Side effects of agents are not just Random

Bruno Mermet, Gaële Simon

Side effects of agents are not just random

Bruno Mermet and Gaële Simon

Greyc – CNRS UMR 6072
Bruno.Mermet@univ-lehavre.fr

Abstract. Side effects are an important characteristic of MAS, and proving them is an interesting issue. They often can be expressed as liveness properties. But there is no system dedicated to this kind of proof. The GDT4MAS framework allows to specify and prove the correctness of multiagent systems. This framework is mainly dedicated to prove safety properties about the system and to prove that agents achieve their goal(s). However, there is no proof principle to prove that agents satisfy liveness properties that are not part of their goal(s). In this article, we propose a proof mechanism that addresses this kind of problem: we show how we can add to GDT4MAS a proof mechanism adapted to prove leads-to properties, a subclass of liveness properties.

1 Introduction

During the execution of a MAS, unexpected system properties are often observed. These properties can either be useful (they can be for example called *emergent properties*) or harmful. In both cases, it may be interesting to prove that such properties will eventually happen in order to understand how they happen. However, to our knowledge, there is no system suitable to prove such properties.

Proving the correctness of multiagent systems is a hard problem that has been tackled for several years. Most of the works on the subject have established that a new formal specification and proof system, dedicated to multiagent systems, should be developed. Among them, GDT4MAS [1] proposes interesting characteristics. Especially, the proof obligation generation process is fully automatisable and it can be applied to very large systems, essentially because it relies on theorem proving and first-order logic rather than on model-checking and propositional logic.

However, if this system is well-suited to prove invariant properties (also called safety properties) and to guarantee that agents satisfy their main goal, it does not propose any proof system to guarantee that an agent establishes liveness properties that are not part of its main goal, which is necessary when considering side effects.

So, in this article, we propose to add a new proof system to GDT4MAS dedicated to the verification of a well-known kind of liveness properties: *leads-to* properties.

Of course, there are techniques to verify leads-to properties in distributed systems [2–4], but these techniques are dedicated to systems where all the processes are globally taken into account for each proof that must be performed.

Most of these techniques are dedicated to systems where processes work on independant variables, and synchronize occasionnaly to exchange information (This for instance the case of the π-calculus [4]).

On the contrary, in a method such as TLA$^+$ [3], shared variables can be specified, but the proof process requires to consider, for each step of the system trace, all the actions that may be considered. This is not suitable for multi-agents systems, because the number of possible actions is very large and thus, the property to prove would be to complicated to be performed by an automatic (or human) prover.

To perform efficient proofs on multi-agents systems, a compositional proof system is required. This is the case of the GDT4MAS proof system, and this is a property we require for a proof system dedicated to leads-to properties in multi-agents systems.

In the next section, we briefly introduce the notion of liveness property. In section 3, we recall the main concepts of the GDT4MAS framework. The new proof system we propose is described in section 4, and its application is examplified in section 5. Finally, a comparison with other works is proposed in section 6.

2 Invariant and liveness properties

When dealing with formal verification of software, many kinds of properties may be considered. In this section, we present two kinds of them: invariant properties and liveness properties.

2.1 Invariant properties

Invariant properties specify a set of states the system must satisfy at every moment. They are often presented as properties specifying that "nothing bad happens". Theses properties are mainly safety properties. Indeed, when specifying safety critical systems (as, for instance, a train control system), a first critial step is to verify that the system does not reach an unsafe state. In a formal verification system such as the B method, used by corporates developing safety critical systems, this kind of property is the only one that is formally proven.

Using temporal logic, an invariant property IP is specified as:

$$\Box(IP)$$

However, a system doing nothing may trivially verify such invariant properties. Indeed, these properties do not specify anything about the task of the system.

2.2 Liveness properties

Contrary to invariant properties, liveness properties specify how the system should modify its state. They are often presented as properties specifying that "something good will happen". There are many kinds of liveness properties. Here are a small presentation of some of them. More details can be found for instance in [2, 3].

- **One-day**: a given property OD will eventually become true:

$$\Diamond(OD)$$

- **Leads-to**: a given property LT will eventually become true every time another property P is true:

$$\Box(P \rightarrow \Diamond LT)$$

- **Until**: a given property UP remains true until another property becomes true (and P will eventually become true):

$$\Box(UP \rightarrow (\Box(UP \vee P) \wedge \Diamond P)$$

In the rest of this article, we will only consider *leads-to* properties. Indeed, a *one-day* property is a subtype of a *leads-to* property (with $P \triangleq true$ and $LT \triangleq OD$), and an *until* property is also a special kind of *leads-to* property.

