A Pattern based Modelling for Self-Organizing Multi-Agent Systems with Event-B

Zeineb Graja[1 2], Frédéric Migeon[2], Christine Maurel[2], Marie-Pierre Gleizes[2],
Linas Laibinis[3], Amira Regayeg[1] and Ahmed Hadj Kacem[1]
[1]*ReDCAD, Faculty of Economics and Management,University of Sfax, Tunisia*
[2]*IRIT, Paul Sabatier University, Toulouse, France*
[3]*Åbo Akademi University, Turku, Finland*
{*zeineb.graja,frederic.migeon,christine.maurel,marie-pierre.gleizes*}*@irit.fr,*
linas.laibinis@abo.fi,{*amira.regayeg,ahmed.hadjkacem*}*@fsegs.rnu.tn*

Keywords: Self-organizing MAS, Formal verification, Design patterns, Refinement, Event-B

Abstract: Self-Organizing Multi-Agent Systems (SO-MAS) are defined as a set of autonomous entities called agents interacting together in order to achieve a given task. Generally, the development process of these systems is based on the bottom-up approach which focuses on the design of the entities individual behavior. The main question arising when developping SO-MAS is how to insure that the designed entities, when interacting together, will give rise to the desired behavior?. Our proposition to deal with this question is to use formal methods. We propose a correct by construction method for systematic design of SO-MAS based on the use of design patterns and formal stepwise refinements. Our work gives guidelines to assist the designer when developing the individual behavior of the entities and prove its correctness at the early stages of the design process. The method is illustrated with the foraging ants case study.

1 INTRODUCTION

With the growing complexity of today's applications, Self-Organizing Multi-Agent Systems (SO-MAS) are becoming more and more popular in the software engineering domain. The main characteristic properties of these systems are decentralised control, robustness and adaptability. Such qualities are very relevant when designing complex applications since they allow the system to overcome from perturbations and continue its execution autonomously and without any external control. Self-Organizing Multi-Agent Systems (SO-MAS) are defined as a set of autonomous entities called agents, having a local knowledge about their environment and interacting together in order to achieve a given task. The global behavior of the overall system emerges from the interactions between the entities and their interaction with the environment ((Di Marzo Serugendo et al., 2005)). We can dsitinguich two levels in a SO-MAS; the micro level corresponding to the local behavior of agents and the macro level referring to the global behavior of the system.

Generally, the development of SO-MAS is based on the bottom-up approch which focuses on the design of the individual behaviour of the entities composing the system. In order to validate the designed local behavior; i.e. insure that the local behavior of entities will give rise to the desired system behavior, designers make use of simulation techniques. Our proposition to deal with SO-MAS validation is to take advantage of formal techniques. We define a correct by construction approach for systematic design of SO-MAS based on the use of design patterns specified with the *Event-B* formal language. The aim is to give guarantees and proofs about the individual behavior correctness at the early stages of the design process. More precisely, we define three patterns: AGP_0, GBP_0 and SOP_0. The first one gives a sequence of refinement steps allowing the design of the individual behavior of the agents and insuring its correctness. The two other patterns are devoted to prove the reachability of the desired global behavior (GBP_0) and the ability of the system to self-adapt (SOP_0). While the AGP_0 can be expressed by *Event-B* and its correctness proved directly by means of the *Rodin* platform, it was necessary to move to the Linear Temporal Logic (LTL) in order to specify the desired global properties of the system. In order to carry on the proofs of the tempo-

ral properties, we are based on the work of Hoang and Abrial described in (Hoang and Abrial, 2011).
This paper is organised as follows. Section 2 describes a background on the *Event-B* language, design patterns and *LTL*. Section 3, presents an overview of the proposed methodology for the development of SO-MAS and gives a detailed description of the design patterns used. Section 4 illustrates our work with the foraging ants example. Section 5 presents a summary of related works. Section 6 concludes the paper and draws futur perspectives.

2 BACKGROUND

2.1 The Event-B formalism

The *Event-B* formalism was proposed by J.R. Abrial (Abrial, 2010) as an evolution of the *B* language. The concept used to make a formal development is that of a *model*. A model can be a *machine* or a *context*. A context is the static part. A machine is the dynamic part of the model and allows to describe the behavior of the designed system. A machine is composed by a collection of variables v and a set of events ev_i. The variables are constrained by conditions called *invariants*. The execution of the events must preserve these invariants. An event is described as follows.

$$e_i \mathrel{\widehat{=}} \textit{any } p \textit{ where } G_i(p,v) \textit{ then } A_i(p,v,v') \textit{ end}$$

An event is defined by a set of parameters p, the guard which gives the necessary conditions for the activation of the event $G\_evi(p,v)$ and the action $A\_evi(p,v,v')$ which describes how variables v are substituted in terms of their old values and the parameters values. The action may consist in several assignements which can be either deterministic or non-detreministic. A deterministic assignement, having the form $x := E(p,v)$ replace values of variables x with the result obtained from the expression $E(p,v)$. A non-detreministic assignement can be of two forms: 1) $x :\in E(p,v)$ which arbitrarily chooses a value from the set $E(p,v)$ to assign to x and 2) $x : |Q(p,v,x')$ which arbitrarily chooses to assign to x a value that satisfies the predicate Q. Q is called a $before-after\ predicate$ and expresses a relation between the previous values v (before the event execution) and the new ones v' (after the event execution). Each event in the machine can be described as a before after predicate predicate $\mathbf{A}(p,v,v')$ obtained by the conjonction of all before-after predicates corresponding to its actions.
The variables are constrained by conditions called *invariants*.

