Using proof failures to help debugging MAS

Bruno Mermet, Gaële Simon

Using proof failures to help debugging MAS

Bruno Mermet[1] and Gaële Simon[1]
[1]*Normandie Univ., Université Le Havre Normandie, CNRS, Greyc, 1400 Caen, France*
{Bruno.Mermet, Gaele.Simon}@unicaen.fr

Keywords:
Multi-Agent Systems, Proof Failure, Debugging

Abstract:
For several years, we have worked on the usage of theorem proving techniques to validate Multi-Agent Systems. In this article, we present a preliminary case study, that is part of larger work whose long-term goal is to determine how proof tools can be used to help to develop error-free Multi-Agent Systems. This article describes how an error caused by a synchronisation problem between several agents can be identified by a proof failure. We also show that analysing proof failures can help to find bugs that may occur only in a very particular context, which makes it difficult to analyse by standard debugging techniques.

1 Introduction

This article takes place in the general context of the validation of Multi-Agents Systems, and more specifically in the tuning stage. Indeed, for several years now, we have worked on the validation of MAS thanks to proof techniques. This is why the designed the GDT4MAS model (Mermet and Simon, 2009) has been designed, which provides both formal tools to specifiy Multi-Agent Systems and a proof system that generates automatically, from a formal specification, a set of *Proof Obligations* that must be proven to guarantee the correctness of the system.

In the same time, we have begun to study how to answer to the following question: "What happens if the theorem prover does not manage to carry out the proof ?". More precisely, is it possible to learn anything from this failures (that we call in the sequel *proof failures*), in order to debug the MAS ? Answering to this question in a general context is tricky. Indeed, a first remark is that a proof failure may occur in three different cases:

- first case: a true theorem is not provable (Gödel Incompleteness Theorem);
- second case: a true theorem can not be automatically proven by the prover because first-ordre logic is semidecidable;
- third case: an error in the MAS specification has led to generate a false theorem that, hence, cannot be proven.

So, when a proof failure is considered, the first problem is to determine the case it corresponds to. It would be rather long and off-topic to give complete explanations here. However, it is important to knwow that the proof system has been designed to generate theorems that have good chances to be proven by standard strategies of provers, without requiring the expertise of a human. Moreover, unprovable true theorems generally do not correspond to real cases. Thus, in most cases, a proof failure will correspond to a mistake in the specification, and this is the context that is considered in the sequel.

The subject of our study is then the following: if some generated proof obligations are note proven automatically, can we learn from that in order to help to correct the specification of the MAS ? So, the main idea is to check wether proof failures can be used to detect, even correct bugs in the specification of the MAS.

Indeed, contrary to what is presented in (Dastani and Meyer, 2010), where authors consider that proof-based approaches are dedicated to MAS validation and that other approaches must be considered for debugging and tuning, we aim at using proof failures to capture mistakes very early in the design of a MAS and to help to correct them.

In this article, we begin by a brief presentation of existing works dealing with the debugging of MAS. In part 3, we present the GDT4MAS model, the proof mechanism, and the associated tools. Section 4 presents the core of our work. In the last part, we conclude on the work presented here and we present the future or our research in this domain.

2 State of the art

Ensuring the correctness of a MAS is a crucial issue, but this is a very hard problem, as it has been established several times (Drogoul et al., 2004). As for classical software, there are mainly two kinds of methods to check the correctness of a MAS: proof and test. If interested, the reader is encouraged to refer to previous articles (Mermet and Simon, 2009; Mermet and Simon, 2013) for a more comprehensive state of the art on the subject. In this section, we focus on works dealing with the tuning of erroneous systems.

2.1 Test

Most works about the debugging of MAS deal with test used to discover a potential problem, and they also deal with how to provide one or more test cases allowing to reproduce the problem. Tests can be situated at different levels, as exposed in (Nguyen et al., 2009):

- at the unit level: this is for instance the goal of the work presented in (Zhang et al., 2009).
- at the agent level: there are numerous works at this level. Some propose "xUnit" tools to specify unit tests (Tiryaki et al., 2006), whereas others propose to add to the system agents dedicated to test (Nguyen et al., 2008)
- at the MAS level: principles of test at the MAS level are not yet many. One of the reasons is certainly the difficulty of the problem, well detailed in (Miles et al.,). But there are however a few works in this domain (Tiryaki et al., 2006; Nguyen et al., 2010).

2.2 Trace analysis

Another kind of works dealing with the debugging of MAS relies on the trace analysis. These works mainly focus on three of the tasks associated to debugging: discovering the problem, identifying the potential causes and finding the true one. (Lam and Barber, 2005; Vigueras and Botía, 2007; Serrano et al., 2009).