3 The GDT4MAS framework and the GDT model

3.1 Main concepts

In the GDT4MAS framework, the MAS is described by an environment, mainly specified by variables, an invariant (denoted $i_{\mathcal{E}}$ in the sequel) 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 following, after a short description of the notations we use, the notions of agent type and of agent behaviour are described.

3.2 Notation

Notation 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 $v' = v + 1$.

3.3 Agent Type Specification

Simplified Definition 1 (Agent Type) *An agent type t is mainly described by a name ($name_t$), a set of variables ($VarI_t$), an invariant (i_A), and a behaviour (b_t) defined by a GDT.*

Definition 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 using decomposition operators. A predicate called Triggering Context (TC) is associated to each GDT: an agent begins the execution of its behaviour every time its TC is true.*

Simplified Definition 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 or an action and an ns flag (ns_g). The satisfaction condition is a predicate specifying the property the goal must establish when it succeeds, whereas the gpf (*Guaranted Property in case of Failure*) is a predicate specifying the property the goal must establish when its execution fails. The ns flag specifies whether the goal always succeeds (*Necessarily Satsfiable or NS*) or not (NNS).*

Please notice that when the execution of a node fails, the invariant must still remain true. The failure of a node represents the fact that, in a real world, an agent is not always guaranted to succeed in realizing a task dealing with the environment. For instance, a robot that must move its arm may be blocked by an object, or its arm may be rusty, reducing the amplitude of its move.

Simplified Definition 3 (Action) *An action α is specified by a name ($name_\alpha$), a precondition (pre_α), a postcondition ($post_\alpha$), an ns flag (ns_α) and a gpf (gpf_α). 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 if the action is guaranteed to always succeed, and NNS if the action may fail. The gpf is a predicate specifying what is however guaranteed to be true if the action fails.*

Definition 2 (Goal decomposition). *A GDT goal is either a leaf goal or an intermediate goal. An action is attached to a leaf goal, whereas an intermediate goal is decomposed into several subgoals, thanks to a decomposition operator. A list of decomposition operators can be found in [5].*

Among others, we can introduce the following decomposition operators:

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

– **SeqAnd**: Sequential And. It decomposes the parent goal N into several subgoals N_i. Subgoals are executed from the left to the right. If the considered subgoal succeeds, the next one is executed. If the last subgoal is executed and succeeds, N is achieved. But if a subgoal fails, the satisfaction condition of N must be evaluated to determine whether N is achieved or not.
– **SyncSeqOr** and **SyncSeqAnd**: These operators are similar to the SeqOr and SeqAnd operator, but a subset of environment variables can be locked during the whole execution of the parent goal decomposition.

An example of GDT is given in figure 1. In this figure, goals are described by their satisfaction condition. Moreover, NS goals are surrounded by a double ellipse. In satisfaction conditions, x and x' respectively represent the value of the variable x before and after the goal execution.

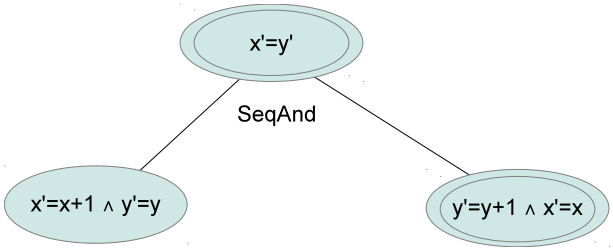

Fig. 1. Simple GDT

3.4 Proof system: general principles

The proof system fo GDT4MAS relies on *Proof Schemas* (PS). Applying a proof schema generates *Proof Obligations* (PO), that may be proven by an automatic prover, such as PVS [6]. At the moment, PS allow us to prove several kinds of properties. We first prove invariants at the agent-type level and at the system-level. Moreover, the proof system of the method verifies that goal decompositions are valid. Most PS rely on goal contexts. These contexts are computed automatically starting from the root goal. Intuitively, the context C_G of a goal G is a predicate summarizing the state in which goal G will be executed.

3.5 Notations

In this section, we present two notations of GDT4MAS that will be used in the sequel.

Notation 2 (Priming) *Let f be a predicate/expression. If f contains at least one primed variable, then $pr(f) = f$. Otherwise, $pr(f)$ is the predicate/expression 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 (Invariant) *Let A an agent situated in an environment $\mathcal{E}$. We write:*

- *i_A the invariant regarding variables of the agent;*
- *$i_\mathcal{E}$ the invariant associated to the environment variables;*
- *$i_{\mathcal{E}A}$ the conjunction of i_A and $i_\mathcal{E}$.*

4 A new proof mechanism dedicated to leads-to properties

This section presents the proof mechanism we propose to verify that some leads-to properties are established by an agent. Here, we only consider leads-to properties that are associated to an agent; no other agent is required to establish this property.