Proof obligations Proof Obligations (PO) are associated to Event-B machines in order to prove that they satisfy certain properties. As an example, we mention the *Preservation Invariant INV* and the *Feasibility FIS* proof obligations. *INV* PO is necessary to prove that invariants hold after the execution of each event. Proving *FIS* PO means that when an event guard holds, every action can be executed.
Refinement The development of models in *Event-B* is based on the principle of refinement. This technique, allowing a *correct by construction* design, consists in adding details gradually while preserving the original properties of the system. The refinement relates two machines, an *abstract* machine and a *concrete* one. Data refinement consists in replacing the abstract variables by the concrete ones. The refinement relation is defined by a particular invariant called *gluing invariant.* The refinement of an abstract event is performed by strengthening its guard and reducing non determinism in its action. The abstract parameters can also be refined. In this case, we need to use *witnesses* describing the relation between the abstract and the concrete parameters. An abstract event can be refined by more than one event. In this case, we say that the concrete event is *split.* In the refinement process, new events can be introduced. In order to preserve the correctness of the model, we must prove that these new introduced events are *convergent.* This is ensured by the means of a *variant* -an expression or a non empty finite set- that should be decreased by each execution of the convergent events. *B-event* is supported by the *Rodin* platform [1] which provides considerable assistance to developers by automating the generation and verification of all the necessary POs.
Design Patterns In (Abrial and Hoang, 2008), Abrial defines an Event-B design pattern as "*a small model (with constants, variables, invariants, and events) devoted to formalise a typical well known sub-problem*". A design pattern is seen as a template of a solution for a given problem that can be reused. Reusability involves not only the model itself, but also the proofs and the refinement associated with it (Hoang et al., 2013). The pattern reuse requires essentially two steps which are 1) the matching of the pattern specification with the problem and 2) the incorporation of the refinement of the pattern to create a refinement of the problem (Hoang et al., 2013).

2.2 Linear Temporal Logic

We give a brief description of the Linear Temporal Logic (LTL) as described by Manna and Pnueli (Manna and Pnueli, 1984). This logic was proposed

[1]http://www.event-b.org/

for expressing temporal properties of concurrent systems. It extends propositional logic based on the Boolean operators: $\neg$, $\vee$, $\wedge$, $\Rightarrow$ by temporal operators: always ($\Box$), eventually ($\Diamond$) and Until ($\mathcal{U}$). An LTL formula can describe the system state evolution through the time. We denote by ϕ an LTL formula and by σ a non empty sequence of states $s_0,\ s_1,\ldots$. We denote by σ^k the sequence of states $s_k,\ s_{k+1},\cdots$ and by $\sigma\ \models\ \phi$ that ϕ is true on σ. The semantic of temporal operators is as follows.

- $\sigma \models \Box\,\phi$ iff for all $k = 0,1,...,$ we have $\sigma^k \models \phi$
- $\sigma \models \Diamond\,\phi$ iff for some $k = 0,1,...,$ we have $\sigma^k \models \phi$
- $\sigma \models \phi_1\,\mathcal{U}\,\phi_2$ iff for some $k = 0,1,...,$ we have $\sigma^k \models \phi_2$ and $\sigma^0 \models \phi_1,\ ...,\ \sigma^{k-1} \models \phi_1$

2.3 Proving temporal properties with Event-B

In this subsection, we give a summary of the work of Hoang and Abrial ((Hoang and Abrial, 2011)) related to reasoning about liveness properties with Event-B. The trace σ of machine M is a sequence of states $s_0,\ s_1,\ ...$ where s_0 is the initial state defined by the initial variables values and for every two successive states $s_i,\ s_{i+1}$, there is an event enabled when the machine is in state s_i leading the machine to the state s_{i+1} when executed. We denote by $\mathcal{T}(M)$ the set of all the possible traces of machine M. A machine satisfay a property ϕ, denoted by $M\ \models\ \phi$, if all its traces satisfy ϕ. (Hoang and Abrial, 2011)

In the following, we give the three different classes of liveness properties considered in (Hoang and Abrial, 2011) and the necessary proof rules allowing to demonstrate them on an *Event-B* machine. These proof rules constitutes the basis of our proofs in section 3.

