GDT4MAS: a formal model and language to specify and verify agent-based complex systems

Bruno Mermet, Gaële Simon

GDT4MAS: a formal model and language to specify and verify agent-based complex systems

B. Mermet [*] **, G. Simon** [*]

[*] *Laboratoire GREYC, UMR 6072 & Université du Havre*
Campus Côte de Nacre
Boulevard du Maréchal Juin
BP 5186 - 14032 CAEN cedex
Bruno.Mermet@univ-lehavre.fr
Gaele.Simon@univ-lehavre.fr

Abstract. *In this article, we briefly present the GDT4MAS model, a formal specification model dedicated to Multi-Agent Systems. We especially explain why we concieved a dedicated model and method, and how we associated to this method a few essential characteristics. We also try to explain why this model is particularly suited to complex systems. We also present the proof process provided by the model. We illustrate on a toy example the proof process of the GDT4MAS model and we show how an automatic verification can be performed thanks to a theorem prover like PVS.*

Keywords: *complex systems, multiagent systems, verification, GDT4MAS, PVS*

1. Introduction

For several years, Multi-Agent Systems (MAS) have been more and more widely used either to solve artificial intelligence problems or to model complex systems. However, their usage is mainly restricted to academic research, essentially because the way they work is hard to understand (especially when modelling complex systems) and, as a con-

sequence, they do not provide enough guarantees on their result. This is for instance discussed in [OFT04].

There are several ways to bring more confidence in MAS, such as semi-formal models (like AUML) or monitoring systems. We have chosen to focus our research on the formal specification and verification of MAS. In this research area, several strategies may be attempted:

– formalizing a standard MAS model;

– adapting a standard formal verification system to MAS;

– developing a dedicated formal MAS model with an associated verification system.

The first approach is not tractable because standard MAS models forget many concepts that are necessary when a formal verification is required. The second approach seems more feasible, because a few verification systems are supported by industrial CASE-tools (like the B-method[Abr96], supported by the Atelier B) and some of them are adapted to distributed systems (like the notion of *system* in the B-method). However, a MAS is not only a distributed systems: agents do not always work alltogether in a cooperative way to solve a shared problem. Indeed, an agent lives in an environment where other entities (agents or artefacts) evolve according to their own goals.

This statement leads us to develop a new formal model and verification system, dedicated to MAS. However, we want to re-use experienced principles, from either the MAS community or the formal verification domain. As a consequence, we must develop a model with the following underlying concepts:

– an agent's behaviour is specified by goals;

– an agent is proactive and cannot assume a given behaviour from other agents;

– formal verification is a so complicated task that it must be performed automatically as much as possible;

– first-order logic, arithmetic and set-theory are very expressive tools to specify programme behaviours. Moreover, there are several automatic provers adapted to these formalisms. Linear Temporal Logic increase the expressivity of the predicate logic;

– most important properties to guarantee the correctness of a system are invariant properties and liveness properties;

– a compositionnal proof system is the only way to obtain formulae simple enough to be automatically proven;

– the earlier a proof is attempted, the easier it is and the less expensive the correction is;

– an automatic code generation from the proven specification to an implementation language is necessary to preserve the guarantee provided by the proof.

As a consequence, specifying and verifying a MAS using our model relies on the following steps:

1) a specification is written in our language;

2) using *Proof Schemas (PS)* that we provide with the model, the specification is automatically translated into a set of predicates, called *Proof Obligations (PO)*, that must be proven to guarantee the correctness of the specification;

3) the set of PO is automatically proven with a proof verification system, such as PVS [SRI], krt [CS] or SPIN [Holcm].

Please notice also that once step 1 is performed, the code can be automatically produced, regardless of the proof process.

The GDT4MAS model is well suited to specify complex systems based on MAS for the following reasons:

– specifying numerous agents is easy to perform, as agents are parameterized instances of agent types. As a consequence, most proofs can be performed directly at the "agent-type" level, and not agent by agent;

– although proofs can be performed by model-checking, they are *theorem-prover oriented*, reducing the consequences of the number of agents on the complexity of the proof;

– proofs can be performed on agent types specified independently.

For instance, a problem of two robots having to clean the Mars planet (and initially specified in [BFVW03]) has been extended to several robots of each type, and has been proven to be correct thanks to our

method [MS09]. The number of proofs to perform depends only on the number of agent types. This seems suitable to complex systems, where the number of entities may be very huge, but the number of entity types is often quite smaller.

In a more recente article, we have extended our model to specify holonic agents [MS10, MS11]. We recall here that holonic agents are agents that can be grouped into organisations that can be seen as agents [Occ00, ASM08]. This kind of modelization is particularily well suited to model multi-scale problems, a usual situation in complex systems. However, to make the presentation of our model clearer, this extension of our model will not be presented in this paper.

In this paper, we begin by describing the GDT4MAS model, that is more widely presented in [MS09, MSSZ07]. Indeed, the goal of this article is not to present the model (it has been already done), but to show the proof mechanism, helping in understanding why it can be used on complex systems. Then, in section 3, we summarize the proof system. This is examplified on a case study presented in section 4. The case study we have chosen may surprise, as it is made of a single agent with a very simple behaviour. However, it has been selected because it allows to detail the proof steps, that are indeed very easy, but produce long formulae.

2. The GDT4MAS model

2.1. *Main concepts*

In the GDT4MAS model, the MAS is described by an environment, mainly described by variables, and a population of agents evolving in this environment. Each agent is described as an instance of an agent type. As a consequence, in the rest of this section, after a short description of the notations we used, we begin by describing the notion of agent type, and of agent behaviour.

2.2. *Notation*

Notation 2.1 (primed and unprimed variable) *When the value of a variable v in two execution states is considered, the value of v in the first state, called the current state, is written v, and its value in the second state is written v'. For instance, the action consisting in increasing the value of v by 1 is specified by the postcondition $v' = v + 1$.*

2.3. *Agent Type Specification*

Simplified Definition 2.1 (Agent Type) *An agent type t is described by a name ($name_t$), a set of internal variables ($VarI_t$), a set of surface variables ($VarS_t$), an invariant (i_t), and a behabiour (b_t).*

In this definition, an *internal variable* is a variable that only the owner agent can see and modify (compare it to a private attribute in the object model); a *surface variable* is a variable that only the owner agent can modify, but that can be seen by the other agents (compare it to a private attribute with a public getter method); an *invariant* is a predicate defined on the internal and surface variables of the agent type and that must always be true for every agent of the given type; and the *behaviour* of an agent is specified by a Goal Decomposition Tree, defined later in this section.

Simplified Definition 2.2 (Action) *An action a is specified by a name ($name_a$), a precondition (pre_a), a postcondition ($post_a$), an ns flag (ns_a) and a gpf (gpf_a). The precondition is a predicate specifying when the action is enabled, the postcondition specifies what that action does ($x' = x - 1$ for instance expresses that the action decreases the value of x by 1), the ns flag has the value NS (necessarily Satifiable) if the action is guaranteed to always succeed, and NNS if the action may fail, and the gpf, the Guaranted Property in case of Failure, is a predicate specifying what is however guaranteed to be true if the action fails.*

Definition 2.1 (Goal Decomposition Tree (GDT)) *A Goal Decomposition Tree describes the behaviour of the agents of a given type. Each*

node of this tree is a GDT goal. The tree structure is defined thanks to the decomposition of each GDT goal into subgoals.