In this section, we consider a *leads-to* property L defined so:

$$L \equiv \Box(P_L \rightarrow \Diamond Q_L)$$

A classical way to establish that a *leads-to* property is verified by a specification consists in associating a *variant* and a *witness* to this property [2].

Informally, a variant expresses the progress towards the establishment of Q_L. If it is proven that an agent makes a variant decrease and that when this variant reaches its lower bound, Q_L is established, then the leads-to property L is proven. A witness is a property that represents the fact that P_L has been true, and thus, that Q_L must be established. In this article we propose to adapt this mechanism to verify leads-to properties of agents.

4.1 Definitions and notations

We begin by a formal definition of a variant:

Definition 3 (Variant). *A variant is a decreasing sequence defined in a well-founded structure.*

Of course, this definition requires to define what a well-founded stucture is.

Definition 4 (Well-founded Structure). *A well-founded structure $(S, <)$ is a set S with an order relation $<$ such that every decreasing sequence in S has a lower bound. For instance, $(\mathbb{N}, <)$ is a well-founded structure.*

Corollary 1. *A variant has a lower bound. We write V_{0_L} the lower bound of a variant V_L.*

In this article, we will only consider variants defined on $(\mathbb{N}, <)$. This property must be added to the invariant of the agent.

Definition 5 (Witness). *Let $L \equiv \Box(P_L \rightarrow \Diamond Q_L)$ a leads-to property. A witness is a property that must be true when P_L is true, and remains true until Q_L is true.*

Notation 4 (Variant and Witness) *Let L a leads-to property associated to an agent A. We write:*

- *V_L the variant we associate to L to prove it;*
- *V_{0_L} the lower bound of the variant V_L;*
- *W_L the witness we associate to L to prove it.*

4.2 Sketch of the proof process

Thanks to the variant and witness we associate to a leads-to property, proving that an agent establishes a leads-to property L consists in proving that:

1. The chosen variant is a variant:
   - when it has reached its lower bound, Q_L is established;
   - once P_L has been true and until Q_L becomes true, the agent must execute its gdt.
   - there is no other agent that increases the variant;
2. The chosen witness is a witness;
   - it is true when P_L is true;
   - when W_L is true, it remains true until Q_L becomes true.
3. The agent progresses: when the agent executes its gdt, it makes the variant decrease or it establishes Q_L.

In the next parts of this section, we detail each of these steps.

4.3 The chosen variant is... a variant!

To prove that V_L is a variant, we have to prove that, when it has reached its lower bound, the desired property is satisfied. So, we have to add the following proof obligation:

$$i_{\mathcal{E}A} \wedge (V_L = V_{0_L}) \rightarrow Q_L \tag{1}$$

Moreover, we also have to prove that once P_L has been true, and until Q_L becomes true, the agent is activated, and thus executes its GDT. This is established by proving the following property, where TC_A is the triggering context of the agent:

$$i_{\mathcal{E}A} \wedge W_L \wedge \neg Q_L \rightarrow TC_A \tag{2}$$

Finally, we also have to prove that no other agent makes the variant increase once P_L has been established until Q_L is established. So, for each other agent A in the system, we have to check for every action α used in a leaf goal G (we recall that *post* and *gpf* of actions contain primed variables):

$$i_{\mathcal{E}A} \wedge C_G \wedge W_L \wedge (post_\alpha \vee gpf_\alpha) \rightarrow pr(V_L) \leq V_L \tag{3}$$

4.4 The witness property... is a witness!

As explained before, we associate to our *leads-to* property L a witness property W_L that verify both following properties:

- **Initialisation** : W_L must be true when P_L is true;
  The property that must be verified is the following:

$$i_{\mathcal{E}A} \wedge P_L \rightarrow W_L \tag{4}$$

- **Finalization** : W_L remains true until Q_L becomes true.
  For each agent, we have to establish that, when it modifies the environment (that is to say, when it performs an action, whether it succeeds or not), if the witness is true before the action, then it is still true after the action has been performed, unless Q_L has become true. So, for each action α associated to a leaf goal G of each agent A, we have to verify:

$$i_{\mathcal{E}A} \wedge C_G \wedge W_L \wedge (post_\alpha \vee gpf_\alpha) \rightarrow pr(W_L \vee Q_L) \tag{5}$$

4.5 The agent progresses

In order to prove that each execution of the GDT of an agent defines a progress towards the establishment of property Q_L, we have to prove that the execution of the main goal performs such a progress, that is to say, the main goal of the agent is a *progress goal*.

Definition 6 (Progress goal (pg)). *We call* Progress Goal *a goal that either makes the variant decrease or establishes property Q_L. For a leads-to property L, we associate to each goal G a boolean pg_{L_G} that is true if and only if G is a progress goal.*

Determining that a goal is a progress goal can be done by inference rules relying on the structure of the gdt, once we know which leaf goals make progress. Moreover, as the gdt execution depends on the success status of goals, we must determine, for each goal, if it is a *success progress goal* and if it is a *failure progress goal*.

