How Variables Graphs May Help to Correct Erroneous MAS Specifications

Bruno Mermet, Gaële Simon

How Variables Graphs May Help to Correct Erroneous MAS Specifications

Bruno Mermet and Gaële Simon

CNRS-GREYC - UMR 6072 and University of Le Havre, Caen, France
firstName.lastName@unicaen.fr

Abstract. This paper is situated in the context of multi-agent systems validation using theorem proving techniques. This document presents a preliminary case study, which is a part of a broader investigation aiming to explore whether such techniques could aid developers in detecting and characterizing errors in MAS specifications. Indeed, regardless of the verification system used (model checking or theorem proving), understanding the reason of a failure in order to correct the specification is most of the time rather difficult. In this article, we propose a method that may help in this task. This method relies on the variables (and their dependencies) that appear in proof obligations generated by GDT4MAS, a specification and verification method dedicated to Multi-Agent Systems. Graphs generated thanks to dependencies between variables occurring in an unproved theorem may indeed help to identify certain types of mistakes, giving a way to correct the specification.

Keywords: Multi-Agent Systems, Proof Failure, Debugging.

1 Introduction

This article is set within the broader context of validating Multi-Agent Systems (MAS), focusing particularly on the tuning stage. Over the past few years, considerable effort has been dedicated to validating MAS through proof techniques. That is precisely why we employ the GDT4MAS model in this work [12], which offers a dual benefit of formal tools for specifying Multi-Agent Systems and an automated proof system. By utilizing a formal specification, the proof system generates a set of *Proof Obligations* that serve as a guarantee for the system's correctness.

Simultaneously, we have embarked on studying how to address the question: "What happens when the theorem prover fails to complete the proof?" Specifically, we explore the potential to extract valuable insights from these failures, referred to as *proof failures*, to aid in debugging the Multi-Agent System (MAS). Addressing this question within a general context presents challenges. Firstly, it is worth noting that a proof failure can transpire in three distinct scenarios:

– The first scenario occurs when a true theorem is unprovable, as demonstrated by Gödel's Incompleteness Theorem. This situation arises when the theorems

generated by GDT4MAS involve first-order logic formulae with arithmetic, which aligns with Gödel's findings that there exist non-provable true theorems within such cases;
– The second scenario occurs when a true theorem cannot be proven automatically by the prover due to the semi-decidability of first-order logic. This means that there is no universal automatic strategy capable of proving all possible theorems. Instead, an expert must provide an *ad hoc* strategy tailored to the specific situation;
– The third scenario arises when an error in the specification of the MAS results in the generation of a false theorem that cannot be proven as a consequence.

So, when a proof failure is considered, the first problem is to determine, among the three cases presented above, which applies. Providing complete explanations here would be too lengthy and would divert from the topic. Nevertheless, it's important to note that the proof system is specifically designed to produce theorems that can be proven by standard provers' strategies, without requiring the expertise of a human. Additionally, true theorems that are unprovable typically do not apply to real-life situations. Therefore, in most instances, proof failures are indicative of errors in the specification, and this is the context that will be considered in the following sections.

Our study revolves around the following question: If certain proof obligations generated are not automatically proven, can we leverage this information to assist in correcting the specification of the MAS? Consequently, the primary notion is to examine whether proof failures can be utilized to identify and rectify bugs in the MAS specification.

Contrary to the perspective presented in [2], where the authors propose that proof-based approaches are solely intended for MAS validation while other methods should be employed for debugging and tuning, our objective is to utilize proof failures to identify and rectify early errors in the design of a MAS. In doing so, we aim to capture and correct these mistakes at a very early stage of the MAS development process.

This article commences with a concise overview of prior research on debugging Multi-Agent Systems (MAS). Subsequently, in section 3, we introduce the GDT4MAS model, along with the proof mechanism and the accompanying tools. Section 4 informally presents the notion of variables graph that we propose to study. Section 5 exemplifies the use of this tool on a case study. In section 6, we formalise the work presented in the two previous sections. Lastly, in the final section, we conclude the presented work and provide insights into the future direction of our research in this domain.

2 State of the Art

Ensuring the correctness of a Multi-Agent System (MAS) is an essential and challenging problem, as has been repeatedly acknowledged in previous studies [4]. Similar to classical software, there are primarily two methods for assessing the

correctness of a Multi-Agent System (MAS): proof and testing, as elucidated in references [12, 13]. In this section, we focus on works dealing with the tuning of erroneous systems. However, as far as we know, there are no work proposing a way to automatically correct wrong specifications.

2.1 Test

The majority of research on debugging Multi-Agent Systems (MAS) primarily focuses on testing methodologies. These approaches aim to identify potential issues and provide one or more test cases that can reproduce the problem. Testing can be carried out at various levels, as outlined in [18]:

- At the unit level, the objective is to detect issues within individual components, as exemplified by the work presented in [27]. This involves a direct adaptation of conventional testing techniques to the context of Multi-Agent Systems (MAS);
- At the agent level, several approaches have been developed to address debugging within Multi-Agent Systems (MAS). Some propose the utilization of "xUnit" tools to specify unit tests [25], often incorporating mock agents into the system [1]. Other approaches involve the inclusion of dedicated testing agents within the MAS [19, 15]. Additionally, certain methods employ evolutionary techniques to monitor the system under various conditions, enabling the management of a large number of initial situations [17];
- At the system level, there are relatively few approaches that specifically address testing principles within the context of Multi-Agent Systems (MAS). This scarcity can be attributed to the inherent complexity of the problem, which is extensively discussed in [16]. Nonetheless, there have been notable works in this domain [6, 25]. Among these, [20] stands out as one of the few studies that acknowledges the importance of the goal notion within MAS testing.