- The *existence* property states that a property P will *always eventually* be true ($\Box\ \Diamond\ P$). To prove that a machine M satisfies an existence property requires to prove that M is convergent in $\neg P$, i.e. every event execution in M decreases the defined variant when M is on a $\neg P$ state and deadlock-free in $\neg P$ i.e. when M is in a $\neg P$ state, at least one event of M is enabled. The necessary assumptions for proving the existence property are given by the rule $\textbf{LIVE}_{\Box\Diamond}$ [2].

$$\dfrac{\begin{array}{l} M\ \vdash\ \downarrow\neg P \\ M\ \vdash\ \circlearrowleft\neg P \end{array}}{M\ \vdash\ \Box\Diamond P}\quad \textbf{LIVE}_{\Box\Diamond}$$

- The *progress* property states that a property $P2$ must *eventually* be true if some condition $P1$ becomes true ($\Box\ (P1\ \Rightarrow\ \Diamond\ P2)$). Proving that a machine M fulfils a progress property is insured by the use of the rule $\textbf{LIVE}_{progress}$.

$$\dfrac{\begin{array}{l} M\ \vdash\ \Box(P1 \wedge \neg P2 \Rightarrow P2) \\ M\ \vdash\ \Box(P3 \Rightarrow (P3\ \mathcal{U}\,\mathrm{P}2)) \end{array}}{\Box\ (P1\ \Rightarrow\ \Diamond\ P2)}\quad \textbf{LIVE}_{progress}$$

The first premise should be declared as an invariant in the machine M. The second premise includes the *until* temporal operator and states that $P3$ is true until $P2$ holds. This assumption is proved by the use of the **Until** rule.

$$\dfrac{\begin{array}{l} M\ \vdash\ \Box((P3 \wedge \neg P2) \curvearrowright (P3\ \vee\ P2)) \\ M\ \vdash\ \Box\Diamond(\neg P3 \vee P2) \end{array}}{M\ \vdash\ \Box(P3 \Rightarrow (P3\ \mathcal{U}\,\mathrm{P}2))}\quad \textbf{Until}$$

The first condition in the **Until** rule means that every event in the machine M leads from $P3 \wedge \neg P2$ to $P3\ \vee\ P2$. An event leads from $P1$ to $P2$ if starting from any $P1$ state, the execution of this event results in $P2$ state. The $Leads\ from$ operator ($\curvearrowright$) is expressed directly with the first-order logic as shown below.

$P1\ \curvearrowright\ P2\ \widehat{=}\ P1(v) \wedge G(x,v) \wedge A(x,v,v')\ \Rightarrow\ P2(v')$

- The *persistence* property states that *eventually* a property P will be *always* true ($\Diamond\Box P$). The proof rule $\textbf{LIVE}_{\Diamond\Box}$ gives the necessary conditions guaranteeing that a machine M satisfies a persistence property. A machine M satisfies a persistence property $\Diamond\Box P$ if it is divergent in P; i.e. any infinite trace of M ends with an infinite sequence of states satisfying P and deadlock-free in $\neg P$. Proving that a machine M is divergent in P needs to prove that every execution of an event in M decreases the defined variant when M is on a $\neg P$ state and does not increase the variant when M is on P state.

$$\dfrac{\begin{array}{l} M\ \vdash\ \nearrow P \\ M\ \vdash\ \circlearrowleft\neg P \end{array}}{M\ \vdash\ \Diamond\,\Box\,P}\quad \textbf{LIVE}_{\Diamond\Box}$$

3 TOWARDS A FORMAL DESIGN PROCESS FOR SO-MAS

3.1 Overview of the design process

The aim of the proposed method is to construct the adequate local behavior leading to the desired global properties by stepwise refinement and the use of design patterns. We describe the formal design process in terms of three phases as depicted in figure 1. In order to guide the designer through the refinement process, a design pattern is assigned to each phase giving the necessary refinements and proof obligations to attain the correct model at the end of each step.

[2]The upper part of the proof rule describes the premises under which the conclusion mentioned in the lower part is true. The rule's name is given at the right.

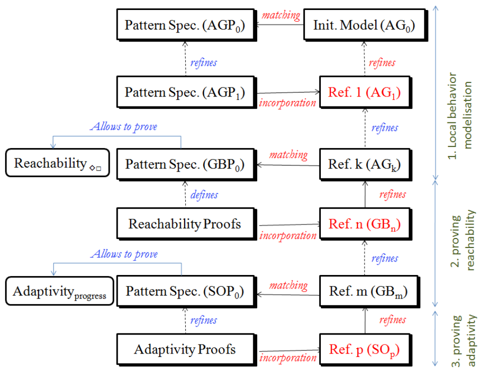

Figure 1: Stepwise refinement for formal development of SO-MAS.

The first phase allows modelisation of the agents local behavior based on the pattern AGP_0. It is composed by several refinement steps as it will be shown in the next subsection. This phase can be performed for several times if the system is composed by many types of agents. The next two phases are devoted to prove convergence and adaptivity properties by using the patterns GBP_0 and SOP_0 respectively. These two patterns can be reused in the case where convergence can take many aspects and adaptivity is needed in many situations.

3.2 Design patterns for SO-MAS

In this section, we define in details the design patterns AGP_0, GBP_0 and SOP_0. For each pattern, we give an informal description plus a formel specification with *Event-B*. We also mention how it can be refined and what proof obligations must be discharged.