Definition 7 (Success Progress Goal (spg)). *We call* Success Progress Goal *a goal that either makes the variant decrease or establishes property Q_L when it is executed and succeeds. For a leads-to property L, we associate to each goal G a boolean spg_{L_G} that is true if and only if G is a success progress goal.*

Definition 8 (Failure Progress Goal (fpg)). *We call* Failure Progress Goal *a goal that either makes the variant decrease or establishes property Q_L when it is executed and fails. For a leads-to property L, we associate to each goal G a boolean fpg_{L_G} that is true if and only if G is a failure progress goal.*

Corollary 2. *A goal is a progress goal if and only if it is a success progress goal and a failure progress goal. So, for every goal G, we have $pg_{L_G} = spg_{L_G} \wedge fpg_{L_G}$.*

In the following paragraphs, we first present how we determine spg and fpg leaf goals, and then, we show how we infer these properties for non-leaf goals. Finally, we give proof schemas that we have to associate to non-spg and non-fpg leaf goals.

Determining the set of spg and fpg leaf goals To determine if a goal is a spg goal, we have to check that when this goal succeeds (and so, establishes its satisfaction condition), it either makes the variant decrease or establishes Q_L. Of course, we must only consider executions of this goal performed when P_L has been true, which is specified by the fact that W_L is true. Hence the following property that must be established by each non lazy[1] spg leaf goal G:

$$(i_{\mathcal{E}A} \wedge W_L \wedge C_G \wedge pr(sc_G)) \rightarrow ((pr(V_L) < V_L \vee pr(Q_L)) \tag{6}$$

In the same way, a goal G is a fpg leaf goal if and only if it verifies the following property:

$$(i_{\mathcal{E}A} \wedge W_L \wedge C_G \wedge pr(gpf_G)) \rightarrow ((pr(V_L) < V_L \vee pr(Q_L)) \tag{7}$$

Please notice that, the gpf of an NS goal being $false$, such goals are *fpg* goals.

Inference of spg and fpg properties A first way to ensure that a non-leaf goal is spg or fpg consists in demonstrating that it is a consequence of the decomposition. In this article, we only detail this process for the SeqAnd/SyncSeqAnd and SeqOr/SyncSeqOr operators.

SeqAnd and SyncSeqAnd : Let G a goal decomposed into G_1 *SeqAnd* G_2.

G is a spg goal, if, in all the cases where G may succeed, the variant decreases. Goal G may succeed in three cases, detailed below:

- Of course, G succeeds when G_1 then G_2 succeed. In this case, if either G_1 or G_2 are spg goals, goal G makes the variant decrease.
- Because of side effects, G may also succeed even if G_1 has failed. Then, G_1 must be fpg.
- Finally, G may also succeed when
  $goal_1$ has succeeded, leading to the execution of G_2, which has failed. In this case, if G_1 is spg or G_2 is fpg, then the variant decreases.

So, we are guaranted that goal G is spg if:

$$\begin{cases} spg_{L_{G_1}} \wedge spg_{L_{G_2}} \\ fpg_{L_{G_1}} \\ spg_{L_{G_1}} \wedge fpg_{L_{G_2}} \end{cases}$$

[1] In this article, we only focus on non lazy goals, that is to say goals that are always executed even if their satisfaction condition is already true when the goal is considered.

As a consequence, here is a sufficient condition to determine that a goal is a spg goal:

$$fpg_{L_{G_1}} \wedge (spg_{L_{G_1}} \vee pg_{LG_2}) \rightarrow spg_{L_G} \tag{8}$$

Now, to determine if G is a fpg goal, we consider both cases where it can fail, that is to say when its first subgoal fails or when its second subgoal fails after the first one has succeded. Hence:

$$fpg_{L_{G_1}} \wedge (spg_{L_{G_1}} \vee fpg_{L_{G_2}}) \rightarrow fpg_{L_G} \tag{9}$$

SeqOr and SyncSeqOr : Let G a goal decomposed into G_1 *SeqOr* G_2.

Goal G may succeed in the three following cases:

- goal G_1 succeeds;
- goal G_1 fails, and then, goal G_2 succeeds.
- goal G_1 fails, and then, goal G_2 fails but, because of side effects, goal G succeeds anyway.