Traces are analysed using two different methods: either they study the ordering of messages exchanged between agents, or they use knowledge provided by the designer of the system using data mining techniques to check if this knowledge can be found in the MAS trace, in order to discover bugs and to explain them. The combination of these two techniques is for example studied in (Dung N. Lam, 2005).

2.3 Visualization

A few work deal with visualization tools for MAS, although this kind of tools may be interesting. But designing relevant views is a very hard problem because of the potential huge number of entities interacting. Some works propose to generate views from traces (Vigueras and Botía, 2007).

2.4 Proof failures

Using proof failure is a prospective domain, that has not yet been examined in depth. Some works propose to provide the prover with tools that might use proof failures (Kaufmann and Moore, 2008). About the usage of these proof failures, a few ideas are proposed in (Dennis and Nogueira, 2005), but they have not been implemented.

3 The GDT4MAS model

This approach, that integrates a model and an associated proof system, offers several interesting characteristics for the design of MAS: a formal language to describe the behaviour of agents and the expected properties of the system, the usage of the well-known and expressive first-order logic, and an automatisable proof process.

We briefly present here the GDTM4MAS method, more detailed in (Mermet and Simon, 2009; Mermet and Simon, 2013).

3.1 Main concepts

The GDT4MAS model requires to specify several concepts described here.

Environment The environment of the MAS is specified 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 is a tree of goals, whose root correspond to the main goal of the agent. A plan is associated to each goal: when this plan is executed with success, the goal it is associated to (called *parent goal*) is achieved. A plan can be made either of a single action, or a set of goals (called *subgoals*) linked by a *decomposition operator.* A goal G is mainly described by its name n_G, a *satisfaction condition* sc_G and a *guaranted 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 succeeds. Otherwise, the guaranted property in case of failure (GPF) of the goal specifies what is however guaranted to be true when the execution of the plan associated to the goal fails (It is of course not considered when the goal is said to be a *NS goal*, that is to say a goal whose plan always succeeds).

SC and GPF are called *state transition formulae* (STF), because they establish a link between two states, called the *initial state* and the *final state*, corresponding to the state of the system just before the agent tries to solve the goal and the state of the system when the agent has just ended the execution of the plan associated to the goal. In an STF, a given variable v can be primed or not. The primed notation (v') represents the value of the variable in the final state whereas a non-prime notation (v) represents the value of the variable in the initial state. A STF can be non deterministic when, considering a given initial state, several final states can satisfy it. This is for instance the case of the following STF : $x' > x$. This means that the value of variable x must be greater after the execution of the plan associated to the goal than before. For instance, if the value of x is 0 in the initial state, final states with a value of 2 or 10 for x would satisfy this STF.

Decomposition operators GDT4MAS proposes several decomposition operators, in order to specify several types of behaviours. In this article, we only use two of them:

- the `SeqAnd` operator specifies that subgoals must be executed in the given order (from the left to the right on the graphical representation of the GDT). If the behaviour of the agent is sound, achieving all of the subgoals achieves the parent goal. If the execution of the first subgoal fails, the second subgoal is not executed;
- the `SyncSeqAnd` operator works similarly to the SeqAnd operator, but it offers the possibily to lock a set of variables in the environment during the execution of the plan. Thus, other agents won't be able to modify these variables as long as the execution of the plan is not finished.

Actions Actions are specified by a precondition, specifying in which states it can be executed, and a postcondition, specifying by an STF the effect of the action.

Agents Agents are defined as instances of agent types, with specific initialisation values for the variables of the type.

3.2 GDT example

Figure 1 shows the GDT of an agent made of three goals (represented by ellipses with their name and their SC): goal A is the root goal. Thanks to the `SeqAnd` operator, it is decomposed into 2 subgoals B and C. These goals are leaf goals and so, an action (represented by an arrow) is associated to each of them.

3.3 Proof principles

3.3.1 General Presentation

The proof mechanism aims at proving the following properties:

- Agents preserve their invariant properties (Mermet and Simon, 2013);
- Agents preserve the invariant properties of the environment;
- Agents behaviours are consistent; (plans associated to goals are correct);
- Agents respect their liveness properties. These properties formalize expected dynamic characteristics.

Moreover, the proof mechanism relies on *proof obligations* (PO). POs are properties that must be proven to guarantee the correctness of the system. They 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. Finally, the proof system is compositional: The proof of the correctness of a given agent type is decomposed into several small independant proof obligations, and most of the time, the proof of a given agent type can be performed independently of the others.

3.3.2 Proof schema

The GDT4MAS method defines several *proof schemas*. These proof schemas are formulae that are used to generate *proof obligations*.

3.3.3 PVS

PVS (Prototype Verification System) (Owre et al., 1992) is a proof environment relying mainly on a theorem prover that can manage specifications expressed in a typed higher order logic. The prover uses a set of predefined theories dealing, among others, with set theory and arithmetic. The prover can work in an interactive way: indeed, the user can interfer in the proof process.

