Formal model of a Multiagent System

Bruno Mermet
Laboratoire d'Informatique du Havre
email: Bruno.Mermet@univ-lehavre.fr

Abstract

In this article, we make a proposal for a general definition of a Multiagent System, whatever may be the agents. We show how the B method can be used to give this definition a formal aspect, and we introduce a few proofs that can be then performed. Finally, we introduce the notion of refinement and its avantages in the context of the formal specification of multi-agent systems.

key-words : MultiAgent system, Formal specification, B method

1 Introduction

Multiagent systems are systems where calculus that must be made are distributes among many different entities. The way these entities interact is a crucial problem, not only for efficiency reasons, but also because it determines the correctness of the result.

Formal specifications are technologies underlying over mathematicals concepts and that allow to prove the "correctness" of a program. A "correct" program is a program that establish the properties it must guarantee. These methods are used by firms to prove "critical" systems, that is systems whose failure could endanger human lifes (transport control systems for instance). However, writing formal specification is so hard that this job concerns only the "critical kernel" of these systems.

In this article, we present a way to write formal specifications of multiagent systems in order to be able to make proofs on its behaviour.

The first part describe our point of view of what a multiagent system is. Then, we briefly describe the formal method we use, namely the B method. The main part of the article is devided into two sections : the first one deals with the formal specification of a multiagent system, and the second one deals with the proof of this specification. At last, we conclude and present the future work planned.

2 The multiagent system notion

There are many different definitions of multiagent systems ([Bouron, 1992], [J. and J., 1991], [Mike, 1999]). Somme people consider every objet playing a role in the system as an agent, whereas others think that an entity must be pro-active to be an agent ([Luck and d'Inverno, 1995a]). Moreover, these definition often confuse properties of the agents with properties of the system. Our goal in this article relying on the characterisation of properties concerning just the system, and as a consequence the evolution of its population and the interactions between agents. So, we use an abtact definition for the agents : an agent is simply described by an automaton. We do not take about the form of its actions : (decided at the beginning or not, determined by a belief about the other agents or not, etc.).

There are two reasons :

- we think that a great part of the properties of a multiagent system is a consequence of the interactions between agents, and the formal study of them is not well studied ;
- the formal method we use implement the refinement process. This is a way of specifying and prooving properties in many steps : the properties that are proved at step n are preseved at step $n+1$ and so, can be used and do not have to be proved again. So, the behaviour of the agents can ben introduced during the refinement process.

The interaction model we choose is a model where at every time, an agent can send messages to a set of other agents. Every agent owns a buffer where it queues messages recieved before it processes them. The notion of "known agents" can be introduce during the refinement process if we want to add the fact that an agent can send messages only to the agents it knows.

3 The B method

The B-method, described in [Abrial, 1996], has been developed by Jean-Raymond Abrial. It is a formal model-oriented specification method, that uses the first-order logics, the arithmetics and the sets theory to describe datas and properties on them. State changes are described by operations, written as *generalized substitutions.*

An other important characteristic of the B method is the refinement process ([Back and Sere, 1993], [Chandy and Misra, 1988]) : from an abstact specification of the problem, we derive more and more concrete and precise specifications until the specification obtained can be automatically translated into a programming language. A each step, the compatibility with the previous step must be proved. An other reason that made us choose the B method is the CASE-Tool «Atelier B» [Stéria Méditerrannée, 1997], performing automatically a great number of proofs.

3.1 Example

A B specification is structured into *machines*. A machine is a module encapsulating variables. Figure 1 presents the beginning of a B machine allowing to manage a car collection, where cars are described by their color and registration number.

```
MACHINE
    collection
SETS
    CARS, NUMBERS, COLORS
VARIABLES
    cars, registration, color
INVARIANT
    cars ⊆ CARS ∧
    registration ∈ cars ↣ NUMBERS ∧
    color ∈ cars → COLORS
```

Figure 1: collection.mch, static part

The *MACHINE* statement introduces the name of the module. The *SETS* part defines data types needed. They can be definite or not.

Types are assigned to variables in the invariant as properties. For instance, the *car* variable represents the set of cars of our collection. As there are `CARS`, this is a subset of the `CARS` type. Each car we owns is identified bye its registration number. Every car has a registration number, and a registration number corresponds to exactly one car. So, the registration is an injection[1] from the set of our cars to the set of legal registration numbers. This is indicated by the second property of the invariant : $\rightarrowtail$ is the injection symbol. At last, every car has one and only one color, but many cars can have the same color. So, their is a total function ($\rightarrow$) from the set of our cars to the `COLORS` set of allowed colors.