So, we have:

$$spg_{L_{G_1}} \wedge (fpg_{L_{G_1}} \vee pg_{L_{G_2}}) \rightarrow spg_{L_G} \tag{10}$$

The only case where Goal G may fail is when G_1 and G_2 fail. So, the fact that one of theses goals is spg ensure that G is spg. Hence:

$$fpg_{L_{G_1}} \vee fpg_{L_{G_2}} \rightarrow fpg_{L_G} \tag{11}$$

Using satisfaction conditions to determine spg goals Inference rules 8 and 10 to determine if a goal is spg give sufficient properties, but theses properties are not always necessary. A typical example is when the satisfaction condition of a non-leaf goal directly establishes either Q_L or makes the variant decrease. So, for every non leaf goal G that has not been characterized as a spg goal by inference rules described above, we will also verify if property 6 is true. If this is the case, goal G can be identified as an spg goal.

Non-fpg and non-spg leaf goals When a leaf goal G is not a spg goal, we however must prove that this goal does not make the variant increase when it succeeds. So, for each non-spg goal, we have to prove the following formula:

$$i_{\mathcal{E}A} \wedge W_L \wedge C_G \wedge pr(sc_G) \rightarrow pr(V_L) \leq V_L \tag{12}$$

In the same way, for each goal G that is not a fpg goal, we must prove:

$$i_{\mathcal{E}A} \wedge W_L \wedge C_G \wedge gpf_G \rightarrow pr(V_L) \leq V_L \tag{13}$$

Indeed, this is necessary to guarantee that between two steps during which the agent makes the variant decrease, it is not increased in another way.

5 Application on a small example

We choose here of course a very simple example, in order to be able to present all the principles of the proof. We consider a "multiagent system" with only one agent modifying the variant.

Please notice that the system may contain several other agents. In this case, as explained in section 4.3, it has to be proven that their actions do not increase the variant. Taking into account the dynamicity of the environment relies on the same principle, because, as explained in previous articles, the dynamicity of the environment can be modeled by an agent modifiying the state of the environment.

The environment contains two variables, x and d, and is specified by the following invariant:

$$i_{\mathcal{E}} = \begin{cases} x \in \mathbb{N} \\ d \in \mathbb{B} \\ d \leftrightarrow (x > 0 \wedge x \leq 10) \end{cases} \tag{14}$$

Our agent has a behaviour described by the GDT given in figure 2. In this figure, goals names (from A to E) and their *simplified* satisfaction conditions are given. By *simplified* SC, we mean that we did not write the part specifying that the value of other variables are not modified. For instance, the full SC of node D is $y' = 2 \wedge x' = x \wedge d' = d$.

Informally, the goal of this agent is to decrease the value of the environment variable x, by 2 if possible, and otherwise by 1.

Moreover, the triggering context of the agent, its invariant and the gpf of node E are defined so:

$$TC_a \triangleq d \tag{15}$$
$$I_a \triangleq (y \in \mathbb{N}) \tag{16}$$
$$gpf_E \triangleq x' = x \wedge d' = d \tag{17}$$
$$gpf_B \triangleq x' = x \tag{18}$$

We want to prove that this agent establishes the following leads-to property:

$$\Box(x = 10 \rightarrow \Diamond x = 0) \tag{19}$$

We will use x as the variant and d as the witness. To conform to the notation used in the previous section, we have:

$$P_L \triangleq (x = 10) \tag{20}$$
$$Q_L \triangleq (x = 0) \tag{21}$$
$$V_L \triangleq (x) \tag{22}$$
$$V_{0_L} \triangleq (0) \tag{23}$$
$$W_L \triangleq (d) \tag{24}$$

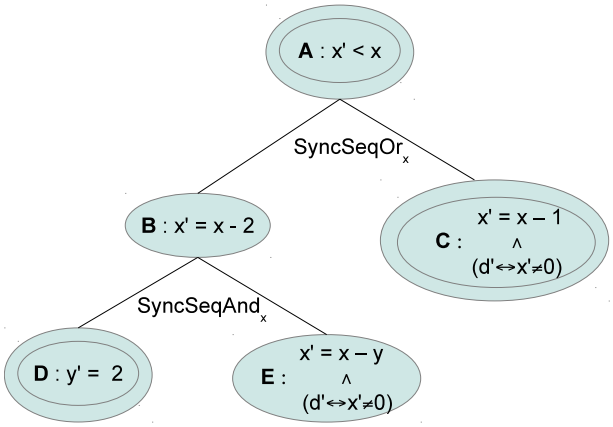

Fig. 2. GDT of the example

This article beeing focused on the proof of liveness properties, we do not present other proofs that must be performed to guarantee the correctness of this specification.

Moreover, in order to give readable formulae, we do not give full contexts of nodes and thus, hypotheses in theorems to prove are simplified.

5.1 Determining leaf progress goals

goal *D* As goal D is a NS goal, it is a fpg goal.