3.2.1 Agents pattern:AGP_0

The Agents pattern gives a very abstract modelisation of the designed system as a set of agents in interaction within an environment. Each agent perceives its local environment, takes a decision about its next action and finally performs the decided action. We refer to these execution steps as a $perceive - decide - act$ cycle. We assume that the system has the following properties: 1) an agent can either be active or inactive, 2) every active agent is initialized in the *perceive* step, 3) the system execution is considered as an alternation of an agents execution and an environment activation; that is when every agent finishes one $perceive - decide - act$ cycle, the environment is activated and 4) an active agent can either be in the *pause* mode; waiting for beginning a new cycle or in the *work* mode; executing its cycle. The pattern AGP_0 is thus an Event-B machine describing the system state by means of the set of active agents, the mode of each active agent (pause or work) and the actual cycle step for each active agent. In the following, the set of the variables used and their corresponding invariants are given [3].

```
VARIABLES
   agStep
   agMode
   ActiveAgents
INVARIANTS
   inv1 : ActiveAgents ⊆ Agents
   inv2 : agMode ∈ ActiveAgents → {work, pause}
   inv3 : agStep ∈ ActiveAgents → {perceive, decide, act}
```

The dynamic of the system is modelled by means of four events. The $ActEnv$ event is triggered when it is the turn of the environment to be activated. At this level of abstraction, the only action that this event does is to reset the active agents in the system at the step of perception. Events *Perceive*, *Decide* and *Act* model the execution of any agent according to the $perceive - decide - act$ cycle. As an example, we give the *Perceive* and the $ActEnv$ events [4].

```
EVENT Perceive
   ANY
      ag
   WHERE
      grd1 : ag ∈ ActiveAgents
      grd2 : agMode(ag) =
            work
      grd3 : agStep(ag) =
            perceive
   THEN
      act1 : agStep(ag) :=
            decide
END
```

```
EVENT ActEnv
WHERE
grd1 :
∀ag·ag ∈ ActiveAgents
 ⇒agMode(ag) = pause
THEN
act1 :
ActiveAgents, agMode, agStep : |
ActiveAgents′ ∈ ℙ(Agents)∧
ActiveAgents′ ≠ ∅∧
(∀ag·ag ∈ ActiveAgents′⇒
agMode′(ag) = work∧
agStep′(ag) = perceive)
END
```

Proof obligations generated for this pattern insures essentially invariants preservation and feasibility of the non deterministic action of the $ActEnv$ event.
The AGP_0 pattern will be subject to a three steps refinement sequence to obtain a more concrete agents behavior. In the first step, the *Act* event is split into the different actions that an agent can perform. Every event resulting from the refinement of *Act* event can have the following template.

[3] $S \to T$ denotes a total function from S to T, i.e. a relation that maps every element in S to an element in T.

[4] The symbol $\mathbb{P}$ denotes the power set.

```
EVENT Act_Action_i
REFINES Act
  ANY
    $ag$
    decideAct
  WHERE
    $grd1 : ag \in ActiveAgents \wedge agMode(ag) = work \wedge$
    $agStep(ag) = act$
    grd2 : decideAct=Action_i
  THEN
    $act1 : agMode(ag) := pause$
END
```

In the second refinement step, agents's actuators are introduced. The agent's actuators should be disabled when the agent move to the state *pause*. This property is insured by adding the gluing invariant:

$$\forall ag.\ ag \in ActiveAgents \wedge agMode(ag) = pause \Rightarrow actuAction\_i(ag) = disabled$$

In addition, *Decide* events are split in turn. When an agent takes a decision, it activates the suitable actuator in order to perform the desired action. To link the agent action with the made decision, we use a witness. So that, the events modelling the action need also to be refined according to the refinement of the event Act_Action_i from AGP_0 pattern.

```
EVENT Dec_Action_i
REFINES Decide
  ANY
    $ag$
    decideAct
  WHERE
    $grd1 : ag \in$
        $ActiveAgents$
    $grd2 : agMode(ag) =$
        $work$
    $grd3 : agStep(ag) =$
        $decide$
    grd4 : decideAct=
        Action_i
  THEN
    $act1 : actuAction\_i(ag)$
        $:= activate$
END
```

```
EVENT Act_Action_i
REFINES Act_Action_i
  ANY
    $ag$
  WHERE
    $grd1 : ag \in ActiveAgents$
    $\wedge agMode(ag) = work$
    $\wedge agStep(ag) = act$
    grd2:actuAction_i(ag)
        =activate
  WITNESSES
    decideAct :decideAct=
        Action_i
$\Leftrightarrow$actuAction_i(ag)=activate
  THEN
    $act1 : agMode(ag) := pause$
    $act2 : actuAction\_i(ag)$
        $:= disabled$
END
```

At this refinement step, we should insure that once an agent made a decision, it should execute an action and avoid to be deadlocked in the action step. This property is specified by the following theorem. $G_{Act\_Action\_i}(ag, p, v)$ denotes the guard of an action event for the agent ag.

$$\forall ag.ag \in ActiveAgents \wedge agStep(ag) = act \Rightarrow (\bigvee_i (\exists p.G_{Act\_Action\_i}(ag, p, v)))$$

In the last refinement step , the agents's sensors are introduced and the event *Perceive* is refined. For each agent, it is necessary that its sensors are active when it is in the *perceive* step. This constraint is captured by the gluing invariant:

$$\forall ag.\ ag \in ActiveAgents \wedge agStep(ag) = perceive \Leftrightarrow sensor\_i(ag) = activate.$$