The dynamic part is implemented by two statements : initialisation and operations.

- the initialisation must establish the invariant, that it it the values it assigns to variables must be such that the invariant is true after the initialisation ;
- operations must keep the invariant true : if it true before an operation is executed, it is still true after.

Figure 2 shows an example of a dynamic part for our *collection* machine.

[1] a function f is an injection if and only if f(x) = fy ⇒ x = y

```
INITIALISATION
    cars := ∅ ||
    registration := ∅ ||
    color := ∅
OPERATIONS
change_color(voit, coul) =
PRE
    voit ∈ cars ∧
    coul ∈ COLORS
THEN
    color(voit) := coul
END
;
add_car(immat, coul) =
PRE
    immat ∈ NUMBERS - ran(registration)
    ∧ coul ∈ COLORS
THEN
    ANY voit WHERE
        voit ∈ CARS-cars
    THEN
        cars := cars ∪ {voit} ||
        registration(voit):=immat ||
        color(voit) := coul
    END
END
```

Figure 2: collection.mch, dynamic part

3.2 Proofs

In the B method, Proofs may be separated in two groups :

- invariance proofs ;
- refinement proofs.

In this article, we are only interested in abstract specifications, so, we will not present refinement proofs. There are two kinds of invariant proofs :

- Establishement of the invariant by the initialisation ;
- preservation fo the invariant by every operation.

3.3 A few symbols

Here are a little explanation of a few symbols used in the sequel :

- $f \lhd\!\!\!- g = \{(x,y) \mid ((x,y) \in f \land x \notin dom(g)) \lor (x,y) \in g\}$
- a pair (x,y) can ben represented by $x \mapsto y$
- $f(x) := y$ is equivalent to $f := f \lhd\!\!\!- \{x \mapsto y\}$

4 Formal specification of a multiagent system in B

A multiagent System (MAS) is made of agents. These agents are described by automata. An automaton is described by a set of states and a set of transitions between states. Transitions can be fired either spontaneously (pro-activity of the agent) or when a given message is received (reactivity of the agent). In the model we choose, a transition can also send a message.

So, there are 3 kinds of data : AGENTS, STATES and STIMULI (a transition is a relation between data of these types, so there is no TRANSITION types). To have an homogenous model, we see spontaneous transitions as standard transitions responding to a special stimulus called NOTHING.

The skeleton of our specification is shown figure 3.

```
MACHINE
        SMA
SETS
        AGENTS; STATES; STIMULUS
CONSTANTS
        NOTHING
PROPERTIES
        NOTHING ∈ STIMULUS
```

Figure 3: skeleton of our MAS model

4.1 Data

Data correspond to everything that can change in the system. At first, it is possible to create et kill agents. So, we need a set `agents` to know agents present in the system. Moreover, each agent is characterized by a set of stages, a set of transitions between these states and its current states. Each agent is in one state and no more, so, the `state` fonction is a total fonction.

The behaviour of an agent is described by a unique transition system. So, The transition variable is a total function, and its domain is the set of agents in the system. A transition is characterized by : a start state, an initiator stimulus (it can be the `NOTHING` one), an end state and a stimulus sent during the transition (potentially `NOTHING`).

Figure 4 show a modelization of the first part of the data of the system.

```
VARIABLES
        agents, state, transition
INVARIANT
        agents ⊆ AGENTS ∧
        state ∈ agents → STATES∧
                transition ∈ agents
→ ((STATES*STIMULUS) ↔
(STATES*STIMULUS))
```

Figure 4: elementary data of a MAS

Moreover, we suppose that an agent sends a message to many agents. These messages remain available to the receiver until it takes them into account (or it dies). So, the set of pending messages is a set of couples whose first element is a message, and whose second element is the list of the agents the message is sent to. So, we add to our model the `pendingMessages` variable, as shown figure 5[2].