To determine if it is spg, we must establish property 6 for this goal. Thus, we have:

$$W_L \triangleq d$$
$$C_D \triangleq d$$
$$pr(sc_D) \triangleq (y' = 2)$$

Of course, the conjunction of these properties with the invariant does not imply $x' < x$ or $x' = 0$. So, D is not an spg goal. So:

$$spg_D = false \tag{25}$$
$$fpg_D = true \tag{26}$$

goal *E* When goal E is considered, we have:

$$W_L \triangleq d$$
$$C_E \triangleq \begin{cases} d_{-2} \wedge y_{-1} = 2 \wedge x_{-1} = x_{-2} \\ d_{-1} = d_{-2} \wedge y = y_{-1} \wedge x = x_{-1} \end{cases}$$
$$pr(sc_E) \triangleq x' = x - y \wedge (d' \leftrightarrow x' \neq 0)$$

The context of goal E given above is calculated by the context inference rules of the GDT4MAS method. Itexpresses the fact that goal E is considered only after goal D has succeeded when it has been executed in its context.

To establish that E is a spg goal, according to 6, we must demonstrate that the conjunction of these properties imply that the variant decreases, that is to say $x' < x$. This is obvious because, from C_E, we can deduce $y = 2$ and from $pr(sc_E)$, we can deduce $x' = x - y$. So, E is a spg goal.

We also have to determing if E is a fpg goal, thanks to rule 7. Among the hypotheses of this rule, we have gpf_E (which implies $x' = x$, see 18) and requires as conclusion either $x' < x$ (which cannot be true!) or $x' = 0$ which cannot be guaranted because the context does not provide any knowledge about the value of x. So, goal E is not a fpg goal.

So, we have:

$$spg_E = true \tag{27}$$
$$fpg_E = false \tag{28}$$

goal *C* About goal C, we have the following properties:

$$W_L \triangleq d$$
$$C_C \triangleq (d_{-2} \wedge x_{-1} = x_{-2} \wedge x = x_{-1})$$
$$pr(sc_C) \triangleq (x' = x - 1 \wedge (d' \leftrightarrow x' \neq 0))$$

To establish that goal C is a spg goal, we must try to establish rule 6. This rule requires to prove, from the conjunction of the above properties, that the variant decreases ($x' < x$) or that property Q_L is true. This is obvious because, from sc_C, we deduce that $x' = x - 1$, which implies $x' < x$. So, goal C is a spg goal. Moreover, as this goal is a NS goal, this is also a fpg goal. So we have:

$$spg_C = true \tag{29}$$
$$fpg_C = true \tag{30}$$

Conclusion As a conclusion, we know that no leaf goal make the variant increase. Moreover, spg goals and fpg goals are respectively the following:

$$SPG = \{C, E\} \tag{31}$$
$$FPG = \{C, D\} \tag{32}$$
$$PG = \{C\} \tag{33}$$

5.2 Inference of the progress property

Goal *B* To determine if goal B is a spg goal, we apply rule 8 that provides the following sufficient condition to guarantee that goal B is spg:

$$fpg_D \wedge (spg_D \vee pg_E)$$

However, D is not a spg goal and E is not a pg goal. Thus, with this rule, we cannot determine that goal B is spg. So, we try to apply rule 6. Considering goal B, we have:

$$W_L \triangleq d$$
$$C_B \triangleq d$$
$$pr(sc_B) \triangleq x' = x - 2$$

And we have to establish that the conjunction of these formulae implies either $x' < x$ of $x' = 0$. As sc_B implies $x' = x - 2$, we obviously have $x' < x$. So, B is a spg goal.

We now have to determine if goal B is a fpg goal, applying rule 9:

$$fpg_D \wedge (spg_D \vee fpg_E)$$

As goal E is not a fpg goal and D is not a spg goal, we can deduce that goal B is not a fpg goal. So we have:

$$spg_B = true \tag{34}$$
$$fpg_B = false \tag{35}$$

Goal *A* To determine if goal A is a spg goal, we apply rule 10, which gives:

$$spg_B \wedge (fpg_B \vee pg_C)) \rightarrow spg_A$$

As we have established before that goal B is spg (34) and that goal C is pg (33), we can establish that goal A is a spg goal.

Conclusion Goal A being a NS goal and a spg goal, we now know that each execution of the GDT of the agent makes the variant decrease.

5.3 The chosen variant is a variant

Correctness According to equation 1, to prove that the chosen variant is effectively a variant, we have to prove:

$$i_{\mathcal{E}A} \wedge x = 0 \rightarrow x = 0$$

This is obviously true !

Activation According to equation 2, we have to prove:

$$i_{\mathcal{E}A} \wedge d \wedge \neg(x = 0) \rightarrow (x = 10 \vee d)$$

Once again, this formula is obviously true.