Moreover, the action in the event *ActEnv* is refined by activating the sensors of each active agent. At this refinement phase, we should insure that the updated perceptions, should allow the agent to make a decision and thus to avoid to be deadlocked in the perception step. This property is specified by the following theorem. $G_{Dec\_Action\_i}(ag, p, v)$ denotes the guard of a decision event for the agent ag.

$$\forall ag.ag \in ActiveAgents \wedge agStep = decide \Rightarrow (\bigvee_i (\exists p.G_{Dec\_Action\_i}(ag, p, v)))$$

3.2.2 Global Behavior pattern:GBP_0

The Global Behavior pattern allows to reason about the behavior that emerges from the interactions between agents. It is used to prove convergence of the system, which means reachability of the desired global behavior. Convergence of the system can be captured formally by means of the *Reach* temporal property: Reach $\widehat{=} \Diamond \Box\, taskAchieved = TRUE$

taskAchieved describes the state of the system when it succeed to achieve its task. The modelisation of this property with Event-B can be done according to the pattern GBP_0.

```
$GBP_0$
  VARIABLES
    $taskAchieved$
    $SysStates$
  INVARIANTS
    $SysStates \subset SYS\_STATES$
    $taskAchieved \in BOOL$
  VARIANTS
    $V$
```

```
EVENT NotYetSuccess
STATUS $convergent$
  ANY   $ag$   WHERE
    $grd1 : taskAchieved$
        $= FALSE$
    $grd2 : V \notin \varnothing$
    $grd3 : ag \in Agents$
  THEN
    $act1 : SysStates :\in V' < V$
END
```

```
EVENT ObserveSuccess
  WHERE
    $grd1 : taskAchieved = TRUE$
  THEN SKIP
END
```

Variable *SysStates* denotes the system state space. Variable *taskAchieved*, when is TRUE, indicates that the global task is achieved and allows to activate *ObserveSuccess*. This event plays the role of an external observer (like in (Hoang et al., 2009))and does not change the system state. In fact, its action is SKIP which does nothing. The event *NotYetSuccess*

is activated when the task is not yet achieved, but must contribute to the fulfilment of the global task by decreasing at each execution the variant V.
According to the rule **LIVE**$_{\Diamond\Box}$, to prove the *Reach* property, we need to prove the convergence of event *NotYetSuccess*. This is guaranteed by discharging the *VAR* proof obligation. Moreover, we should prove that the event *ObservSuccess* does not increase the variant and that the machine is deadlock free for all the states where the task is still not fulfilled. The first statement is guaranteed since the action of the event *ObservSuccess* is *SKIP*. The second statement is ensured by proving the following theorem stating that from an intermediate state, the machine can evolve either to another intermediate state or to the success state.

$$taskAchieved = FALSE \Rightarrow ((\exists ag, V.ag \in ActiveAgents \wedge V \notin \varnothing) \vee taskAchieved = TRUE)$$

The incorporation of this pattern in the design process allows to refine *AG_k* (Figure 1) with *GB_n*. It is performed by two actions: 1)introducing event *ObserveSuccess* and 2)refining each *Act* event (an event describing an agent action) with the *NotYetSuccess* event. At this level, we should prove convergence of all the *Act* events and the deadlock freeness of *GB_n* in a non desirable state. Since Event-B allows to use only one variant per machine, *GB_n* could be refined in many steps. In each step, the convergence of one event (or a group of events which decrease the same variant) is proved. The events which are not considered by the proof at a given step, must be *anticipated*, this means that proving their convergence is postponed for further refinement steps. Choosing the suitable variant to prove convergence is not always trivial with SO-MAS. In fact, the evolution of the agents can not always be described as a progression twoards fulfillment of their goals. An agent can change its goal according to the actions of the other agents. In this work, we don't address this problem of proving convergence. But this is still an ongoing work.

3.2.3 Self-Organisation pattern:SOP_0

The Self-Organisation pattern allows to reason about the ability of the system to self-adapt in order to overcome perturbations in its environment. A rigorous analyzis of self-organisation can be captured by the use of the *Adaptivity* temporal formula stating that if a perturbation occurs, the system will eventually be able to carry on its execution thanks to its self-organization mechanisms.

$$\text{Adaptivity} \mathrel{\widehat{=}} \Box(perturbation = TRUE \Rightarrow \Diamond SuccessSO = TRUE)$$

The pattern SOP_0 (given below) [5] allows to reason about this property by applying **LIVE**$_{progress}$ rule and proving the following two theorems.
Theorem1.

$$\Box(perturbation = TRUE \wedge SuccessSO = FALSE \Rightarrow (\exists ag.\ ag \in ActiveAgents \wedge V \notin \varnothing))$$

Theorem2.

$$\Box(\exists ag.\ ag \in ActiveAgents \wedge V \notin \varnothing \Rightarrow ((\exists ag.\ ag \in ActiveAgents \wedge V \notin \varnothing)\, \mathcal{U}\, \text{SuccessSO=TRUE}))$$

According to **Until** rule, the demonstration of **Theorem2** needs to prove the following two theorems.
Theorem2.1.

$$\exists ag.\ ag \in ActiveAgents \wedge V \notin \varnothing \wedge SuccessSO = FALSE \Rightarrow ((\exists ag.\ ag \in ActiveAgents \wedge V \notin \varnothing) \vee SuccessSO = TRUE)$$