2.2 Trace Analysis

There is another type of research that addresses the debugging of multi-agent systems, which involves trace analysis. This research primarily concentrates on three tasks related to debugging: detecting the problem, pinpointing the possible causes, and determining the root cause [10, 26, 24].

Two different methods are used to analyze traces: the first involves examining the sequence of messages exchanged between agents, while the second utilizes data mining techniques to verify if the knowledge provided by the system designer can be identified within the multi-agent system, with the objective of detecting and explaining bugs. A combination of these two approaches is explored in the paper [5].

2.3 Visualisation

While visualisation tools for multi-agent systems are potentially useful, there are only a few studies that focus on them. The main challenge in designing such tools

is the difficulty of creating relevant views due to the vast number of interacting entities. One approach to tackling this issue is to generate views from traces, as proposed in [26], while others concentrate on real-time visualisation, as explored in [15].

2.4 Proof Failures

The utilization of proof failures as a prospective domain remains largely unexplored and requires further investigation. Some studies suggest equipping the prover with tools that can leverage proof failures [8]. While [3] presents a few ideas regarding the application of these proof failures, they have yet to be implemented.

2.5 And Outside of the Multi-Agent Systems World

Several recent works have been developed to help to manage proof failures thanks to test-case generation. This is for instance the case of the STaDy tool presented in [23]. This work has been extended more recently to help to determine the reason of a proof failure (prover weakness or subcontract weakness) [22]. As in [9] and in [7], they use counterexamples generation to help to determine the reason of the proof failure. However, none of this work is dedicated to multi-agent systems with autonomous agents. Moreover, as these work are not dedicated to a structured model as a GDT4MAS specification, they cannot use the proof obligation structure to provide more specific information. There are also works proposing to use abduction to help uderstand errors [11], but as far as we know, these works have not been applied to programs.However, it is clear that works on this type of research in the general case, and a fortiori in the field of MAS, are very rare.

3 The GDT4MAS Model

This approach, which combines a dedicated model and proof system, offers several compelling features for the design of Multi-Agent Systems (MAS). It encompasses a formal language to describe agent behavior and desired properties, utilizes expressive and widely recognized first-order logic, and incorporates an automated proof process. The GDTM4MAS method, which is elaborated further in [12, 13], is briefly introduced here.

3.1 Main Concepts

The GDT4MAS method necessitates the specification of several concepts outlined herein.

Environment The environment of the MAS is defined by a set of typed variables and an invariant property $i_{\mathcal{E}}$.

Agent Types Each agent type is specified by a set of internal typed variables, an invariant and a behaviour. The behaviour of an agent is mainly defined by a *Goal Decomposition Tree* (GDT). A GDT, or goal decomposition tree, is a hierarchical structure of goals, with the root representing the primary goal of the agent. Each goal is associated with a plan, which, when successfully executed, results in the achievement of the associated goal (known as the "parent goal"). Plans can consist of either a single action or a set of subgoals, which are linked together using a "decomposition operator." A goal G is typically defined by its name n_G, a "satisfaction condition" sc_G, and a "guaranteed property in case of failure" gpf_G.

The satisfaction condition (SC) of a goal is formally specified by a formula that must be true when the execution of the plan associated to the goal succeed. Otherwise, the guaranteed property in case of failure (GPF) of the goal specifies what is however guaranteed to be true when the execution of the plan associated to the goal fails (this has of course no sense when the goal is said to be a *NS goal*, that is to say a goal whose plan always succeed).

SC and GPF are referred to as *state transition formulae* (STF) because they establish a connection between two states: the *initial state*, which is the system state just before the agent attempts to solve the goal, and the *final state*, which is the system state after the agent has completed the execution of the goal's associated plan. In an STF, a variable v can be primed or unprimed. The primed notation (v') represents the value of the variable in the final state, while the unprimed notation (v) represents the value of the variable in the initial state. A STF can be non-deterministic when, given an initial state, multiple final states can satisfy it. For example, consider the STF $y' < y$. This implies that the value of variable y must be lower after the execution of the goal's plan compared to its initial value. If y is initially 10, final states with values of 2, 5 or 9 for y would satisfy this STF.

Decomposition Operators GDT4MAS offers multiple decomposition operators to specify different types of behaviors. However, in this article, we only employ two of these operators:

- the `SeqAnd` operator in GDT4MAS dictates that the subgoals of a plan must be executed in a specific order, progressing from left to right on the graphical representation of the GDT. If the agent's behavior adheres to this order, successfully achieving all the subgoals results in the attainment of the parent goal. However, if the execution of the first subgoal fails, the subsequent subgoal is not executed;
- The `SyncSeqAnd` operator operates similarly to the SeqAnd operator, with the added capability of locking a set of variables in the environment. This locking mechanism prevents other agents from modifying these variables until the execution of the plan is completed.

Actions Actions are defined by a precondition that outlines the states in which they can be executed, and a postcondition that specifies the effect of the action using a state transition formula (STF).

Agents Agents are defined as instances of agent types, with effective values for the potential parameters.

3.2 GDT Example

Fig. 4 depicts the GDT (Goal Dependency Tree) of an agent comprising three goals, visually represented by ellipses accompanied by their names and satisfaction conditions (SC). The root goal, labeled as goal A, is decomposed into two subgoals, B and C, utilizing the SeqAnd operator. Both subgoals B and C are leaf goals, and as such, an action is associated with each of them, represented by arrows in the diagram.