Definition 2.2 (GDT goal) *A GDT goal g is described by a name ($name_g$), a satisfaction condition (sc_g), a gpf (gpf_g), a decomposition, an ns flag (ns_g) and a laziness flag (l_g). The satisfaction condition is a predicate specifying what the goal must establish, the gpf is a predicate specifying what is guaranted to be established if the execution of the goal fails, the ns flag specifies whether the goal always succeed or not, and the laziness flag specifies whether the goal decomposition is executed when the satisfaction condition of the goal is already true when the goal is considered.*

Definition 2.3 (Goal decomposition) *A GDT goal is either a leaf goal or an intermediate goal. In the latter case, the goal is decomposed into one or several subgoals, thanks to a decomposition operator. A list of decomposition operators can be found in [MSSZ07].*

Among others, we can informally introduce the following decomposition operators:

– **SeqOr**: Sequential Or. It decomposes the parent goal into several subgoals N_i. Subgoals are considered from the the left to the right. If the considered subgoal succeeds, the parent goal is achieved and the execution of the decomposition is ended. But if it fails, the next subgoal is considered. If the last subgoal is reached and fails, the satisfaction condition of the parent goal must be evaluated to know if it is achieved or not.

– **SeqAnd**: Sequential And. It decomposes the parent goal into several subgoals N_i. Subgoals are considered from the left to the right. If the considered subgoal succeeds, the next one is considered. If the last subgoal is considered and succeeds, the parent goal is achieved. But if it fails, the satisfaction condition of the parent goal must be evaluated to determine whether the parent goal is achieved or not.

– **SyncSeqOr** and **SyncSeqAnd**: These operators are similar to the SeqOr and SeqAnd operator, but environment variables can be locked during the whole execution of the parent goal decomposition.

2.4. *Properties proven by the method*

The GDT4MAS method allows to prove several kinds of properties. We first prove invariant and liveness properties, at the agent-type level and at the system-level. We recall here that invariant properties are properties that must be always true, and that liveness properties are properties that must eventually be true. Moreover, the proof-system of the method verifies that goal decompositions are valid. In this article, we focus on the proof of decompositions and of invariant properties. This is the topic of the next section.

3. Proving a GDT4MAS Model

In this section, we do not deal with the verification of the MAS; we only briefly describe how the correctness of an agent type is established. In order to make proofs compositional, context propagation rules and gpf propagation rules are associated to each decomposition operator. Moreover, a *Proof Schema* is associated to each operator. It generates proof obligations whose verification proves the correctness of the decomposition. We recall here such rules and schemas. For more details on the MAS proof or on the other proof schemas, the reader may refer to [MSSZ07, MS09].

In proof schemas, we use several predicat transformers that are presented in the next paragraph.

3.1. *Predicate transformers*

Notation 3.1 (At) *Let f a predicate. $f[i]$ is a predicate where each non-subscripted variable in f is subscripted by i.*

Example: $(x = y_0)[1] \equiv (x_1 = y_0)$.

Notation 3.2 (Between) *Let f a predicate. $f^{i \to j}$ is a predicate derived from f where each unprimed and unsubscripted variable is subscripted by i and each primed variable becomes unprimed and subscripted by j.*

Example: $(y' < x \wedge x' = x_0)^{1 \to 2} \equiv (y_2 < x_1 \wedge x_2 = x_0)$.

Notation 3.3 (Temporal switch) *Let f a predicate. $f^{\rightarrow i}$ is predicate derived from f where each subscript is increased by i.*

Example: $(x = x_1 \wedge y_2 = x_1)^{\rightarrow -2} \equiv (x = x_{-1} \wedge y_0 = x_{-1})$.

Notation 3.4 (Priming) *Let f a predicate. If f contains at least one primed variable, then $pr(f) = f$. Otherwise, $pr(f)$ is the predicate derived from f where each unsubscripted variable is primed.*

Examples: $pr((x = x_0)) \equiv (x' = x_0)$ *and* $pr((x = x')) \equiv (x = x')$.

Notation 3.5 (Stability) *Let t and agent type with two internal variables via, vib and one surface variable vs (internal and surface variables are described in the next section). Then, when one agent a of this type is considered $stab^{i \rightarrow j}$ is the predicate $via_i = via_j \wedge vib_i = vib_j \wedge vs_i = vs_j$.*

Notation 3.6 (Untemporalization) *Let f a predicate. $f^{\divideontimes}$ is the formula f in which all subscripts of value x are removed.*

Example: $(x_1 = x_2)^{\divideontimes} \equiv x = x_2$.

Notation 3.7 (Invariant) *Let A an agent situated in an environment $\mathcal{E}$. We write:*

– i_A the invariant associated to the internal variables of the agent;

– $i_\mathcal{E}$ the invariant associated to the environment variables;

– $i_{\mathcal{E}A}$ the conjunction of i_A and $i_\mathcal{E}$.

3.2. *Context inference*

In order to make proof obligations compositionnal, each proof schema has a *context* as hypothesis. This context can be calculated automatically in a top-down manner thanks to *context inference rules*.

A context is associated to a goal in a GDT. This is a first-order formula that is guaranted to be true when the goal is considered. In a context formula, it may be necessary to refer to the value of a variable

in a previous state. In that case, the variable is subscripted by a negative integer. The value in the current state is represented by the variable name neither subscripted nor primed. For instance, consider the GDT presented in figure 1, and suppose the context of the root goal is $x = y$.

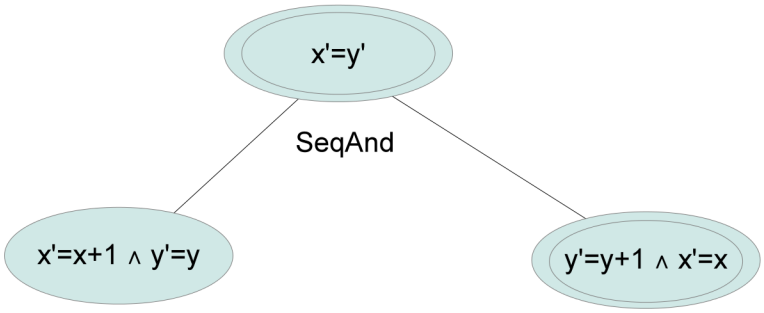

Figure 1: Simple GDT

The different states to consider are the following, as shown on figure 2.

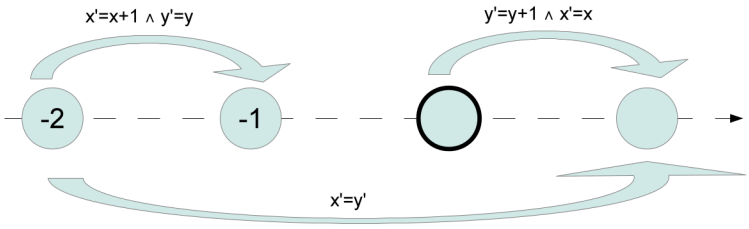

Figure 2: Simple GDT

In the state in which the right subgoal is considered (with a bold outline in figure 2), we know that:

– in state -2, from the parent goal, x and y are equals; So: $x_{-2} = y_{-2}$;

– between states -2 and -1, the value of x is increased, whereare the value of y is preserved, and so : $x_{-1} = x_{-2} + 1 \wedge y_{-1} = y_{-2}$;

– between state -1 and the state in which the right subgoal is considered, if x and y are internal variables (and thus, cannot be modified by other agents), we have: $x = x_{-1} \wedge y = y_{-1}$.