5.4 the witness is a witness

Initialisation From formula 4, weh have to verify:

$$x = 10 \wedge (d \leftrightarrow (x > 0 \wedge x \leq 10)) \rightarrow d$$

This is still an obviously true formula.

Finalization We have to apply proof schema 5 for every leaf goal (and we recall here that, according to GDT4MAS principles, the gpf of an NS action is $false$).

Goal D The NS action δ associated to goal D is defined by:

$$post_\delta \triangleq y' = 2 \wedge d' = d$$
$$gpf_\delta \triangleq false$$

So, with the context of goal D given above, we must establish:

$$i_{\mathcal{E}A} \wedge d \wedge d \wedge ((d' = d \wedge y' = 2) \vee false) \rightarrow pr(d \vee x = 0)$$

That can be simplified into:

$$i_{\mathcal{E}A} \wedge d \wedge d' = d \wedge y' = 2 \rightarrow d' \vee x' = 0$$

This property is obviously true (as d and $d' = d$ can be found among the hypotheses).

Goal E The action η associated to goal E is defined by:

$$post_\eta \triangleq x' = x - y \wedge (d' \leftrightarrow x' \neq 0)$$
$$gpf_\eta \triangleq x' = x \wedge d' = d$$

Using C_E given above, applying proof schema 5, we obtain the following proof obligation:

$$i_{\mathcal{E}A} \wedge d_{-2} \wedge y_{-1} = 2 \wedge x_{-1} = x_{-2}$$
$$d_{-1} = d_{-2} \wedge y = y_{-1} \wedge x = x_{-1} \wedge d$$
$$((x' = x - y \wedge (d' \leftrightarrow x' \neq 0)) \vee (x' = x \wedge d' = d))$$
$$\rightarrow pr(d \vee x = 0)$$

In order to simplify the explanation of the demonstration (that can be however easily performed by an automatic prover), we remove useless hypotheses. So, we have to prove:

$$(d \wedge x' = x - y \wedge (d' \leftrightarrow x' \neq 0)) \vee (d \wedge x' = x \wedge d' = d)$$
$$\rightarrow d' \vee x' = 0$$

The structure of this formula being $a \vee b \rightarrow c$, we will successively demonstrate $a \rightarrow c$ and $b \rightarrow c$.

- $(d \wedge x' = x - y \wedge (d' \leftrightarrow x' \neq 0)) \rightarrow d' \vee x' = 0$
  We use a proof-by-case on the value of d'. Either d' is true, and so, the goal is true, or d' is false. In the latter case, according hypothesis 2, $x' = 0$, and so the goal is true. QED.
- $(d \wedge x' = x \wedge d' = d) \rightarrow d' \vee x' = 0$
  As d and $d' = d$ are hypotheses, we obviously deduce d'. QED.

So the proof obligation generated by applying proof schema 5 to goal E is true.

Goal C The action associated to goal C is defined by:

$$post_\gamma \triangleq x' = x - 1 \wedge (d' \leftrightarrow x' \neq 0)$$
$$gpf_\gamma \triangleq false$$

Using C_C and applying proof schema 5 to goal C, we have to prove:

$$\left.\begin{array}{l} i_{\mathcal{E}A} \wedge d_{-2} \wedge x_{-1} = x_{-2} \wedge x = x_{-1} \wedge d \\ ((x' = x - 1 \wedge (d' \leftrightarrow x' \neq 0)) \vee false) \end{array}\right\} \rightarrow pr(d \vee x = 0)$$

In order to simplify the explanation of the demonstration (that can be however easily performed by an automatic prover), we remove useless hypotheses. So, we have to prove:

$$(d' \leftrightarrow x' \neq 0) \rightarrow (d' \vee x' = 0)$$

The proof is obvious: either d' is true, and so, the goal is true, or d' is false and so, from hypotheses, $x' \neq 0$ is false, and so, $x' = 0$. QED.

5.5 Conclusion

Following the proof system described in section 4, we have been able to establish that an agent whose behaviour is described by the gdt given in figure 2 satisfies a liveness property that is not a part of its main goal.

6 Comparison with other works

Several formal specification languages dedicated to multiagent systems exist. However, they are often not dedicated to the proof. This is for instance the case of 2apl [7], that is finally more a programming language than a specification language suited to proof. MetateM [8] gives the developer a way to specify properties, and the system controls that the execution does not violate these properties. However, this is a proof-by-construct process; this means that the proof is performed only at the execution time, and if the initial conditions change, a new proof (consisting in an execution of this new initial state) must be performed.