3.3 Proof Principles

General Presentation The proof mechanism aims at proving the following properties:

- Agents preserve their invariant properties [13];
- Agents preserve the invariant properties of the environment;
- The agents behaviours are consistent; (plans associated to goals are correct);
- Agents respect their liveness properties. These properties formalise expected dynamic characteristics.

In addition, the proof mechanism utilized in the system depends on "proof obligations" (PO). These POs are properties that must be proven to ensure the accuracy of the system and can be automatically generated from a GDT4MAS specification. They are expressed in first-order logic and can be verified by any first-order logic prover. Furthermore, the proof system is designed to be compositional: The proof of an individual agent type's correctness is broken down into several smaller, independent proof obligations. In most cases, the proof of a given agent type can be established independently of the other types.

Context Notion A key notion that has been associated to the proof process is the *context of execution of a goal* notion. This is a predicate summarising the states in which a goal of a GDT may be executed. The context of a goal can be automatically inferred in a top-down way, starting from the *triggering context* (TC) of the agent. The TC of an agent is a predicate specifying when the agent begins to act.

The context inference mechanism is illustrated in the sequel on a small example presented in Fig. 1.

In this example, we suppose that the triggering context associated to this GDT is $x < 10$, where x is an environment variable. The invariant property of the environment is $0 \leq x \leq 20$. Moreover, y is an internal variable of the agent. The invariant property of the agent is $y > 0$.

As the invariant of the environment is an invariant, it is obviously true when the execution of goal A is about to begin. Moreover, by definition, the triggering

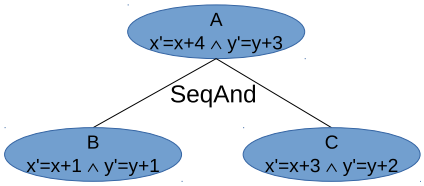

Fig. 1. Mathematical GDT.

context of the agent is also true at this moment. So, the context of goal A is the following:

$$C_A \equiv x < 10 \wedge 0 \leq x \wedge x \leq 20 \wedge y > 0$$

The plan associated to goal A consists in directly executing goal B. So, the context of goal B is the same as the context of goal A. So we have:

$$C_B \equiv x < 10 \wedge 0 \leq x \wedge x \leq 20 \wedge y > 0$$

We now consider goal C. Schema in Fig. 2 represents the trace of the system evolution.

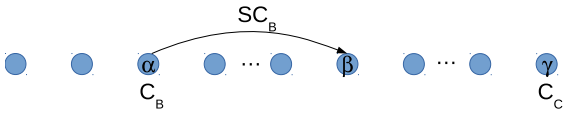

Fig. 2. Trace of the evolution of the system.

In this trace, three states can be identified:

- state α is the state in which the system is when the agent begins to act (and so, will try to achieve goal A);
- state β is the state in which the system is when the execution of goal B ends;
- state γ is the state in which the system is when the execution of goal C begins.

As it can be seen, states α and β are not necessarily consecutive states, because goal B can be itself decomposed into several subgoals, and because other agents may act meanwhile. Similarly states β and γ are not necessarily consecutive because other agents may act in the system in the meantime. However, we know that:

- context of node B is true in state α:

– the agent invariant and the environment invariant are true in state β and γ;
– the satisfaction condition of goal B is true between states α and β. In other words, the value of x in state β, written x_β, is equal to the value of x in state x_α increased by 1 and the value of y in state β is equal to y_α increased by 1;
– the value of y in state γ is equal to the value of y in state β because y is an internal variable of the agent and so, it cannot be modified between states β and γ (only the owner agent of internal variables can modify them, and the agent we consider cannot perform any action between states β and γ).

Thus, the context of goal C could be something like that:

$$\begin{cases} x_\alpha < 10 \wedge 0 \leq x_\alpha \wedge x_\alpha \leq 20 \wedge y_\alpha > 0 \\ 0 \leq x_\beta \wedge x_\beta \leq 20 \wedge y_\beta > 0 \\ 0 \leq x_\gamma \wedge x_\gamma \leq 20 \wedge y_\gamma > 0 \\ x_\beta = x_\alpha + 1 \wedge y_\beta = y_\alpha + 1 \\ y_\gamma = y_\beta \end{cases}$$

In the sequel, goals α, β and γ will be respectively denoted -2, -1 and 0. So, the context of goal C is the following:

$$C_C \equiv \begin{cases} x_{-2} < 10 \wedge 0 \leq x_{-2} \wedge x_{-2} \leq 20 \wedge y_{-2} > 0 \\ 0 \leq x_{-1} \wedge x_{-1} \leq 20 \wedge y_{-1} > 0 \\ 0 \leq x_0 \wedge x_0 \leq 20 \wedge y_0 > 0 \\ x_{-1} = x_{-2} + 1 \wedge y_{-1} = y_{-2} + 1 \\ y_0 = y_{-1} \end{cases} \quad (1)$$

Proof Schema The GDT4MAS method defines several *proof schemas*. These proof schemas are formulae that are used to generate *proof obligations*. For instance, the proof schema that is used to verify that the decomposition of a goal A into two subgoals B and C thanks to the `SeqAnd` operation is correct is:

$$C_C \wedge [SC_C]^{0 \to 1} \to [SC_A]^{-2 \to 1}$$

In this formula, $[P]^{a \to b}$ represents the predicate obtained by substituting in P every occurrence of variables v without a subscript by v_a and by substitution in P every occurrence of variables v' by v_b. So, considering the example in Fig. 1, the proof obligation generated it the following:

$$C_C \wedge (x_1 = x_0 + 3 \wedge y_1 = y_0 + 2) \to (x_1 = x_{-2} + 4 \wedge y_1 = y_{-2} + 3) \quad (2)$$

The verification of invariant properties is also verified thanks to proof schemas that must be applied to each action of a the GDT.