So, the context of the right subgoal is:

$$ctx = (x_{-2} = y_{-2}) \wedge (x_{-1} = x_{-2} + 1 \wedge y_{-1} = y_{-2}) \wedge (x = x_{-1} \wedge y = y_{-1})$$

Please notice that this allows to deduce that in the state in which the right subgoal is considered, we have $x = y + 1$.

For instance, the context propagation schema for the SyncSeqAnd operator, when the parent goal is NL, is the following:

$$\begin{cases} C_{N_1} = C_N \\ C_{N_{i(i>1)}} = \left(\left(\left(C_{N_{i-1}} \wedge sc_{N_{i-1}} \right)^{0 \to 1} \wedge stab^{1 \to 2} \wedge i_{\mathcal{E}A}[1] \wedge i_{\mathcal{E}A}[2] \right)^{\to -2} \right)^{\mathbb{N}} \end{cases}$$

And the context propagation schema for the SyncSeqOr operator when the parent goal is lazy is:

$$\begin{cases} C_{N_1} = \left(\left(C_N[0] \wedge stab^{0 \to 1} \right)^{\to -1} \right)^{\mathbb{N}} \\ C_{N_{i(i>1)}} = \left(\left(\left(\left(C_{N_{i-1}} \wedge gpf_{N_{i-1}} \right)^{0 \to 1} \wedge stab^{1 \to 2} \wedge i_{\mathcal{E}A}[1] \wedge i_{\mathcal{E}A}[2] \right)^{-2} \right)^{\mathbb{N}} \right)^{\mathbb{N}} \end{cases}$$

3.3. *GPF inference*

As explained above, each NNS goal is specified by a *Goal Property in case of Failure*. However, we have to prove that this property is verified when the goal decomposition fails. Thus, for each decomposition, we also have to infer a GPF from the decomposition, and we must establish that this inferred property implies the given GPF.

3.4. *Proof schemas*

A proof schema is associated to each decomposition operator. For instance, here is the proof schema associated to the SyncSeqOr operator

requires to prove both following formulae (where the value of l is 1 if the parent goal is lazy and 0 otherwise):

$$\bigwedge_{i=1}^{i=k} \left(\left(C_{N_i}[0] \wedge pr(sc_{N_i})^{0 \to 1} \wedge i_{\mathcal{E}A}[1] \right) \to pr(sc_N)^{(-2(i-1)-l) \to 1} \right) \tag{1}$$

Informally, it specifies that when a subgoal is executed in its context (which specifies that the eventual preceeding subgoal has failed) and succeeds, the parent goal is established.

Moreover, a proof schema is also associated to each NNS goal to establish that the given GPF is implied by the inferred GPF:

$$infgpf_N \to gpf_N \tag{2}$$

Finally a proof schema is also associated to each action node. It must establish that

– the precondition of the action is implied by the context of the node;

– the postcondition of the action implies the satisfaction condition of the node.

This proof schema is so the following (if α is the action associated to node N and $TY_{\mathcal{E}A}$ represents the typing of the agent and environment variables):

$$\begin{cases} C_N \to pre_\alpha \\ C_N[0] \wedge pr(post_\alpha)^{0 \to 1} \wedge TY_{\mathcal{E}A}[1] \to pr(sc_N)^{0 \to 1} \wedge i_{\mathcal{E}A}[1] \end{cases} \tag{3}$$

4. Application

4.1. *Introduction*

The main goal of this section is to illustrate the differents aspects of the model. So, a very simple but complete example of a GDT agent is detailed. The context of this example is the following: a robot R must turn on the light in several rooms, each room being identified by a

number. The behaviour specified by the GDT given here describes how the robot can proceed for a given room, numbered n. This GDT can be seen as a part of the whole GDT, managing the iteration on the set of rooms. Each room has at least one door with an electric eye which can turn on the light, and a traditional switch. As a consequence, the robot has two possibilities: coming into the room by the right door or using the switch. Moreover, it is supposed that the electric eye can be out of order, or does not always work as expected. These two contexts prevent the electric eye from turning on the light in the room. The given GDT does not explicitely specify how the robot reaches the considered room. We only consider the part of the behaviour where the robot is ready to enter into the required room. It is only supposed that the robot is able to move from a room to another and always stops inside a room. If the moving part of the behaviour has to be proven, the GDT we give here would be embedded into a bigger one.

4.2. *Specification*

First of all, here are the different constants, variables and types to be used in this context:

– **Types**

- the set of rooms numbers NUM : $NUM \subset \mathbb{N}$;

– **Environment variables**

- the set of room states S and the state of each door: $(S \in [NUM \rightarrow \{true, false\}]) \wedge (door \in [NUM \rightarrow \{true, false\}])$. The previous formula defining S and $door$ is called TY_E. If $S(n) = true$ with $n \in NUM$, it means that the room number n is lighted, and if $room(n)$ is true, it means that the door of room n is opened.

– **Internal variables** (that is to say, variables managed by the robot)

- the variable $inRoom$ indicates the room in which the robot is : $(inRoom \in NUM)$. The previous formula is called TY_R.

- the variable n indicates the room the Robot must lighten : $n \in NUM$.

The invariants associated to the environment(I_E) and to the robot(I_R) are equals to $true$.

Three clauses are associated to a GDT which are now defined for the GDT of the R robot:

– the *triggering context* TC which defines the conditions to be verified in order the GDT to be executed by the agent. Here, $TC = true$;

– the *precondition* $precGDT$ which defines additional conditions which must be verified when the triggering context is true for the GDT to be executed by the agent. Here, $precGDT = true$;

– the *initialisation clause* which defines how the different variables must be initialised at the begining of the GDT execution. Here the variable $inRoom$ can be initialised with any value of NUM (ie. the number of the room where the robot is at the begining of the behaviour). The variables S and $door$ are initialised according to the state of the different rooms considered. Please notice that our proof system specifies that the initialisation clause must logically imply I_R, which is always true here whatever the initialisation clause is.

Figure 3 shows the GDT of the robot agent. The main goal (LightedRoom - LR) consists in ligthening the chosen room. This goal is decomposed into two subgoals, thanks to the $SyncSeqOr$ operator. This specifies that, in order to lighten room n, the robot may first use the cellular eye (goal UsingCellularEye, UCE) by entering into the room. However, as explained above, this goal may fail. In that case, the robot will use the switch (goal UsingSwitch, US), a goal that always succeeds. The goal UCE is itself decomposed into two subgoals, thanks to the SyncSeqAnd operator: the first subgoal consists in opening the door, and the second one consists in entering into the room and switching the light on thanks to the cellular eye.