[2] $\mathcal{P}(X)$ means the set of parts of X

```
VARIABLES
        pendingMessages
INVARIANT
        pendingMessages ⊆ (STIMULUS *
𝒫(agents))
```

Figure 5: other necessary data of a MAS

At the beginning, we do not know the precise state of our system ; the only thing we know is that there is no pending message.

```
INITIALISATION
        ANY ag, et, tr WHERE
                ag ⊆ AGENTS ∧
                et ∈ ag → STATES ∧
                tr ∈ ag → ((STATES * STIMU-
LUS) ↔ (STATES * STIMULUS))
        THEN
                agents := ag ||
                state := et ||
                transition := tr ||
                pendingMessages := ∅
        END
```

Figure 6: initialisation of the system

4.2 Evolution of the system

Of course, a multiagent system evolves. For instance, it is possible to create an agent with its own transition system. This is the goal of the `create` operation shown figure 7. This function create an agent with a new identifier (this is the role of the `ANY` statement), assigns to it the transition system (`tr`) and the initial state (`et` given in parameters.

```
OPERATIONS
create(et, tr) =
        PRE
                et ∈ STATES ∧
                tr ∈ (STATES * STIMULUS) ↔
(STATES * STIMULUS)
        THEN
                ANY ag WHERE
                        ag ∈ AGENTS \agents
                THEN
                        agents := agents∪ {ag} ||
                        state(ag) := et ||
                        transition(ag) := tr
                END
        END
```

Figure 7: creation of an agent

Agents can alos be created by cloning of an existing agent. A clone has the same transition table as the cloned agent, and starts its life in the state the cloned agent was at the cloning time. However, pending messages sent to the cloned agent are not preserved for the clone. The cloning operation is shown figure 8.

```
clone(ag) =
PRE
        ag ∈ agents
THEN
        ANY clone WHERE
                clone ∈ AGENTS \agents
        THEN
                agents := agents∪ {clone} ||
                state(clone) := state(ag) ||
                transition(clone) := transition(ag)
        END
END
```

Figure 8: cloning operation

There is another way to modify the agent population : the death of an agent. When an agent dies, of course, we must remove it from the set of agents in our system. Moreover, to preserve invariant properties, we also have to remove its transistion table and its current state from the data we manage. If it was enough, we would obtain the specification shown figure 9[3].

```
die(ag) =
PRE
        ag ∈ agents
THEN
        agents := agents \{ag}||
        state := {ag} ⩤ state ||
        transition := {ag} ⩤ transition ||
        pendingMessages := messages
END
```

Figure 9: die, first version

This operation is however not so simple. Indeed, some pending messages were possibly sent to the dying agent. So, we have to remove the fact that these messages where sent to the dying agent. In order to achieve this operation, a new set of pending messages will be created. This new set will include all the messages from the old set minus the receipient removed, and only those messages. Moreover, if a message was only sent to the dying agent, this message must be removed. The complete code of the operation is shown figure 10.

4.3 Agent Action

At any moment, an agent can act (we use a model of parallelism with interleaving). An agent can act when it receives a stimulus (reactivity) or when it decides to (proactivity). In our model, we introduced the `NOTHING` stimulus. In our model, we introduced the `NOTHING` stimulus, so, a unique operation can be used to decribe reactivity and proactivity of the agents.

To act, an agent must have received a stimulus t (it can be the `NOTHING` stimulus) and must be in a state s where this stimulus can be taken into account, ie.

[3] $g = a \lhd f$ represents the anti-resticition of the f function to the a set, ie. the g function is equal to the fonction f restricted to its domain minus the set a

```
die(ag) =
PRE
        ag ∈ agents
THEN
        ANY messages WHERE
                messages ⊆ (STIMULUS * 𝒫(agents)) ∧
                dom(messages) = dom(messagesEnAttente) ∧
                ∀(xx,yy).(xx ↦ yy ∈ messages ⇒
                        ag ∉ yy ∧
                        (xx ↦ yy ∈ pendingMessages or xx ↦ (yy ∪ {ag}) : pendingMessages)) ∧
                ∀(xx,yy).(xx ↦ yy ∈ pendingMessages ∧ yy ≠ {ag} ⇒ xx ↦ (yy - {ag}) : messages) ∧
                {} ∉ ran(messages)
        THEN
                agents := agents \{ag}||
                state := {ag} ⩤ state ||
                transition := {ag} ⩤ transition ||
                pendingMessages := messages
        END
END
```

Figure 10: die, second version

a transition from the state s on the stimulus t must exist.

At last, the message received must be removed from the list of the pending messages, the state of the agent must be changed, and a message can potentially be sent to other agents. This is illustrated on figure 11.

5 Proof of the specification

A formal specification of a system is useful only if we try to prove it. In our context, we have to realize invariance proofs. So, here is what we have to prove :

- at any time an agent must be in a valid state ;
- any agent of the system has a valid transition table ;
- any message sent is meaningful for the system and is sent to a subset of the agents present in the system.