PVS PVS (Prototype Verification System) [21] is a proof environment that relies on a theorem prover for managing specifications expressed in a typed higher-order logic. The system includes a set of predefined theories for handling various concepts, such as set theory and arithmetic. The theorem prover operates interactively, allowing the user to intervene in the proof process as needed.

One of the goal of the GDT4MAS method is to minimise the user intervention. It is the reason why Proof Obligations are generated in a way maximising the success rate of automatic proof strategies. The strategy of PVS used is a very general one called *grind* that uses, among others, propositional simplification, arithmetic simplification, skolemisation and the disjunctive simplification.

The PVS proof process uses the sequent calculus. Thus, each theorem that must be proven by PVS is at first translated into an initial sequent with an empty antecedent and a consequent consisting in the theorem to prove. In our context, the theorem to prove is often an "imply" formula. In such a case, the initial sequent is transformed into a sequent whose antecedent is the left part of the imply and whose consequent is the right part of the imply. So, PVS builds a proof tree where each sequent is decomposed into one ore several sequents (called *child node*) using deduction rules. The proof is a success if, in the proof tree built by PVS, each lief node is proven.

3.4 Execution Platform

The GDT4MAS model is supported by a too which provides the following features:

- execution of a GDT4MAS specification;
- generation of proof obligations in the PVS language;
- proof of a GDT4MAS specification using PVS.

This platform provides various modes for executing specifications, including a *random* mode where agents are activated randomly and a *trace* mode where agents are activated in a predetermined order. Dynamic charts display the values of selected variables in real-time during execution, and a log console provides information on system activity such as the activated agent, the goal being executed, and the action being executed. These features make the GD4MAS model an ideal candidate for our experiments.

4 Variables Graphs

As shown in formula 2, a proof obligation is a horn-clause $h(v) \rightarrow g(v)$, where v is a set of variables considered at different instants. There are a few cases where a relation between the hypotheses and the goal is not required, namely:

- when hypotheses entail $false$;
- when the goal is $true$.

In all the other cases, variables used in the goal formula $g(v)$ must occur in the hypotheses ($h(v)$). Moreover, a relationship between them must be deduced from the hypotheses. And in some cases, it may be required that properties on these variables can be entailed from the context of the goal (it is a necessary condition, not a sufficient condition). So, we decided to product variables graph for each unproved proof obligation, and to study different cases of proof failures.

Definition 1. *A variables graph, associated to a Proof Obligation* $P = h(v) \rightarrow g(v)$ *where* $h(v)$ *in its normal conjunctive form, is represented by a tuple* (V, L, V_c, V_g) *where:*

- V *is the set of variables occurring in* P *;*
- $L \in V^2$ *is the relation determining each couple of variables occurring in the same term of* P *;*
- $V_c \in V$ *is the set of variables associated to the older state in* P *;*
- $V_g \in V$ *is the set of variables in* $g(v)$*.*

Proof obligation 2 is not a Horn Clause, so, it must be split into two proof obligations (remind that the Context C_C is given in formula 1):

$$C_C \wedge (x_1 = x_0 + 3 \wedge y_1 = y_0 + 2) \rightarrow (x_1 = x_{-2} + 4) \tag{3}$$

$$C_C \wedge (x_1 = x_0 + 3 \wedge y_1 = y_0 + 2) \rightarrow (y_1 = y_{-2} + 3) \tag{4}$$

The graphs associated to these two proof obligationsare represented in Fig. 3. On this figure, variables of V_c are hatched and variables of V_g have a pink background. The left part of the Fig. represents the graph associated to proof obligation 3 whereas the right part represents the graph associated to proof obligation 4.

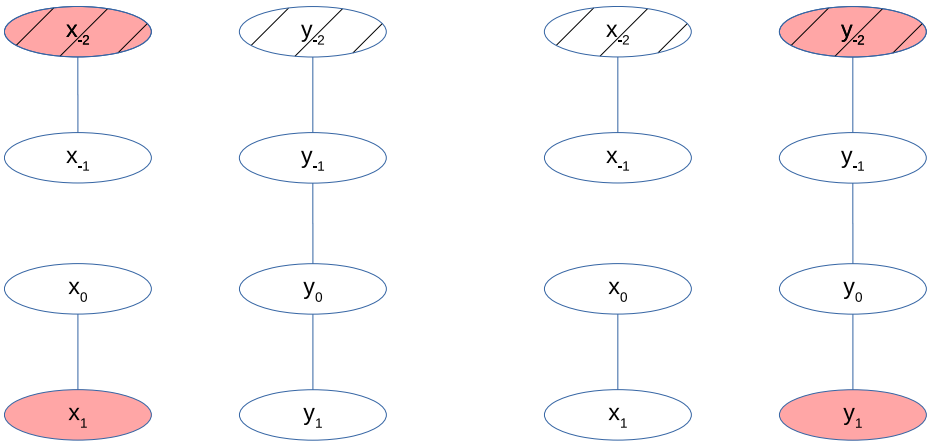

Fig. 3. Variables graphs associated to proof obligations.