Please notice that the *SyncSeqOr* operator used in the GDT allows to lock variables during the execution of the parent goal. Here, we guarantee that the state of the considered room is not modified by another agent during this part of the robot's behaviour. We give now more details on the five goals of this GDT.

– **Goal LightedRoom (LR)**

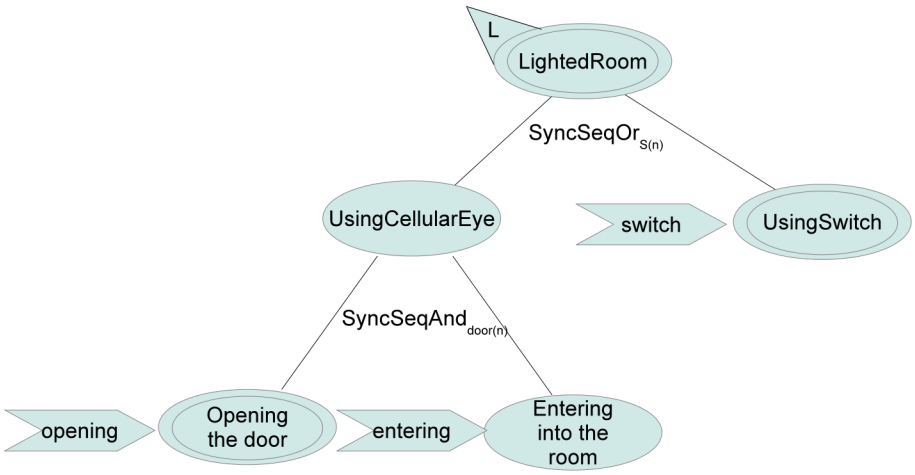

Figure 3: The GDT of the case study

- This goal is necessarily satisfiable (NS) which means that when the agent executes its GDT, it always achieves this goal.

- This goal is lazy which means that when the robot tries to execute the GDT, if the room to be lighted is already lighted, the goal is not executed.

- The satisfaction condition is: $SC_{LR} \equiv (S(n) = true)$

– **Goal UsingCellularEye (UCE)**

- this goal is decomposed into two subgoals thanks to the SynSeqAnd operator

- This goal is not necessarily satisfiable (NNS) which means that the robot does not always achieve to turn on the light when trying to use the cellular eye (see the beginning of this section)

- This goal is not lazy (NL) which means that the robot always tries to execute this goal.

- The satisfaction condition is the following : $SC_{UCE} \equiv (S'(n) = true \wedge inRoom' = n \wedge n' = n)$. It means that if the goal is achieved then the chosen room is lighted and the robot is in the room.

- The GPF of this goal is the following: $\neg S'(n) \wedge inRoom' = n \wedge n' = n$: the robot is still in the considered room (which remains unchanged) but the light of this room is off.

– **Goal UsingSwitch (US)**

- This goal is an elementary one which means that an action (described in the following) is associated to it.

- This goal is necessarily satisfiable (NS) which means that the robot always achieves to turn on the light using the switch.

- This goal is not lazy (NL).

- The satisfaction condition is the following : $SC_{US} \equiv (S(n) = false \rightarrow S'(n) = true) \wedge n' = n$. It means that if the goal is achieved then if the room was not lighted, it becomes lighted.

- The action associated to this goal is the *switch* action. Its precondition is $PRE_{switch} \equiv (inRoom = n)$ (the action can be executed only if the robot is inside the room, its postcondition is $POST_{switch} \equiv (S'(n) = \neg S(n) \wedge n' = n)$ (this means that using a switch consists in changing the state of the light in the room - whatever the state of the switch is) and its guaranteed property in case of failure is $GPF_{US} = GPF_{switch} = false$ (it is the default GPF when an action always succeeds).

– **Goal Opening the door (OD)**

- This goal is an elementary one;

- This goal is an NS goal, which means that the robot always succeed in opening the door;

- This goal is non-lazy;

- The satisfaction condition of this goal is $SC_{OD} \equiv (door'(n) \wedge n' = n)$: after the goal execution, the door is opened;

- The action associated to this goal is the *opening* action. Its precondition is $PRE_{opening} \equiv (true))$ and its postcondition is $POST_{opening} \equiv (door'(n) \wedge n' = n)$

– **Goal Entering into the room (ER)**

- This goal is an elementary one;

- This goal is an NNS goal. Indeed, wherease the robot always succeed in entering into the room, the whole action does not always succeed because it does not always turn the light on;

- This goal is non-lazy;

- The satisfaction condition of this goal is $SC_{ER} \equiv (inRoom' = n \wedge n' = n \wedge S'(n))$;

- The guaranted property in case of failure of this goal is $GPF_{ER} \equiv (inRoom' = n \wedge n' = n \wedge \neg S'(n))$;

- the action associated to this goal is the *entering* action. Its precondition is $PRE_{entering} \equiv door(n)$ (to enter in a room, its door must be opened), its postcondition is $POST_{entering} \equiv (inRoom' = n \wedge n = n' \wedge S'(n))$ and its gpf is $GPF_{entering} \equiv (inRoom' = n \wedge n = n' \wedge \neg S'(n))$.

4.3. *Proof obligations*

In the sequel, we generate predicates that must be proven to guarantee the correctness of the specification. Please notice that these predicates, called *Proof Obligations*, are calculated using *Proof Schemas* described above.

4.3.1. *Context inference*

Using context propagation rules established by the GDT model, we obtain the following context for the five nodes of our GDT:

$$C_{LR} = S \in \{NUM \rightarrow \mathbb{B}\} \wedge door \in \{NUM \rightarrow \mathbb{B}\} \wedge inRoom \in NUM \wedge n \in NUM$$

$$C_{UCE} = \begin{cases} S_{-1} \in \{NUM \rightarrow \mathbb{B}\} \wedge door_{-1} \in \{NUM \rightarrow \mathbb{B}\} \\ inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\ n_{-1} = n \wedge inRoom_{-1} = inRoom \wedge S_{-1}(n_{-1}) = S(n_{-1}) \\ S \in \{NUM \rightarrow \mathbb{B}\} \wedge door \in \{NUM \rightarrow \mathbb{B}\} \\ inRoom \in NUM \wedge n \in NUM \\ \neg S_{-1}(n_{-1}) \end{cases}$$

$$C_{US} = \begin{cases} S_{-3} \in \{NUM \to \mathbb{B}\} \wedge door_{-3} \in \{NUM \to \mathbb{B}\} \\ inRoom_{-3} \in NUM \wedge n_{-3} \in NUM \\ n_{-3} = n_{-2} \wedge inRoom_{-3} = inRoom_{-2} \wedge S_{-3}(n_{-3}) = S_{-2}(n_{-3}) \\ S_{-2} \in \{NUM \to \mathbb{B}\} \wedge door_{-2} \in \{NUM \to \mathbb{B}\} \\ inRoom_{-2} \in NUM \wedge n_{-2} \in NUM \\ \neg S_{-3}(n_{-3}) \\ \neg S_{-1}(n_{-2}) \wedge inRoom_{-1} = n_{-2} \wedge n_{-1} = n_{-2} \\ inRoom = inRoom_{-1} \wedge S(n_{-1}) = S_{-1}(n_{-1}) \wedge n = n_{-1} \\ S_{-1} \in \{NUM \to \mathbb{B}\} \wedge door_{-1} \in \{NUM \to \mathbb{B}\} \\ inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\ S \in \{NUM \to \mathbb{B}\} \wedge door \in \{NUM \to \mathbb{B}\} \\ inRoom \in NUM \wedge n \in NUM \end{cases}$$

