trying to reach the graph chromatic number.

Local decomposition The previous property is decomposed into subproblems for each node : the algorithm tries to turn its *ccn* down while it does not satisfy its *lcn*.

Agentification The previous decomposition does not follow the locality principle (to make its *ccn* decrease, an agent should modify the color of another agent). So, the *POSP* of our general problem are to make the *ccn* of the neighbours of a node u decrease, changing the color of u. Then, each *POSP* is assigned to an agent called a *node agent*. It can be reached by solving a set of subgoals (decreasing the *ccn* of a giben neighbour). Notice that as each node agent has at least one neighbour, its neighbours will make its *ccn* decrease.

[1] For $k \geq 3$, decide wether a graph is k-colourable or not is NP-hard.

Agents Behaviour When the *POSP* assigned toan agent is not satisfied (that is one of its neighbours has a *ccn* greater than its *lcn*), it has to choose a color :

- existing in the graph ;
- making the *ccn* of the neighbour n decrease ;
- being different from the colors of its neighbours.

As a node agent can only see (and communicate with) its neighbours, to find a color verifying the two first items enumerated above, it asks to its neighbour u the colors of its neighbours, which gives a first set $C(V(u))$, the colors set of the neighbours of u.

To verify the third point, the acting agent a first asks to its neighbours their colors and constructs the set $C(V(a))$ of these colors. Then it chooses a color among the new set $S = C(V(u)) \backslash \{C(V(a)) \cup C(a)\}$[2].

If a node agent u cannot change its colour, necessarily, the set $C(V(u))$ is a member of the set $C(V(a))$. In such a case, the node agent u attacks one of its neighbours whose color is in the set $C(V(u))$. If it is attacked, it flees, trying to take another color. It chooses a color among all ones of the neighbours of its neighbours, but not a color of its neighbours.

Two other agents have to be created for coordination and implementation reasons. The topological agent creates the initial graph, node agents with their characteristics (e.g. list of neighbours, initial colors), and a drawer agent giving a graphic view of the graph updated when colors change.

Now we can precise the structure of the automaton with multiplicities which defines the node agents behaviour. It has been given in figure 1 of paragraph 2.2 under its general form. With the notations of figure 1, we associate the IC perception to the impossibility to change its own colour, and the C action to the fact of recolouring itself.

5 Conclusion and future work

The last part presented an application of the methodology presented before to a graph colouring problem. It allowed to illustrate that the methodology presented in this paper can be applied to real problems.

Our research now leads in adding to the methodology a fully specified method with formal or semi-formal models (like, for instance, automata with multiplicities presented here) to help designers of MAS.

CASE-Tools will also have to be developed to support both the methodology and the method.

References

1. F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, and J. Treur. DESIRE: Modelling multi-agent systems in a compositional formal framework. *Int Journal of Cooperative Information Systems*, 6(1):67–94, 1997.

[2] The new color must be different from the previous one, that is why $C(a)$ is removed from possible colors.

2. V. Jay D. Olivier C. Bertelle, M. Flouret and J.-L. Ponty. Automata with multiplicities as behaviour model in multi-agent simulations.
3. F. Comellas. An ant algorithm for the graph colouring problem. http://citeseer.nj.nec.com/112038.html.
4. R. Diestel. *Graph Theory*. Springer-Verlag, New-York, 2000.
5. dimacs92. Clique and coloring problems, a brief introduction, with project ideas, 1992. ftp://dimacs.rutgers.edu/pub/challenge.
6. Raphaël Dorne and Jim-Kao Hao. A new genetic local search algorithm for graph coloring. In Agoston E. Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature – PPSN V*, pages 745–754, Berlin, 1998. Springer. http://citeseer.nj.nec.com/dorne98new.html.
7. A. Drogoul. *De la simulation multi-agents à la résolution collective de problèmes : une étude de l'émergence de structure d'organisation dans les systèmes multi-agents*. PhD thesis, Univ. Paris VI, 1993.
8. J. Ferber and O. Gutknecht. Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems, 1998.
9. G. Ribeiro Filho. Improvements on constructive genetic approaches to graph coloring. http://citeseer.nj.nec.com/242708.html.
10. G. Ribeiro Filho and G. Lorena. A constructive genetic algorithm for graph coloring, 1997. http://citeseer.nj.nec.com/filho97constructive.html.
11. G. Ribeiro Filho and G. Lorena. Constructive genetic algorithm and column generation: an application to graph coloring, 2000. http://citeseer.nj.nec.com/filho00constructive.html.
12. Carlos Iglesias, Mercedes Garrijo, and José Gonzalez. A survey of agent-oriented methodologies. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 317–330. Springer-Verlag: Heidelberg, Germany, 1999.
13. Bruno Mermet. Formal model of a multiagent system. In Robert Trappl, editor, *Cybernetics and Systems*, pages 653–658. Austrian Society for Cybernetics Studies, 2002.
14. J. Ferber O. Gutknecht and F. Michel. Madkit : une expérience d'architecture de plateforme multi-agent générique. 2000.
15. Arsène Sabas, Sylvain Delisle, and Mourad Badri. A comparative analysis of multi-agent system development methodologies : Towards a unified approach. In Robert Trappl, editor, *Cybernetics and Systems*, pages 599–604. Austrian Society for Cybernetics Studies, 2002.
16. A. Vesel and J. Zerovnik. How good can ants color graphs? *Journal of computing and Information Technology - CIT*, 8:131–136, 2000. http://citeseer.nj.nec.com/443529.html.
17. Michael Wooldridge and Nicholas R. Jennings. Pitfalls of agent-oriented development. In Katia P. Sycara and Michael Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98)*, pages 385–391, New York, 9–13, 1998. ACM Press.
18. Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.