On these graphs, it is easy to understand that the first proof obligation may be wrong: a link between x_1 and x_{-2} must be established, but it seems that there is no way to do that from the hypotheses.

5 Using Proof Failures To Help to Debug MAS

When a MAS is bugged, this may be induced by several kinds of such errors. Here are a few examples:

- the decomposition operator utilized to define the plan linked to a goal is incorrect, thereby causing the failure to achieve the parent goal even when subgoals are achieved;
- the associated Satisfaction Condition of a goal is inadequate in its strength and does not offer the necessary properties for later use;
- The agent's *triggering context* is incorrect, which may lead to its activation at inappropriate times or its failure to activate when necessary;
- the invariant associated to an agent type is wrong or too weak.

In [14], the authors have shown how the localisation of proof failures can help to correct wrong specifications. In this paper, we show how a study of variables graphs generated from unproved proof obligations can help to propose solutions to correct the specification. To illustrate our work, we will refer a case study that we describe in the next section.

5.1 Case Study

Description of the MAS Used Our case study revolves around a producer-consumer system. In its basic version, the system consists of two agents belonging to two distinct agent types: the Producer type and the Consumer type. However, as we will demonstrate, from a proof standpoint, the analysis of the system remains the same regardless of the number of agents for each type. Nevertheless, there are execution-related differences that we will explore. The environment incorporates a variable called $stockE$, which is an integer representing the quantity of resources (e.g., pounds of flour) that the producer has already placed in the environment.

In order to generate some resources, the producer agent exploits internal resources of another type (such as wheat), which are tracked by an internal variable called $stockPro$. This variable represents the amount of wheat (in pounds) currently owned by the producer. To create a single unit of a particular environmental resource (such as flour), the producer consumes one unit of its own resources (one pound of wheat). This production process is described by the GDT (Goal-Delegation Tree) of the Producer type, as shown in Fig. 4, which consists of three goals. Goal B represents the consumption of the internal resource, while goal C models the creation of the new resource in the environment.

On the other hand, the consumer is responsible for producing resources of a different type, such as bags of bread. The amount of resources produced is

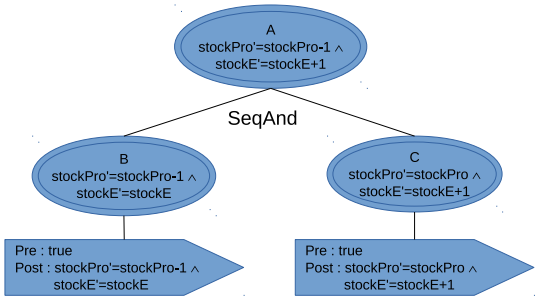

Fig. 4. GDT of the producer type.

represented by an internal variable of the Consumer type, called *stockCons*. To produce these resources, the consumer needs to use two resources from the environment (producing a bag of bread requires two pounds of flour). The GDT of the Consumer type (shown in Fig. 5) formalizes this process: goal B represents the consumption of two resources from the environment, and goal C represents the production of a new resource of the third type.

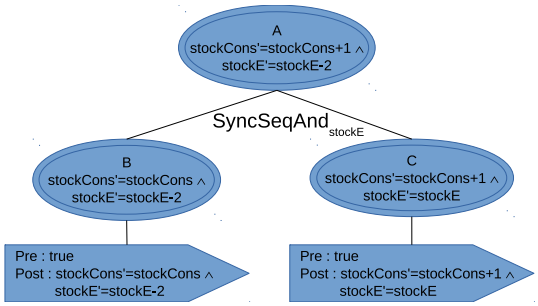

Fig. 5. GDT of the consumer type.

Additionally, there is a further limitation to the process described above. The environment can only stock two resources of the environment type at a time (in the example, there is only enough space on the shelf for two pounds of flour). This implies that if there are already two resources in the environment, one must be consumed before the producer can add a new one. The triggering context for the producer formalizes this constraint as follows: $stockPro > 0 \wedge stockE < 2$. With this triggering context, the producer can only be activated if it still has resources and if there is room in the environment to store its production. Furthermore, the environment has the following invariant: $stockE > -1 \wedge stockE < 3$. This invariant specifies the range of valid values for the $stockE$ variable, in accordance

with the aforementioned constraint. The Producer type also has an associated invariant: $stockPro > -1$. This is because the number of internal resources cannot be a negative value.

The triggering context for the Consumer type is straightforward: $stockE > 1$. This implies that the consumer cannot act unless there are at least two resources available in the environment. The invariant associated with this agent type is similar to that of the Producer, specifying that the number of its internal resources is a non-negative integer: $stockCons > -1$.

5.2 Identifying Proof Failure Cases

Concurrency Problem A first kind of proof failure may arise if an environment variable is not locked by an agent when it should be. In the case study presented above, this is for instance the case for the producer agents : the main goal specifies that the stock in the environment should be increased by one and, indeed, the two subgoals specify that the $stockE$ variable is first maintained, and then increased by one. However, as this variable is not locked, between the executions of the first and second subgoals of the root goal, another agent may modify the value of $stockE$.