$$C_{OD} = \begin{cases} S_{-1} \in \{NUM \to \mathbb{B}\} \wedge door_{-1} \in \{NUM \to \mathbb{B}\} \\ inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\ n_{-1} = n \wedge inRoom_{-1} = inRoom \wedge S_{-1}(n_{-1}) = S(n_{-1}) \\ S \in \{NUM \to \mathbb{B}\} \wedge door \in \{NUM \to \mathbb{B}\} \\ inRoom \in NUM \wedge n \in NUM \\ \neg S_{-1}(n_{-1}) \end{cases}$$

$$C_{ER} = \begin{cases} S_{-3} \in \{NUM \to \mathbb{B}\} \wedge door_{-3} \in \{NUM \to \mathbb{B}\} \\ inRoom_{-3} \in NUM \wedge n_{-3} \in NUM \\ n_{-3} = n_{-2} \wedge inRoom_{-3} = inRoom_{-2} \wedge S_{-3}(n_{-3}) = S_{-2}(n_{-3}) \\ S_{-2} \in \{NUM \to \mathbb{B}\} \wedge door_{-2} \in \{NUM \to \mathbb{B}\} \\ inRoom_{-2} \in NUM \wedge n_{-2} \in NUM \\ \neg S_{-3}(n_{-3}) \\ door_{-1}(n_{-2}) \wedge n_{-1} = n_{-2} \\ inRoom_{-1} = inRoom \wedge n_{-1} = n \wedge door_{-1}(n_{-1}) = door(n) \\ door_{-1} \in NUM \to \mathbb{B} \wedge S_{-1} \in NUM \to \mathbb{B} \\ inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\ door \in NUM \to \mathbb{B} \wedge S \in NUM \to \mathbb{B} \\ inRoom \in NUM \wedge n \in NUM \end{cases}$$

4.3.2. *gpf inference*

Using the gpf inference rule of the method, we have:

$$
infgpf_{UCE} = \left\{
\begin{array}{l}
S_{-5} \in \{NUM \to \mathbb{B}\} \wedge door_{-5} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-5} \in NUM \wedge n_{-5} \in NUM \\
n_{-5} = n \wedge inRoom_{-5} = inRoom \wedge S_{-5}(n_{-5}) = S(n_{-5}) \\
S \in \{NUM \to \mathbb{B}\} \wedge door \in \{NUM \to \mathbb{B}\} \\
inRoom \in NUM \wedge n \in NUM \\
\neg S_{-5}(n_{-5}) \\
door_{-3}(n) \wedge n_{-3} = n \\
inRoom_{-3} = inRoom_{-2} \wedge n_{-3} = n_{-2} \wedge door_{-3}(n_{-3}) = door_{-2}(n_{-2}) \\
door_{-3} \in NUM \to \mathbb{B} \wedge S_{-3} \in NUM \to \mathbb{B} \\
inRoom_{-3} \in NUM \wedge n_{-3} \in NUM \\
door_{-2} \in NUM \to \mathbb{B} \wedge S_{-2} \in NUM \to \mathbb{B} \\
inRoom_{-2} \in NUM \wedge n_{-2} \in NUM \\
(inRoom_{-1} = n_{-2} \wedge n_{-2} = n_{-1} \wedge \neg S_{-1}(n_{-2})) \\
(inRoom' = inRoom_{-1} \wedge n' = n_{-1}) \\
door_{-1} \in NUM \to \mathbb{B} \wedge S_{-1} \in NUM \to \mathbb{B} \\
inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\
door' \in NUM \to \mathbb{B} \wedge S' \in NUM \to \mathbb{B} \\
inRoom' \in NUM \wedge n' \in NUM
\end{array}
\right.
$$

4.3.3. *Proofs*

4.3.3.1. SyncSeqOr decomposition

From proof schema (1), we have two formulae to check.

The first formula is equivalent to:

$$
\begin{array}{l}
\left\{
\begin{array}{l}
S_{-1} \in \{NUM \to \mathbb{B}\} \wedge door_{-1} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\
n_{-1} = n_0 \wedge inRoom_{-1} = inRoom_0 \wedge S_{-1}(n_{-1}) = S_0(n_{-1}) \\
S_0 \in \{NUM \to \mathbb{B}\} \wedge door_0 \in \{NUM \to \mathbb{B}\} \\
inRoom_O \in NUM \wedge n_0 \in NUM \\
\neg S_{-1}(n_{-1}) \\
S_1(n_0) \wedge inRoom_1 = n_0 \wedge n_1 = n_0 \\
S_1 \in \{NUM \to \mathbb{B}\} \wedge door_1 \in \{NUM \to \mathbb{B}\} \\
inRoom_1 \in NUM \wedge n_1 \in NUM
\end{array}
\right. \\
\to \\
S_1(n_1)
\end{array}
$$

And the second one is equivalent to:

$$
\begin{array}{l}
\left\{\begin{array}{l}
S_{-3} \in \{NUM \to \mathbb{B}\} \wedge door_{-3} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-3} \in NUM \wedge n_{-3} \in NUM \\
n_{-3} = n_{-2} \wedge inRoom_{-3} = inRoom_{-2} \wedge S_{-3}(n_{-3}) = S_{-2}(n_{-3}) \\
S_{-2} \in \{NUM \to \mathbb{B}\} \wedge door_{-2} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-2} \in NUM \wedge n_{-2} \in NUM \\
\neg S_{-3}(n_{-3}) \\
\neg S_{-1}(n_{-2}) \wedge inRoom_{-1} = n_{-2} \wedge n_{-1} = n_{-2} \\
inRoom_0 = inRoom_{-1} \wedge S_0(n_{-1}) = S_{-1}(n_{-1}) \wedge n_0 = n_{-1} \\
S_{-1} \in \{NUM \to \mathbb{B}\} \wedge door_{-1} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\
S_0 \in \{NUM \to \mathbb{B}\} \wedge door_0 \in \{NUM \to \mathbb{B}\} \\
inRoom_0 \in NUM \wedge n_0 \in NUM \\
\neg S_0(n_0) \to S_1(n_0) \\
n_1 = n_0 \\
S_1 \in \{NUM \to \mathbb{B}\} \wedge door_1 \in \{NUM \to \mathbb{B}\} \\
inRoom_1 \in NUM \wedge n_1 \in NUM
\end{array}\right. \\
\to \\
S_1(n_1)
\end{array}
$$

Please notice that we also have a proof obligation generated by the SyncSeqAnd operator decomposing the goal UCE, but we do not present it here. Notice also that this decomposition does not impact the proof obligations associated to the SyncSeqOr operator.