Theorem2.2.

$$\Box\Diamond((\forall ag.ag \in ActiveAgents \wedge V \in \varnothing) \vee SuccessSO = TRUE)$$

By applying the rule **LIVE**$_{\Box\Diamond}$, the proof of **Theorem2.2** necessitates 1) to prove the convergence of the event *ApplySO* and 2) to prove deadlock-freedom in a state satisfying the property $\exists ag.\ ag \in ActiveAgents \wedge V \notin \varnothing \wedge SuccessSO = FALSE$.
The incorporation of this pattern in the design process allows to refine *GB_m* (Figure 1) with *SO_p*. It is performed by three actions: 1) introducing the event *PerturbationOccurs* performed when a perturbation takes place in the environment, 2) adding the *ObserveSO_Success* event describing the success of a self organization operation and 3) refining the agent actions by the event *ApplySO* which modelises the self-organizing mechanism. **Theorem1** and **Theorem2** should be proved for every action refining the event *ApplySO*.

```
SOP_0
  VARIABLES
    SuccessSO
    SysStates
    pertturbation
  INVARIANTS
    SysStates ⊂ STATES
    SO_STATES ⊂ STATES
    SuccessSO ∈ BOOL
    perturbation ∈ BOOL
  VARIANTS
    V
EVENT ObserveSO_Success
  WHERE
    grd1 : SuccessSO = TRUE
  THEN SKIP
END
```

```
EVENT ApplySO
STATUS convergent
  ANY     ag   WHERE
    grd1 : SuccessSO =
           FALSE
    grd2 : V ∉ ∅
    grd3 : ag ∈ ActiveAgents
  THEN
    act1 : SysStates :∈ V' < V
END
EVENT PerturbationOccurs
  REFINES ActEnv
  THEN
    act1 : SysStates ∈
             SO_STATES
    act2 : perturbation :=
             TRUE
END
```

[5] *SO_STATES* is the set of states where the self-organization mechanism is needed to overcome the perturbation.

4 APPLICATION ON THE FORAGING ANTS

The case study is a formalization of the behavior of a foraging ants colony. The considered system is composed of several ants exploring the environment and looking for food.Each ant begins by exploring the environment being mainly attracted by food or pheromone. If it smell pheromone, it takes the direction in which this smell is stronger. When discovering food on a location, the ant collects a part of it and goes back to the nest by dropping pheromone.
The properties we are trying to prove are summarized as follows.
- The correctness of the model of the agents behavior; i.e. $Corr_1$: Each ant behaves according to the $perceive - decide - act$ cycle, $Corr_2$: Deadlock freeness of the ants in any step of its cycle, $Corr_3$: The ants must avoid obstacles.
- $Reach_1$ The ants are able to bring all the food to the nest. This is the main property of the system: the reachability property.
- SO_2: When a source of food is detected, the ants are able to focus on its exploitation. This property evaluates the ability of the ants to self-organise in order to brought entirely the detected source of food to the nest.
- SO_1: When a detected source of food is completely exploited, the ants can carry on environment exploration and detect new food.
In order to guarantee the correctness of our model we apply the pattern AGP_0. The $Reach_1$ property is modelled by applying GBP_0. Self-organisation properties are modelled by applying twice the pattern SOP_0. In the remainder of this section, we illustrate the use of the AGP_0 pattern. For the three other properties ($Reach_1$, SO_2 and SO_1), we give a formulation of each of them in the temporal logic. Their proofs are an ongoing work.
The environment in which the ant interact is formalized as a set of connected locations (*Locations*) with a particular one called *Nest* modelling the nest of the colony. Guided by the refinement steps indicated in AGP_0, we obtain a machine modelling the local behavior of ants. Each ant has a current location in the grid (*currentLoc*) and can decide about its next location (*nextLoc*). The ant has information about the environment elements which are inside its perception field, i.e. food (*food*), pheromone (*pheromone*) and obstacles (*obstacles*). The definition of these caracteristics in Event-B are given as follows.

$$
\begin{array}{l}
inv1 : currentLoc \in Ants \rightarrow \mathbb{P}(Locations) \\
inv2 : nextLoc \in Ants \rightarrow \mathbb{P}(Locations) \\
inv3 : food \in Ants \rightarrow \mathbb{P}(Locations) \\
inv4 : pheromone \in Ants \rightarrow \mathbb{P}(Locations)
\end{array}
$$