The first proof obligation corresponding to the verification of this decomposition is the following:

$$
\begin{array}{l}
stockPro_{-2} > 0 \wedge stockE_{-2} < 2 \wedge stockPro_{-2} > -1 \wedge \\
stockE_{-2} > -1 \wedge stockE_{-2} < 3 \wedge stockPro_{-1} = stockPro_{-2} - 1 \wedge \\
stockE_{-1} = stockE_{-2} \wedge stockPro_{-1} = stockPro_0 \wedge stockPro_{-1} > -1 \wedge \\
stockE_{-1} > -1 \wedge stockE_{-1} < 3 \wedge stockPro_0 > -1 \wedge StockE_0 > -1 \wedge \\
stockE_0 < 3 \wedge stockE_1 = stockE_0 + 1 \wedge stockPro_1 = stockPro_0 \wedge \\
stockPro_0 > -1 \wedge stockE_0 > -1 \wedge stockE_0 < 3 \wedge \\
stockPro_1 > -1 \wedge stockE_1 > -1 \wedge stockE_1 < 3 \\
\rightarrow \\
stockE_1 = stockE_{-2} + 1
\end{array}
$$

The variables graph associated to this proof obligation is shown Fig. 6.

The lack of lock corresponds to a variables graph where the path between several variables arising in the goal of a proof obligation could be completed by an edge between nodes corresponding to the same variables in two states corresponding to the sates just before and just after a *SeqAnd* or a *SeqOr* operator. The solution to correct the bug consists in changing the operator (*SeqAnd* or *SeqOr* in its synchronised version (*SyncSeqAnd* or *SyncSeqOr*) with a lock on the concerned variable.

Lack of Preservation In some cases, the developer may forget to express that a goal execution must no modify a variable. Let consider the producer agent of our case study once again, and suppose that its main goal is this time well decomposed thanks to a *SyncSeqAnd* operator, but with the satisfaction

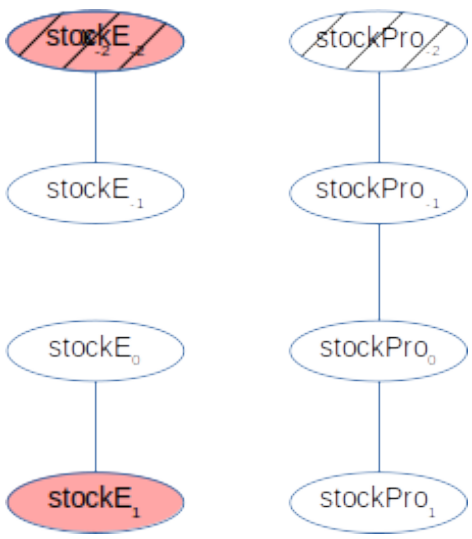

Fig. 6. Lack of lock.

condition of its first subgoal being : $stockPro' = stockPro - 1$. We obtain the following proof obligation:

$$
\begin{array}{l}
stockPro_{-2} > 0 \wedge stockE_{-2} < 2 \wedge stockPro_{-2} > -1 \wedge \\
stockE_{-2} > -1 \wedge stockE_{-2} < 3 \wedge stockPro_{-1} = stockPro_{-2} - 1 \wedge \\
stockPro_{-1} = stockPro_0 \wedge stockPro_{-1} > -1 \wedge stockE_0 = stockE_{-2} \wedge \\
stockE_{-1} > -1 \wedge stockE_{-1} < 3 \wedge stockPro_0 > -1 \wedge stockE_0 > -1 \wedge \\
stockE_0 < 3 \wedge stockE_1 = stockE_0 + 1 \wedge stockPro_1 = stockPro_0 \wedge \\
stockPro_0 > -1 \wedge stockE_0 > -1 \wedge stockE_0 < 3 \wedge \\
stockPro_1 > -1 \wedge stockE_1 > -1 \wedge stockE_1 < 3 \\
\rightarrow \\
stockE_1 = stockE_{-2} + 1
\end{array}
$$

The variables graph generated from this proof obligation is shown Fig. 7.

As in the previous case, this kind of bug is characterised by a hole in the chain between to variables used in the goal of a proof obligation. But in this case, the edge that lacks is not at the same place : it is between two states corresponding to the first one and the last one of a goal. By adding an equality between the values of the concerned variables in these two states, the proof obligation can be proved.

Uncompleted Behaviour This kind of bug is the consequence of an erroneous specification, where an agent does not do all what it should. Let's consider now our case study where the producer agent does not increase the stock in the environment. Thus, the satisfaction condition of its second subgoal is

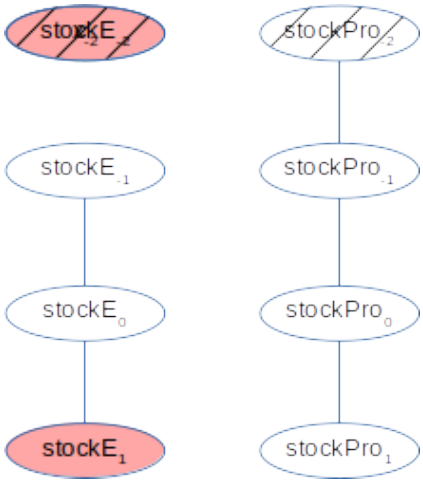

Fig. 7. Lack of Preservation Condition

$stockPro' = stockPro$ instead of $stockPro' = stockPro \wedge stockE' = stockE + 1$. The proof obligation is then the following:

$$
\begin{array}{l}
stockPro_{-2} > 0 \wedge StockE_{-2} < 2 \wedge StockPro_{-2} > -1 \wedge \\
stockE_{-2} > -1 \wedge StockE_{-2} < 3 \wedge stockPro_{-1} = stockPro_{-2} - 1 \wedge \\
stockE_{-1} = stockE_{-2} \wedge stockPro_{-1} = stockPro_0 \wedge stockPro_{-1} > -1 \wedge \\
stockE_0 = stockE_{-2} \wedge stockE_{-1} > -1 \wedge stockE_{-1} < 3 \wedge stockPro_0 > -1 \wedge \\
stockE_0 > -1 \wedge stockE_0 < 3 \wedge stockPro_1 = stockPro_0 \wedge \\
stockPro_0 > -1 \wedge stockE_0 > -1 \wedge stockE_0 < 3 \wedge \\
stockPro_1 > -1 \wedge stockE_1 > -1 \wedge stockE_1 < 3 \\
\rightarrow \\
stockE_1 = stockE_{-2} + 1
\end{array}
$$

The variables graph generated from the previous proof obligation is the same as the graph in Fig. 7. However, in this case, adding a preservation property $stockE' = stockE$ to the satisfaction condition of the second subgoal of the main goal does not correct the proof obligation (the prover still fails to prove the formula).