4.3.3.2. Elementary goal US

The first proof obligation generated by the proof schema associated to elementary goals consists in verifying that the precondition of the $switch$ action is true when this action has to be executed :

$$
\begin{array}{l}
\left\{\begin{array}{l}
S_{-3} \in \{NUM \to \mathbb{B}\} \wedge door_{-3} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-3} \in NUM \wedge n_{-3} \in NUM \\
n_{-3} = n_{-2} \wedge inRoom_{-3} = inRoom_{-2} \wedge S_{-3}(n_{-3}) = S_{-2}(n_{-3}) \\
S_{-2} \in \{NUM \to \mathbb{B}\} \wedge door_{-2} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-2} \in NUM \wedge n_{-2} \in NUM \\
\neg S_{-3}(n_{-3}) \\
\neg S_{-1}(n_{-2}) \wedge inRoom_{-1} = n_{-2} \wedge n_{-1} = n_{-2} \\
inRoom = inRoom_{-1} \wedge S(n_{-1}) = S_{-1}(n_{-1}) \wedge n = n_{-1} \\
S_{-1} \in \{NUM \to \mathbb{B}\} \wedge door_{-1} \in \{NUM \to \mathbb{B}\} \\
inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\
S \in \{NUM \to \mathbb{B}\} \wedge door \in \{NUM \to \mathbb{B}\} \\
inRoom \in NUM \wedge n \in NUM
\end{array}\right. \\
\to \\
inRoom = n
\end{array}
$$

The second proof obligation consists in verifying that SC_{US} is satisfied when the action has been executed with success and that the invariant is preserved:

$$
\begin{array}{l}
\left\{\begin{array}{l}
S_{-3} \in \{NUM \rightarrow \mathbb{B}\} \wedge door_{-3} \in \{NUM \rightarrow \mathbb{B}\} \\
inRoom_{-3} \in NUM \wedge n_{-3} \in NUM \\
n_{-3} = n_{-2} \wedge inRoom_{-3} = inRoom_{-2} \wedge S_{-3}(n_{-3}) = S_{-2}(n_{-3}) \\
S_{-2} \in \{NUM \rightarrow \mathbb{B}\} \wedge door_{-2} \in \{NUM \rightarrow \mathbb{B}\} \\
inRoom_{-2} \in NUM \wedge n_{-2} \in NUM \\
\neg S_{-3}(n_{-3}) \\
\neg S_{-1}(n_{-2}) \wedge inRoom_{-1} = n_{-2} \wedge n_{-1} = n_{-2} \\
inRoom_0 = inRoom_{-1} \wedge S_0(n_{-1}) = S_{-1}(n_{-1}) \wedge n_0 = n_{-1} \\
S_{-1} \in \{NUM \rightarrow \mathbb{B}\} \wedge door_{-1} \in \{NUM \rightarrow \mathbb{B}\} \\
inRoom_{-1} \in NUM \wedge n_{-1} \in NUM \\
S_0 \in \{NUM \rightarrow \mathbb{B}\} \wedge door_0 \in \{NUM \rightarrow \mathbb{B}\} \\
inRoom_0 \in NUM \wedge n_0 \in NUM \\
S_1(n_0) = \neg S_0(n_0) \wedge n_1 = n_0 \\
S_1 \in \{NUM \rightarrow \mathbb{B}\} \wedge door_1 \in \{NUM \rightarrow \mathbb{B}\} \\
inRoom_1 \in NUM \wedge n_1 \in NUM
\end{array}\right. \\
\rightarrow \\
((\neg S_0(n_0) \rightarrow S_1(n_0)) \wedge n_1 = n_0 \wedge true)
\end{array}
$$

4.3.3.3. Other proof obligations

Six other proof obligations are engendered by the method but not detailed here. Here is a list of them:

– The decomposition of the UCE goal with the SyncSeqAnd operator is valid: 1 PO;

– The actions associated to elementary goals *Opening the door* and *entering into the room* are correct: 4 PO;

– The gpf associated to the UCE goal is correct: 1 PO.

5. Proof with PVS

5.1. *Specification*

The case study presented in the previous section has been proven with PVS [SRI]. PVS is a verification system that integrates a powerful theorem prover and a model checker (that has not been used here).

Proofs are performed on specifications based on an expressive language that augments classical higher-order logic with, especially, a type system.

Figures 4 to 6 show the specification obtained for the case study. It is very close to the set of proof obligations generated by our proof system (which is, indeed, one of the aims of this system). Each proof obligation is specified as a named theorem, that is to say a formula that has to be proven by PVS. Let notice that trivial proof obligations (proof obligations in which we have to prove $true$) have not been added to the PVS specification.

The first part of the specification is dedicated to types, variables or constants declarations. When an identifier is declared as a variable (with the VAR keyword), it implies that the identifier will be universally quantified in each formula of the theory. This explains that no explicit universal quantification appears in theorems. Moreover, these declarations are automatically added as typing hypotheses in theorems to be proved.

So, the number of rooms max is declared as a constant (with no explicit associated value) and other identifiers are declared as variables. A parametrized type $roomSet$ is used to specify the set of existing room numbers in the following declarations. As a consequence, $roomSet(max)$ corresponds to the set NUM in the previous specification.

```
lumieresEtendu: THEORY
 BEGIN

  max: nat

  roomSet(l: nat): TYPE = {n: nat | n ≤ l}

  np, n, n_5, n_3, n_2, n_1, n_0, n_1: VAR roomSet(max)

  Sp, S, S_5, S_3, S_2, S_1, S_0, S_1: VAR [roomSet(max) → bool]

  doorp, door, door1, door0, door_1, door_2, door_3, door_5: VAR
        [roomSet(max) → bool]

  inRoomp, inRoom, inRoom_5, inRoom_3, inRoom_2, inRoom_1, inRoom0, inRoom1: VAR
        roomSet(max)

  PO1: THEOREM
    (¬ S_3(n_3)) &
     (S_3(n_3) = S_2(n_3)) &
      (inRoom_3 = inRoom_2) &
       (n_3 = n_2) &
        (door_1(n_2)) &
         (n_1 = n_2) &
          (inRoom0 = inRoom_1) &
           (n_0 = n_1) &
            (door_1(n_1) = door0(n_0)) & (inRoom1 = n_0) & (n_1 = n_0) & (S_1(n_0))
    ⇒ S_1(n_2) & (inRoom1 = n_2) & (n_1 = n_2)

  PO2: THEOREM
    (¬ S_1(n_1)) &
     (S_1(n_1) = S_0(n_1)) &
      (inRoom_1 = inRoom0) & (n_1 = n_0) & S_1(n_0) & (inRoom1 = n_0) & (n_1 = n_0)
     ⇒ S_1(n_1)

  PO3: THEOREM
    (¬ S_3(n_3)) &
     (S_3(n_3) = S_2(n_3)) &
      (inRoom_3 = inRoom_2) &
       (n_3 = n_2) &
        (¬ S_1(n_2)) &
         (inRoom_1 = n_2) &
          (n_1 = n_2) &
           (inRoom0 = inRoom_1) &
            (S_0(n_1) = S_1(n_1)) &
             (n_0 = n_1) & ((¬ S_0(n_0)) ⇒ S_1(n_0)) & (n_1 = n_0)
    ⇒ S_1(n_1)

...
```