The invariant $inv1$, for example, is a total function which defines the current location for each ant. *Ants* is the set of the active Ants. The ants behavior is depicted by the following events:
- *Perceive*: enables to each ant to update its perceptions according to its current location.
- *Dec_Move_Rand*: the ant decide to go randomly because it does not perceive any thing, *Dec_Move_Food*: decide to follow sensed food, *Dec_Move_Phero*: decide to follow sensed pheromone, *Dec_Drop_Back*: decide to go back to the nest and drop pheromone along the return path, *Dec_Back*: decide to go back to the nest without dropping pheromone along the return path, *Dec_Harvest_Food*: decide to take food, *Dec_Drop_Food*: decide to drop the food at the nest.
- *Act_Move_Rand*,*Act_Move_Food*, *Act_Move_Phero*, *Act_Drop_Back* and *Act_Back*: activated when an ant moves from one location to another.
- *Act_Harvest_Food*: the ant takes some food and *Ac_Drop_Food*: the ant drop the food on the Nest.
As an example, we give the description of the event *Act_Move_Food*.

```
EVENT Act_Move_Food    REFINES Act_Move_Food
  ANY
    ant
  WHERE
    grd1 : $ant \in Ants \wedge agMode(ant) = work \wedge agStep(ant) = act$
    grd2 : $paw(ant) = activate$
    grd3 : $actMove(ant) = Follow\_Food$
  THEN
    act1 : $currLoc(ant) := nextLoc(ant)$
    act2 : $agMode(ant) := wait$
END
```

The reachability property is defined as follows.

$$
\begin{array}{l}
\Diamond(\Box(QuantityFood(Nest) = TotalFood(InitDistFood)\wedge \\
\forall\ loc.loc \in Locations\backslash\{Nest\} \Rightarrow QuantityFood(loc) = 0)).
\end{array}
$$

where the $QuantityOfFood$ relation specifies for each location the quantity of food in it, $TotalFood$ calculates the sum of quantities of food in the environment and $InitDistFood$ is the initial distribution of food.
The SO_1 property is defined by the following temporal formula.

$$
\begin{array}{l}
\Box(\forall\ loc.loc \in Locations\backslash\{Nest\}\wedge \\
InitDistFood(loc) \neq 0 \wedge Detected(loc) \Rightarrow \\
\Diamond((QuantityFood(loc) = 0)).
\end{array}
$$

The SO_2 property is defined by the following temporal formula.

$\square(\forall\, ant, loc.\ ant \in Ants \wedge loc \in Locations \setminus \{Nest\} \wedge$
$EntirelyExploited(loc1) \Rightarrow$
$\Diamond(\exists loc2.\ loc2 \neq loc1 \wedge QuantityFood(loc2) \neq 0 \wedge Detected(loc2)).$

5 RELATED WORK

Developpement methods. In (Orfanus et al., 2011), a design process for the construction of emergent self-organizing behavior in large-scale distributed embedded systems is proposed. This process relies on two models: the model of microscopic layer and the model of macroscopic layer and three steps: simulation, validation and update. The first model should describe the local activities of the entities composing the system as well as the local information trigerring these activities. The second one is described by a set of Macroscopic variables. The simulation step, applied to a given model of microscopic behavior under certain scenario, allows to generate a macroscopic model called *Current macro*. The current macro if then validated in the next step by comparing the set of the current macroscopic variables with the desired macroscopic ones. The comparison is done thanks to a $Matchfunction$ and returns a *Deviation* which, if needed, will be used in order to update the microscopic layer model.
A top-down property driven design for swarm robotic, composed by four phases, was proposed in (Brambilla et al., 2012). The aim of the first phase is to give a clear and complete formal specification of the desired properties of the system. The second phase consists in defining a macroscopic model of the system and checking whether the desired properties are satisfied. These two activities of modelling and validation can be repeated until generating a model satisfying the desired properties. The phase three is devoted to the implementation of a simulation of the swarm system. This phasee represents a transition from the macroscopic model to the microscopic implementation and is guided by the ingenuity and the expertise of the designer (Brambilla et al., 2012). In the final phase, the system is deployed on real robots. The authors choose to specify the swarm model by the Deterministic Time Markov Chains (DTMC) and its properties by the Probabilistic Computation Tree Logic* (PCTL*). Moreover, they use the technique of model checking and particularly the PRISM model checker for verification.
The Adelfe methodology was proposed for the development of self-organizing systems based on the *AMAS* (Adaptive MAS) theory ((Bernon et al., 2005)). The AMAS theory depends on the ability of the agents to cooperate and thus, the design of the local behavior of the entities relies on identifying and resolving the non cooperative situations an agent may encounter. In order to guarentee that all non cooperative situations are taken into account, the process of Adelfe is enriched by a simulation step in (Bernon et al., 2006).
Formal modelisation of self-organizing systems. In (Gardelli et al., 2006), *Gardelli* uses *stochastic Pi-Calculus* for modeling SO-MAS for intrusion detection. This formalization was used to perform simulations using the *SPIM* tool to assess the impact of the number of agents and frequency of inspections on the system behavior. In (Casadei and Viroli, 2009), a hybrid approach has been proposed. This approach uses stochastic simulations to model the system described as Markov chains and the technique of probabilistic model checking for verification. The approach was tested for the problem of collective sorting using the *PRISM* tool. Konur and colleagues ((Konur et al., 2012)) use also the *PRISM* tool and probabilistic model checking to verify the behavior of robot swarm, particularly foraging robots. The authors verify properties expressed by $PCTL$ logic for several scenarios. These properties provide information ,in particular, on the probability that the swarm acquires a certain amount of energy for a certain number of agents and in a certain amount of time. Simulations were also used to show the correlation between the density of foraging robots in the arena and the amount of energy gained.
An *Event-B* modeling for fault tolerant MAS was proposed in (Pereverzeva et al., 2012). The authors propose a refinement strategy that starts by specifying the main purpose of the system, defines the necessary agents to accomplish it, then introduces the various failures of agents and ends by introducing the communication model and error recovery mechanisms. The refinement process ensures a set of properties, mainly 1) reachability of the main purpose of the system, 2) the integrity between agents local information and global information and 3) efficiency of cooperative activities for error recovery. The aim of the works presented above is to insure that the designed individual behavior will give rise to the desired global properties. Some of them make use of simulation, while others employ formal techniques. The majority of these works utilize a bottom-up approach (except (Brambilla et al., 2012) and (Pereverzeva et al., 2012)) which is ideally suited to self-organizing systems. The use of Event-B in (Pereverzeva et al., 2012) is extremely important because of the use of the refinement principle that permits a progressive, guided and correct construction of the desired system, which is not allowed in the other works. In our proposition,