Under-Specified Triggering Context When the triggering context of an agent is under-specified, a proof obligation may be wrong. Let's consider our producer agent once more. The proof schema associated to the first subgoal requires to prove the invariant, and so, we have to prove that $stockPro$ remains greater or equal to zero. If the triggering context of the agent is under-specified

(for example, it is just *true*), this leads to the following proof obligation:

$$stockPro_0 > -1 \wedge stockE_0 > -1 \wedge$$
$$stockPro_1 = stockPro_0 - 1 \wedge stockE_1 = stockE_0$$
$$\rightarrow$$
$$stockPro_1 > -1$$

The variables graph generated from this proof obligation is represented Fig. 8. As we can see on this figure, this case of bug correspond to a graph without missing edge. So, it can be easily distinguished from the previous ones.

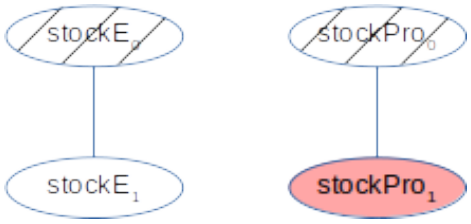

Fig. 8. Under-Specified Triggering Context

Wrong Satisfaction Condition Another kind of bug corresponds to a wrong satisfaction condition. On the producer agent of the case study, it would be the case if the satisfaction condition of the first subgoal was $stockPro' = stockPro - 2 \wedge stockE' = stockE$, for instance. In this case as in the previous one, no problem can be detected on the variables graph.

Wrong Decomposition Operator Finally, the last kind of bug we have identified corresponds to a bad choice of the decomposition operator. This would be the case if, for the producer agent of the case study, we had used a *SyncSeqOr* operator instead of a *SyncSeqAnd*. Once again, with such a kind of bug, no problem can be detected from the variables graph.

6 Variables Graph Usage: Formalisation and Generalisation

In the previous section, we analysed variables graph on a case study. In the section, we give more general rule to use such graphs. However, we must precise that the work presented here, consider only proof obligations that are horn clauses, that is to say that look like:

$$h_1 \wedge h_2 \wedge ... \wedge h_n \rightarrow g$$

with no *or* logical operator in every hypothesis h_i, and neither *or* nor *and* in the goal g.

Definition 2 (variables graph associated to a proof obligation). *a variables graph for such a proof obligation is an undirected graph:*

$$VG = (V, E, V_{init}, V_{goal})$$

where V is the set of vertices and E is the set of edges. This graph is built as follow:

- *V corresponds to the set of each sub-scripted variable used in any hypothesis or in the goal of the proof obligation ;*
- *$V_{init} \subseteq V$ is the set of sub-scripted variables corresponding to the state in which the agent began the execution of its behaviour ;*
- *$V_{goal} \subseteq V$ is the set of sub-scripted variables used ind the goal of the proof obligation ;*
- *$(v_a, v_b) \in E$ if and only if there is at least one hypothesis h_i that uses both v_a and v_b.*

We can now formally describe several kinds of bugs when a proof obligation is not proven and when vertices in V_{goal} correspond to a same variable v in different states.

If two vertices v_{t_a} and v_{t_b} in V_{goal} corresponding to the same variable v in different states t_a and t_b ($t_a < t_b$) do not belong to the same connected component of the graph, we determine t_{a_2}, the last state for which v is present in the connected component of v_{t_a} and t_{b_1}, the first state for which v is present in the connected component of v_{t_b}.

- if t_{a_2} and t_{b_2} correspond respectively to the initial state and to the final state of a goal and adding an hypothesis $v_{t_{a_2}} = v_{t_{b_1}}$ make the proof feasible, the problem is a lack of preservation;
- if t_{a_2} and t_{b_2} correspond respectively to the initial state and to the final state of a goal g' and adding an hypothesis $v_{t_{a_2}} = v_{t_{b_1}}$ does not make the proof feasible, the problem might be an under-specified satisfaction condition for the goal g';
- if t_{a_2} and t_{b_2} correspond respectively to the final state and to the initial state of two subgoals g_1 and g_2 connected by a SEQAND operator or a SEQOR operator, and adding an hypothesis $v_{t_{a_2}} = v_{t_{b_1}}$ make the proof feasible, the problem is a concurrency problem;

If the vertices in V_{goal} correspond to the same variable v in different states t_a and t_b ($t_a < t_b$) and if they belong to the same connected component of the graph, then we have to consider three potential causes: under-specified triggering context, wrong satisfaction condition, of wrong decomposition operator. Variables graph do not give anymore information.