Figure 4: PVS specification of the case study (1)

```
...

  PO4: THEOREM
    (¬ S_5(n_5)) &
     (S_5(n_5) = S(n_5)) &
      (inRoom_5 = inRoom) &
       (n_5 = n) &
        door_3(n) &
         (n_3 = n) &
          (inRoom_3 = inRoom_2) &
           (n_3 = n_2) &
            (inRoom_1 = n_2) &
             (n_2 = n_1) & (¬ S_1(n_2)) & (inRoomp = inRoom_1) & (np = n_1)
    ⇒
    ((Sp(n) & (inRoomp = n) & (np = n)) ∨
       ((¬ Sp(n)) & (inRoomp = n) & (np = n)))


  PO5: THEOREM
    (¬ S_1(n_1)) &
     (S_1(n_1) = S0(n_1)) &
      (inRoom_1 = inRoom0) & (n_1 = n0) & door1(n0) & (n1 = n0)
    ⇒ (door1(n0) & (n1 = n0))

  PO6: THEOREM
    (¬ S_3(n_3)) &
     (S_3(n_3) = S_2(n_3)) &
      (inRoom_3 = inRoom_2) &
       (n_3 = n_2) &
        door_1(n_2) &
         (n_1 = n_2) &
          (inRoom_1 = inRoom) & (n_1 = n) & (door_1(n_1) = door(n))
    ⇒ door(n)

...
```

Figure 5: PVS specification of the case study (2)

```
...

  PO7: THEOREM
    (¬ S_3(n_3)) &
     (S_3(n_3) = S_2(n_3)) &
      (inRoom_3 = inRoom_2) &
       (n_3 = n_2) &
        door_1(n_2) &
         (n_1 = n_2) &
          (inRoom_1 = inRoom0) &
           (n_1 = n_0) &
            (door_1(n_1) = door0(n_0)) & (inRoom1 = n_0) & (n_0 = n_1) & S_1(n_0)
     ⇒ (inRoom1 = n_0) & (n_1 = n_0) & S_1(n_0)

  PO8: THEOREM
    (¬ S_3(n_3)) &
     (S_3(n_3) = S_2(n_3)) &
      (inRoom_3 = inRoom_2) &
       (n_3 = n_2) &
        (¬ S_1(n_2)) &
         (inRoom_1 = n_2) &
          (n_1 = n_2) & (inRoom = inRoom_1) & (S(n_1) = S_1(n_1)) & (n = n_1)
     ⇒ (inRoom = n)

  PO9: THEOREM
    (¬ S_3(n_3)) &
     (S_3(n_3) = S_2(n_3)) &
      (inRoom_3 = inRoom_2) &
       (n_3 = n_2) &
        (¬ S_1(n_2)) &
         (inRoom_1 = n_2) &
          (n_1 = n_2) &
           (inRoom0 = inRoom_1) &
            (S_0(n_1) = S_1(n_1)) &
             (n_0 = n_1) & (S_1(n_0) = (¬ S_0(n_0))) & (n_1 = n_0)
     ⇒ (((¬ S_0(n_0)) ⇒ S_1(n_0)) & (n_1 = n_0))

END lumieresEtendu
```

Figure 6: PVS specification of the case study (3)

5.2. *Proof*

Each theorem of the PVS specification has been automatically proven by the PVS prover using the default strategy called $grind$. The proof is based on sequent calculus. The default strategy integrates different techniques such as definition expansion and arithmetic, equality or quantifier reasoning. We only give here the details of the proof of one PO (PO9). What is important to notice is that the proof is performed completely automatically.

The trace of the proof of $PO9$ is given in figure 7 through 10. The first step is obtained after a skolemization process. The second step is reached after a simplification process to flatten the sequent. In step 3, equalities between variables are used to unify terms. And finally, in step 4, the last sequent is simplified, making the proof feasible.

Verbose proof for PO9.
PO9:

$$
\begin{array}{ll}
\{\text{-}1\} & \text{inRoom0}' \leq \text{max} \\
\{\text{-}2\} & \text{inRoom}' \leq \text{max} \\
\{\text{-}3\} & \text{inRoom}'' \leq \text{max} \\
\{\text{-}4\} & \text{inRoom}''' \leq \text{max} \\
\{\text{-}5\} & n'_0 \leq \text{max} \\
\{\text{-}6\} & n'_1 \leq \text{max} \\
\{\text{-}7\} & n' \leq \text{max} \\
\{\text{-}8\} & n'' \leq \text{max} \\
\{\text{-}9\} & n''' \leq \text{max} \\
\hline
\{1\} & (\neg\ S'''(n''')) \ \& \\
& (S'''(n''') = S''(n''')) \ \& \\
& (\text{inRoom}''' = \text{inRoom}'') \ \& \\
& (n''' = n'') \ \& \\
& (\neg\ S'(n'')) \ \& \\
& (\text{inRoom}' = n'') \ \& \\
& (n' = n'') \ \& \\
& (\text{inRoom0}' = \text{inRoom}') \ \& \\
& (S'_0(n') = S'(n')) \ \& \\
& (n'_0 = n') \ \& \ (S'_1(n'_0) = (\neg\ S'_0(n'_0))) \ \& \ (n'_1 = n'_0) \\
& \Rightarrow (((\neg\ S'_0(n'_0)) \Rightarrow S'_1(n'_0)) \ \& \ (n'_1 = n'_0))
\end{array}
$$

Figure 7: PVS proof, step 1 (skolemization)

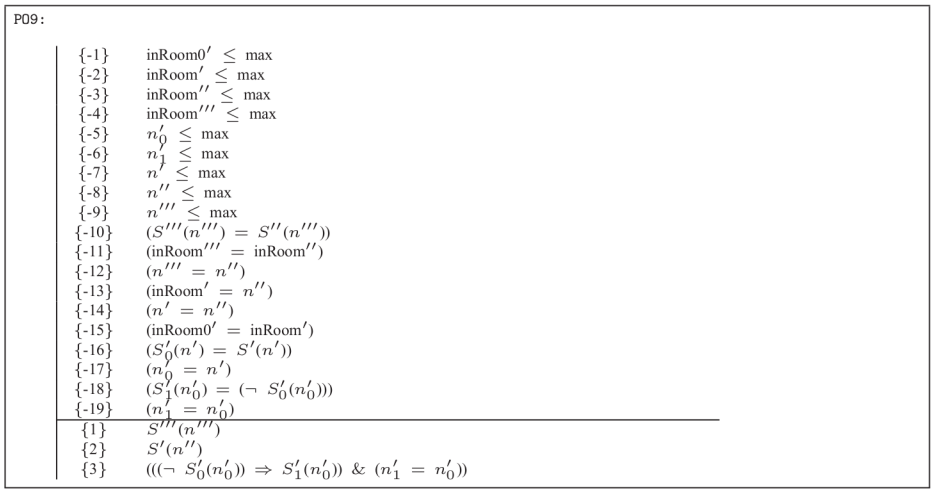

Figure 8: PVS proof, step 2 (flattening)