we combine a bottom-up approach with the use of refinement and design patterns in order to give more guidance to the designer when designing the individual behavior (*AGP_0* pattern) and when doing proofs (*GBP_0* and *SOP_0* patterns).

6 CONCLUSION

We have presented in this paper a formal approach for the design of SO-MAS based on design patterns, refinement and *Event-B*. Three patterns was proposed; *AGP_0* gives refinement steps for modelling the local behavior of the agents and guarantees deadlock freeness of any agent, *GBP_0* allowing to prove that the modelled local behavior will converge towards the desired global behavior and finally *SOP_0* letting the evaluation of the ability of self-organizing mechanisms to encounter the environment perturbations.
The main challenges for future work can be summarized in the three following points:
- Proving the convergence of the events when applying the patterns *GBP_0* and *SOP_0* which is not trivial task because of the non determinism in SO-MAS. One possible solution for this is to prove the convergence under fairness assumption like in (Méry and Poppleton, 2013).
- Automation of the refinement process and the generation of machines according to the design patterns.
- Formal reasoning about the improvement of the system performance. A probabilistic approach coupled with *Event-B* can be useful in this case.

REFERENCES

Abrial, J.-R. (2010). *Modeling in Event-B - System and Software Engineering*. Cambridge University Press.

Abrial, J.-R. and Hoang, T. S. (2008). Using design patterns in formal methods: An event-b approach. In *ICTAC*, pages 1–2.

Bernon, C., Camps, V., Gleizes, M.-P., and Picard, G. (2005). Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology . In Henderson-Sellers, B. and Giorgini, P., editors, *Agent-Oriented Methodologies* , volume ISBN1-59140-581-5, pages 172–202. Idea Group Pub, NY, USA.

Bernon, C., Gleizes, M. P., and Picard, G. (2006). Enhancing self-organising emergent systems design with simulation. In *ESAW*, volume 4457 of *Lecture Notes in Computer Science*, pages 284–299. Springer.

Brambilla, M., Pinciroli, C., Birattari, M., and Dorigo, M. (2012). Property-driven design for swarm robotics. In Conitzer, Winikoff, P. and van der Hoek, editors, *AAMAS 2012*, pages 139–146.

Casadei, M. and Viroli, M. (2009). Using probabilistic model checking and simulation for designing self-organizing systems. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 2103–2104, New York, NY, USA. ACM.

Di Marzo Serugendo, G., Gleizes, M.-P., and Karageorgos, A. (2005). Self-organization in multi-agent systems. In *Knowl. Eng. Rev.*, volume 20, pages 165–189, New York, NY, USA. Cambridge University Press.

Gardelli, L., Viroli, M., and Omicini, A. (2006). Exploring the dynamics of self-organising systems with stochastic π-calculus: Detecting abnormal behaviour in mas. In *(AT2AI5)*.

Hoang, T. S. and Abrial, J.-R. (2011). Reasoning about liveness properties in event-b. In *ICFEM*, pages 456–471.

Hoang, T. S., Fürst, A., and Abrial, J.-R. (2013). Event-b patterns and their tool support. *Software and System Modeling*, 12(2):229–244.

Hoang, T. S., Kuruma, H., Basin, D. A., and Abrial, J.-R. (2009). Developing topology discovery in event-b. *Sci. Comput. Program.*, 74(11-12):879–899.

Konur, S., Dixon, C., and Fisher, M. (2012). Analysing robot swarm behaviour via probabilistic model checking. *Robot. Auton. Syst.*, 60(2):199–213.

Manna, Z. and Pnueli, A. (1984). Adequate proof principles for invariance and liveness properties of concurrent programs. Technical report, Stanford, CA, USA.

Méry, D. and Poppleton, M. (2013). Formal Modelling and Verification of Population Protocols. In Johnsen, E. B. and Petre, L., editors, *iFM - 10th International Conference on integrated Formal Methods - 2013*, LNCS, Turku, Finlande. Springer.

Orfanus, D., Janacik, P., Eliassen, F., and Orten, P. (2011). High-level construction of emergent self-organizing behavior in massively distributed embedded systems. In *NaBIC*, pages 335–341.

Pereverzeva, I., Troubitsyna, E., and Laibinis, L. (2012). Development of fault tolerant mas with cooperative error recovery by refinement in event-b. *CoRR*, abs/1210.7035.