7 Conclusion and Perspectives

In this article, we have presented a promising way to identify proof failures explanations, associated to a guide to their correction. More precisely, we have determined six different kinds of errors, and we have shown that three of these kinds correspond to different characteristics on variables graph. We are now working on an implementation of an algorithm that aims to automate the correction of these three first kinds of bugs when it is possible. Using a very strongly structured specification, as GDT4MAS clearly appeared as a key characteristics to have better chances to correct mistakes. This could explain the fact that there was no other preliminary work in this field, and that the work presented here, although offering only partial results for the moment, seems to open the way for future much more complete results

In the near future, we plan to work on extending the usage of variables graph when the goal of a proof obligation mentions different variables.

Finally, we plan to use automatic test case generation to help to distinguish cases that variables graph cannot isolate.

References

1. O. Coelho, U. Kulesza, A. von Staa, and C. Lucena. Unit testing in multi-agent systems using mock agents and aspects. In *SELMAS 06*, pages 83–90, 2006.
2. M. Dastani and J.-J. Ch Meyer. *Specification and Verification of Multi-agent Systems*, chapter Correctness of Multi-Agent Programs: A Hybrid Approach. Springer, 2010.
3. L. A. Dennis and P. Nogueira. What can be learned from failed proofs of non-theorems. Technical report, Oxford University Computer Laboratory, 2005.
4. A. Drogoul, N. Ferrand, and J.-P. Müller. Emergence : l'articulation du local au global. *ARAGO*, 29:105–135, 2004.
5. K. Suzanne Barber Dung N. Lam. Automated Interpretation of Agent Behaviour. In *AOIS*, pages 1–15, 2005.
6. J. J. Gómez-Sanz, J. A. Botía Blaya, E. Serrano, and J. Pavón. Testing and Debugging of MAS Interactions with INGENIAS. In *AOSE*, pages 199–212, 2008.
7. David Hauzar, Claude Marché, and Yannick Moy. Counterexamples from proof failures in SPARK. In Rocco De Nicola and eva Kühn, editors, *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-8, 2016, Proceedings*, volume 9763 of *Lecture Notes in Computer Science*, pages 215–233. Springer, 2016.
8. M. Kaufmann and J.S. Moore. Proof Search Debugging Tools in ACL2. http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html, 2008.
9. Sebastian Krings, Jens Bendisposto, and Michael Leuschel. From failure to proof: The prob disprover for B and event-b. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, volume 9276 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2015.
10. Dung N. Lam and K. Suzanne Barber. Comprehending agent software. In *AAMAS*, pages 586–593, 2005.

11. Xue Li, Alan Bundy, and Alan Smaill. ABC repair system for datalog-like theories. In David Aveiro, Jan L. G. Dietz, and Joaquim Filipe, editors, *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2018, Volume 2: KEOD, Seville, Spain, September 18-20, 2018*, pages 333–340. SciTePress, 2018.
12. 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.
13. B. Mermet and G. Simon. A new proof system to verify gdt agents. In *IDC*, pages 181–187, 2013.
14. Bruno Mermet and Gaële Simon. Using proof failures to help debugging MAS. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019*, pages 523–530. SciTePress, 2019.
15. D. Meron and B. Mermet. A Tool Architecture to Verify Properties of Multiagent System at Runtime. In *PROMAS*, pages 201–216, 2006.
16. S. Miles, M. Winikoff, S. Cranefield, C.D. Nguyen, A. Perini, P. Tonella, M. Harman, and M. Luck. Why testing autonomous agents is hard and what can be done about it. AOSE Technical Forum 2010 Working Paper.
17. C. D. Nguyen, S. Miles, A. Perini, P. Tonella, M. Harman, and M. Luck. Evolutionary testing of autonomous software agents. *JAAMAS*, 25(2):260–283, 2012.
18. C.D. Nguyen, A. Perini, C. Bernon, J. Pavón, and J. Thangarajah. Testing in Multi-Agent Systems. In *AOSE*, pages 180–190, 2009.
19. C.D. Nguyen, A. Perini, and P. Tonella. Ontology-based test generation for multiagent systems. In *AAMAS*, pages 1315–1320, 2008.
20. Cu D. Nguyen, Anna Perini, and Paolo Tonella. Goal-oriented testing for MASs. *IJAOSE*, 4(1):79–109, 2010.
21. S. Owre, N. Shankar, and J. Rushby. Pvs: A prototype verification system. In *CADE 11*, 1992.
22. Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti, and Jacques Julliand. How testing helps to diagnose proof failures. *Formal Aspects Comput.*, 30(6):629–657, 2018.
23. Guillaume Petiot, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. How test generation helps software specification and deductive verification in frama-c. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP@STAF 2014, York, UK, July 24-25, 2014. Proceedings*, volume 8570 of *Lecture Notes in Computer Science*, pages 204–211. Springer, 2014.
24. E. Serrano, J.J. Gómez-Sanz, J.A. Botía, and J. Pavón. ntelligent data analysis applied to debug complex software systems. *Neurocomputing*, 72(13-15):2785–2795, 2009.
25. A.M. Tiryaki, S. Öztuna, O. Dikenelli, and R.C. Erdur. SUNIT: A Unit Testing Framework for Test Driven Development of Multi-Agent Systems. In *Agent Oriented Software Engineering (AOSE)*, pages 156–173, 2006.
26. G. Vigueras and J.A. Botía. Tracking Causality by Visualization of Multi-Agent Interactions Using Causality Graphs. In *PROMAS*, pages 190–204, 2007.
27. Zhiyong Zhang, John Thangarajah, and Lin Padgham. Model based testing for agent systems. In *AAMAS'09*, pages 1333–1334, 2009.