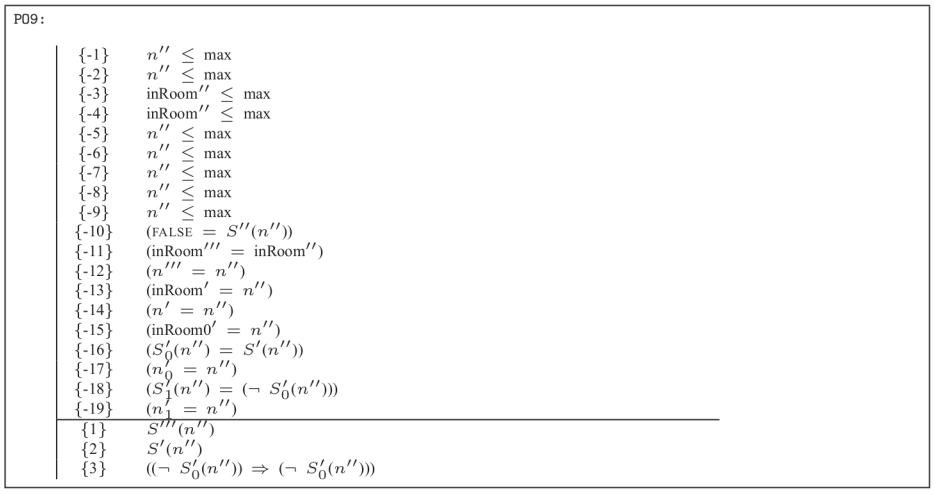

Figure 9: PVS proof, step 3 (replacement)

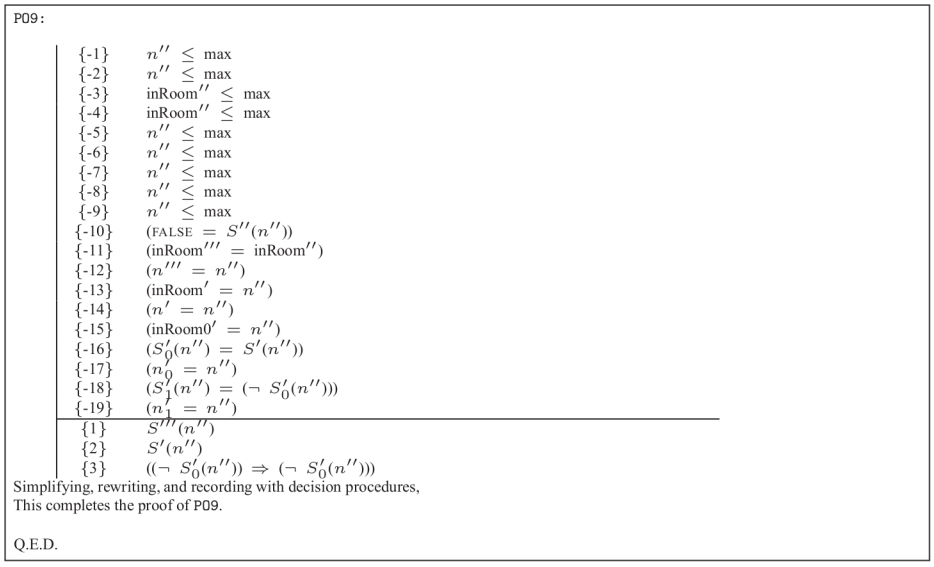

Figure 10: PVS proof, step 4 (simplification)

6. Conclusion

Presenting the whole GDT4MAS model and its application is, of course, not possible in such a presentation. For instance, communications have not been considered (but more can be found on this subject in [MS10, MS11]. However, we hope we managed to illustrate a few important characteristics of the model, among others:

– its compositional aspect;

– its expressiveness, also other capabilities using holons are not presented here (but can be found in [MS10]);

– its capability to be automated;

– its adequation to any theorem prover, thanks to proof schemas generating proof obligations in first order logic.

– the success of automatic proofs.

The case study we used in this article, is of course, a very simple example, with only one agent. We chose it however to present in details how proof obligations are produced, and how they are proven to be correct. But of course, the model has also been illustrated with several agents of several types. The essential specificity of this model making it more adequate than others to complex systems is that it uses theorem proving for system of several agents. Most other agent verification systems are either limited to one agent (like goal [dBHvdHM00] for instance) or use model checking, making the verification of system with a greate number of agents impossible, as it is presented in other works trying to use model-checking on complex systems [TMBDLK04, Bon10]. Moreover, the compositional aspect of the system, and the fact that proofs are associtated to agent types rather than to agents, makes the complexity of the proof process linear in the number of agent types.

References

[Abr96] J.-R. Abrial. *The B-Book*. Cambridge Univ. Press, 1996.

[ASM08] E. Adam, E. Grislin-Le Strugeon, and R. Mandiau. Flexible hierarchical organisation of role based agents. In *2nd IEEE Int. Conf. on Self-Adaptive and Self-Organizing Systems Workshops*, pages 186–191, 2008.

[BFVW03] R.H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In M. Dastani, J. Dix, and A. Seghrouchni, editors, *ProMAS*, 2003.

[Bon10] F. Bonnefoi. *Vérification formelle des spécifications de systèmes complexes par réseaux de Pétri : application aux systèmes de transport intelligents*. PhD thesis, Université Pierre et Marie Curie, 2010.

[CS] Clear-Sy. B for free. http://www.b4free.com.

[dBHvdHM00] F.S. de Boer, K.V. Hindriks, W. van der Hoek, and J.-J.Ch. Meyer. Agent programming with declarative goals. In *7th International Workshop on Intelligent Agents. Agent Theories Architectures and Language*, pages 228–243, 2000.

[Holcm] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997 .

[MS09] B. Mermet and G. Simon. GDT4MAS: an extension of the GDT model to specify and to verify MultiAgent Systems. In Decker *et al.*, editor, *Proc. of AAMAS 2009*, pages 505–512, 2009.

[MS10] B. Mermet and G. Simon. Specifying and verifying holonic agents with gdt4mas. *Int. Journal of Agent-Oriented Software Engineering*, 4(3):281–303, 2010.

[MS11] B. Mermet and G. Simon. Specifying recursive agents with gdts. *Autonomous agents and Multi-Agent Systems*, 23(2):273–301, 2011.

[MSSZ07] B. Mermet, G. Simon, A. Saval, and B. Zanuttini. Specifying, verifying and implementing a MAS: A

case study. In M. Dastani, A. E. F. Segrouchni, A. Ricci, and M. Winikoff, editors, *Post-Proc. of ProMAS'07*, number 4908 in Lecture Notes in Artificial Intelligence, pages 172–189. Springer, 2007.

[Occ00] M. Occello. Towards a generic recursive agent model. In *Int. Conf. on Artificial Intelligence*, pages 649–654, 2000.

[OFT04] *SystèmesMulti-Agents*, volume29 of*ARAGO*.OFTA, 2004.

[SRI] SRI International. PVS. http://pvs.csl.sri.com.

[TMBDLK04] Y. Thierry-Mieg, S. Baarir, A. Duret-Lutz, and F. Kordon. Nouvelles techniques de model-checking pour la vérification de systèmes complexes. *Revue Génie Logiciel*, 69:17–23, 2004.