There are 42 proof obligations (theorems) to prove to establish properties described before. Among these one, 23 are “trivial”. Among the 19 leaving, 17 are proved automatically by the CASE tool we use (namely, the Atelier B).

In ordre to prove the 2 other proof obligations, we must use the *interactive* mode of the prover : this a man-machine interface to the prover, letting the user guiding it. So, the proof is still made by the prover (and can be considered as correct if we consider the prover correct). Here are the two theorems that need the interactive mode :

- when an agent dies, there is no more pending message sent to him ;

- we an agent act, the new set of pending messages is a valid set of messages sent to agents present in the system.

The first theorem is easily proved (one step is enough). The second one is a little more complicated, because it requires to prove two intermediate lemmas (the proof is made in 11 steps). However, the global proof of the specification is achieved.

6 Conclusion and future works

In this article, we present how the B method can be used to specify multiagent systems. But the main aspect of this article is that we obtained a general formal definition of a multiagent system. Then, this general definition can be specialized through the refinement steps.

Through the refinement process, proofs can be performed at different levels. At the moment, only abstract proofs have been achieved. However, even at this level, some properties can be specified.

Other works hav been published on the formal specification of multiagent systemes. A survey can be found in [Chaib-draa, 1997]. These works are classified in [d'Inverno *et al.*, 1996], where three types of works are determined : works around the Z method, around temporal logic and on the combination of different logics. So, why is our work original ?

Most existing research concern the formal specification of agents [Brazier *et al.*, 1997; d'Inverno *et al.*, 1998], whereas our interest is about the specification of a multiagent system. Moreover, many people use the Z method [d'Inverno *et al.*, 1998; d'Inverno and Luck, 1996; Luck and d'Inverno, 1995b]. We prefer the B method, and this difference is important, because the B method give the set of proof obligations to achieve to prove the correctness of the specification, whereas this is not clear in Z [Spivey, 1987; Abrial, 1996]. Moreover, the language is more structured than Z, making the existance of an industrial tool such as the Atelier B, a powerfull CASE tool with a prover, possible. The refinement process is also clearer in B. Finally, recent works exist on the usage of the B method to specify distributed systems [Mermet and Méry, 1997a; 1997b] and on the extension of the B method to prove temporal properties in distributed systems [Mermet, 1997].

Other approaches use Petri nets, and especially colored Petri nets [El-Fallah-Seghrouchni *et al.*, 1999]. A model oriented method, such as B, makes the specification of big system easier, and the proof on big system remain possible.

In the future, we will extend the work presented here by a first refinement step corresponding to given architectures such as eco-resolution, ants algorithms, etc. At this level, new properties should be established. Moreover, we will adapt works presented in [Mermet, 1997] on the proof of liveness properties, in order to perform proofs on the emergence of solution when using multiagent systems to distributely solve problems.

```
action =
ANY agent, stimulus, newd_state, mess, oldReceivers, newReceivers, messageRecieved,
deltaMessages WHERE
/* typing */
    agent ∈ agents ∧ stimulus ∈ STIMULUS ∧
    /* stimulus recieved and the agent that recieved it */
    new_state ∈ STATES ∧ mess ∈ STIMULUS ∧
    /* next state and stimulus to be sent */
    oldRecievers ⊆ agents ∧
    /* receivers of the message to send (mess) */
    newRecievers ⊆ agents ∧
    /* recievers of the sent message */
    messageRecieved ∈ STIMULUS * 𝒫(agents) ∧
    /* the pair: (stimulus, receivers) */
    deltaMessages ⊆ STIMULUS * 𝒫(agents) ∧
    /* the new pair(stimulus,recievers) for the recieved message after it has been recieved */

/* link stimulus - agent - messageRecieved */
    messageRecieved ∈ pendingMessages∧
    messageRecieved = (mess,oldRecievers)∧
    agent ∈ oldRecievers∧
    (oldRecievers = {agent} ⇒ deltaMessages = ∅) ∧
    (oldRecievers ≠ {agent} ⇒ deltaMessages =
{mess ↦ (oldRecievers \ {agent})})∧

/* the action itself */
        ((new_state,mess) ∈ transition(agent)[{state(agent) ↦ stimulus}]
        ∨ ( new_state = state(agent) ∧ mess = NOTHING))∧
    (new_state = state(agent) ⇒
            (state(agent) ∈ dom(transition(agent)[{state(agent) ↦ stimulus}]) ∨
        transition(agent)[{state(agent) ↦ stimulus}] = ∅))∧
    (newRecievers = ∅ ⇔ (mess = NOTHING))
THEN
    state(agent) := new_state ||
    pendingMessages := pendingMessages \ {messageRecieved} ∪ deltaMessages ∪ ({NOTHING} ⩤ {mess ↦ newRecievers})
END
```

Figure 11: action of an agent

References

[Abrial, 1996] Jean-Raymond Abrial. *The B-Book.* Cambridge University Press, 1996.