Finally, most works dealing with the verification of multiagent systems rely on model-checking principles. One of the most recent work in this area is the definition of AJPF [9], a model-checker relying on JPF [10] and the *Agent Infrastructure Layer AIL*. This is, as far as we know, the only system that proposes a way to verify leads-to properties on multi-agent systems. However, a first drawback of the method is the time taken by the system to establish the property (several hours for a very simple system). Of course, a more optimized model-checker such as spin [11], may greatly reduce the time required. However, such systems remain dedicated to small-size systems. Moreover, such systems have a more serious drawback: also they can be used to prove a property such as the property we have proven in section 5: $\Box(x = 10 \rightarrow \Diamond x = 0)$, they cannot be applied when the left-hand side property (here, $x = 10$) characterize an infinite number of states. For instance, if we would be interested in proving the following leads-to property: $\Box(x \geq 10 \rightarrow \Diamond x = 0)$, a model-checking-based method would fail, whereas the process we propose would be as efficient as it is in the given example.

The same problem can be found with MCMAS [12], which moreover does not provide a way to verify leads-to properties. This model-checking technique tries to verify formulae specified in propositionnal logic, as AJPF. The main disadvantage of this technique is that, relying on propositionnal logic, proofs cannot be generalized on systems of any size. For instance, in the cited article, it is shown that the verification of the dinning cryptographers must be performed for each number of cryptographers we are interested in. Moreover, even if the time taken for 10 cryptographers is quite good, performances decrease dramatically when the number of cryptographers increase. Finally, with such a technique, to prove that the MAS work with any number of cryprgrapher, an infinite number of verifications must be performed, requiring, of course, an infinite time.

Indeed, as model checking techniques may be applided on systems with several millions of states, their complexity is a critical aspect that must be taken into consideration. But with theorem proving techniques, this criterion is quite less important. Indeed, each proof requires a very short time, and the number of proofs is very low, compared to the number of states generated in model checking techniques (for instance, even on a very large industrial system, less that 50,000 proofs had to be verified [13]). For instance, with the GDT4MAS model, if we call $n(t)$ the number of nodes of an agent type t and T the set of agent types, the number of proofs to perform is approximately $2\Sigma_{t \in T} n(t)$.

7 Conclusion and Perspectives

In this article, we have shown that the GDT4MAS model, that was mainly dedicated to the proof of invariant properties, can be extended to prove liveness properties such as lead-to properties. As other proof obligations of the GDT4MAS framework, the new proof obligations generated are easily proven by an automatic theorem prover such as PVS.

This kind of proof can help in analyzing the behaviour of a MAS. In the work presented here, we have only considered liveness properties associated to a single agent. Of course, more general liveness properties at the system level will have to be considered, especially properties that are established not only by a single agent, but by a subset of the agents in the system. This is a short-term perspective. Moreover, at it is classically performed in standard verification systems, our proof system can only prove leads-to properties $P\ leads-to\ Q$ for which there is a continuous progress to Q once P has been true. In a multiagent system where agents are fully autonomous, we also have to consider properties for which this progress is not continuous. This is a long-term perspective for us.

References

1. Mermet, B., Simon, G.: GDT4MAS: an extension of the GDT model to specify and to verify MultiAgent Systems. In *et al.*, D., ed.: Proc. of AAMAS 2009. (2009) 505–512
2. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988)
3. Lamport, L.: The syntax and semantics of tla$^+$. Part 1: Definitions and Modules (June 1996)
4. Milner, R., Parrow, J., Wlaker, D.: A calculus of mobile processes. Journal of Information and Computation **100** (1992)
5. Mermet, B., Simon, G., Saval, A., Zanuttini, B.: Specifying, verifying and implementing a MAS: A case study. In Dastani, M., Segrouchni, A.E.F., Ricci, A., Winikoff, M., eds.: Post-Proc. of ProMAS'07. Number 4908 in Lecture Notes in Artificial Intelligence, Springer (2007) 172–189
6. SRI International: PVS http://pvs.csl.sri.com.
7. Dastani, M.: 2APL: a practical agent programming language. Journal of Autonomous Agents and Multi-Agent Systems **16** (2008) 214–248
8. Fisher, M.: A survey of concurrent METATEM – the language and its applications. In Gabbay, D.M., Ohlbach, H.J., eds.: Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827), Springer-Verlag: Heidelberg, Germany (1994) 480–505
9. Dennis, L., Fisher, M., Webster, M., Bordini, R.: Model checking agent programming languages. Automated Software Engineering **19**(1) (2012) 5–63
10. NASA: Java Path Finder http://babelfish.arc.nasa.gov/trac/jpf.
11. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Softw. Eng. **23** (May 1997) 279–295
12. Lomuscio, A., Qu, H., Raimondi, F.: Mcmas: A model checker for the verification of multi-agent systems. In Bouajjani, A., Maler, O., eds.: Computer Aided Verification. Volume 5643 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2009) 682–688

13. Abrial, J.R.: Formal methods in industry: achievements, problems, future. In: International Conference on Software Engineering. (2006) 761–768