[Back and Sere, 1993] R.J.R. Back and K. Sere. Action Systems with Synchronous Communication. Technical Report 142, Abo Akademi, 1993.

[Bouron, 1992] T. Bouron. *Structure de Comunication et d'Organisation pour la Coopération dans un Univers MultiAgents.* PhD thesis, Université de Paris VI, 1992.

[Brazier *et al.*, 1997] F.M.T. Brazier, P.A.T. van Eck, and J. Treur. *Simulating Social Phenomena*, volume 456, chapter Modelling a Society of Simple Agents : from Conceptual Specification to Experimentation, pages pp 103–109. Lecture NOtes in Economics and Mathematical Systems, 1997.

[Chaib-draa, 1997] B. Chaib-draa. Formal Tools for Multi-Agent Systems. Technical report, Département Informatique, Université de Laval, 1997.

[Chandy and Misra, 1988] K. Mani Chandy and Jayadev Misra. *Parallel Program Design : A Foundation.* Addison-Wesley, 1988.

[d'Inverno and Luck, 1996] Marc d'Inverno and Michael Luck. A Formal View of Social Dependence Networks. In *Distributed Artificial Intelligence and Modelling : Proceedings of the First Australian Workshop on Distributed Artificial Intelligence*, pages 115–129. Springer-Verlag, LNAI 1087, 1996.

[d'Inverno *et al.*, 1996] Mark d'Inverno, Michael Fisher, Alessio Lomuscio, Michael Luck, Maarten de Rijke, Mark Ryan, and Michael Wooldridge. Formalisms for Multi-Agent Systems. Technical report, FORMAS'96, 1996.

[d'Inverno *et al.*, 1998] Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A Formal Specification of dMARS. In *Intelligent Agent IV : Proceedings of the Fourth International Workshop on Agent Theories, Architectures and Languages*, pages 155–176. Springer-Verlag, LNAI 1365, 1998.

[El-Fallah-Seghrouchni *et al.*, 1999] A. El-Fallah-Seghrouchni, S. Haddad, and H. Mazouzi. A Formal Study of Interactions in Multi-agent Systems. In *14th ISCA-CATA*, 1999.

[J. and J., 1991] Erceau J. and Ferber J. L'Intelligence Artificielle Distribuée. *La Recherche*, (233):750–758, 1991.

[Luck and d'Inverno, 1995a] Michael Luck and Mark d'Inverno. A Formal Framework for Agency and Autonomy. In *First International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press / MIT Press, 1995.

[Luck and d'Inverno, 1995b] Michael Luck and Mark d'Inverno. Structuring a Z Specification to Provide a Formal Framework for Autonomous Agent Systems. In *ZUM'95 : The Z Formal Specification Notation*, pages 47–62. Springer-Verlag, LNCS 967, 1995.

[Mermet and Méry, 1997a] Bruno Mermet and Dominique Méry. Safe Combination of Services using B. In Peter Daniel, editor, *Safecomp'97, the 16th International Conference on Computer Safety, Reliability and Security.* Springer, 1997.

[Mermet and Méry, 1997b] Bruno Mermet and Dominique Méry. Spécification de services et gestion des interactions. *Lettre B*, 2, 1997.

[Mermet, 1997] Bruno Mermet. *Qualité de Service dans une Logique Temporelle Compositionnelle.* PhD thesis, Université Henri Poincaré - Nancy 1, 1997.

[Mike, 1999] Wooldridge Mike. *Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence*, chapter Intelligent Agents. MIT Press, 1999.

[Spivey, 1987] J. M. Spivey. *Understanding Z : a specification language and its formal semantics.* Cambridge University Press, 1987.

[Stéria Méditerrannée, 1997] Stéria Méditerrannée. Atelier B, Guide de l'utilisateur v3.0. Technical report, Stéria Méditerranée, 1997.