For our part, we want to minimize the user intervention. It is the reason why Proof Obligations are generated so as to maximize the success rate of automatic proof strategies. The strategy of PVS we use is a very general one called *grind* that uses, among others, propositional simplification, arithmetic simplification, skolemisation ad the disjunctive simplification.

3.4 Execution platform

In order to carry out many experiments on the GDT4MAS model, we are developing a platform with 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 is developed mainly in Java. Specifications are written in XML and can be executed in several modes. Among them, there is a *random* mode (agents are activated at random) and a *trace* mode (agents are activated in a predefined order). During the execution, dynamic charts show in real time the values of selected agent variables. Moreover, a log console gives information on the system activity (activated agent, goal being executed, action executed, etc.).

4 Using Proof Failures

The problem this article deals about is the following. Let suppose we have a GDT4MAS specification of a Multi-Agent System. Using the proof system associated to the model, we can generate a set of Poof Obligations that must be proven to guarantee that the specification is correct. But if at least one of these proof obligations cannot be proven, it may indicate a bug in the specification, than might be detected during a particular execution of the MAS. So, can the information provided by the proof failure (in particular by the prover) be used, at least by hand, to identify the error in the specification ?

There are several kinds of such errors. Here are a few examples:

- the decomposition operator used to define the plan associated to a goal is not the good one, so achieving subgoals does not achieve the parent goal;
- the Satisfaction Condition associated to a goal is too weak, and so does not provide properties required later;
- The *triggering context* of the agent is wrong, so the agent can be activated when it must not, or can not be activated when it would be necessary;
- the invariant associated to an agent time is wrong or too weak.

In the long term, our goal is to identify several error categories for which proof failures are similar and can be used to help to identify the error in the specification that has generated the proof failure. In order to adress this problem, we have specified a case-study based on the producer/consumer system. In this example, we have introduced two kinds of errors, and we have studied the proof failures that they have generated, in order to determine if it was possible, to identify the errors starting from the proof failures. This study is presented in the sequel.

4.1 Case study

4.1.1 Description of the MAS used

As written before, our case study relies on a producer-consumer system. In the basic version, the system is made of 2 agents, with two dedicated agent types: the Producer type and the Consumer type. For the proof point of view, the analysis of the system is the same whatever the number of agents of each type is, even if from the point of view of the execution, we will see that there are differences. The environment of the MAS includes a variable named *stockE* (an integer). This variable corresponds to the number of resources (pounds of floor for instance) that the producer has already put in the environment.

To produce these resources, the producer uses internal resources of an other type (wheat for instance), represented by an internal variable *stockPro* whose value represents the amount of wheat (in pounds) that the producer owns. To produce one resource of the environment (one pound of floor), the producer consumes one of its own resources (one pound of wheat). This process is represented by the GDT of the Producer type (shown in figure 1) made of three goals: goal B models the consumption of the internal resource and goal C models the production of the new resource in the environment.

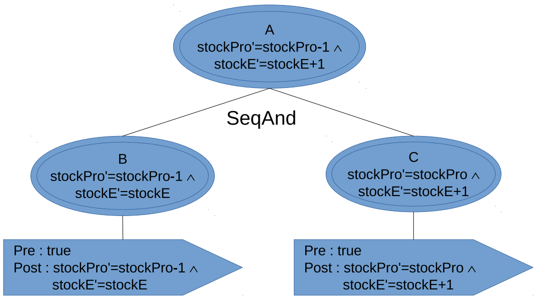

Figure 1: GDT of the Producer type

For its part, the consumer produces resources of a third type (bags of bread, for instance). The number of resources produced is represented by the value of an internal variable of the Consumer type called *stockCons*. To produce this kind of resources, the producer needs to use 2 resources of the environment (producing a bag of bread requires 2 pounds of floor). This process is formalized by the GDT of the Consumer type (see figure 2): goal B models the consumption of 2 resources of the environment and goal C models the production of a new resource of the third type.

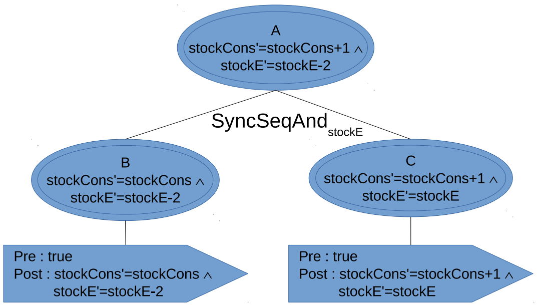

Figure 2: GDT of the Consumer type

In addition to the process described above, there is an additional constraint: only two resources can be stocked in the environment (in our example, there is a place for only two pounds of floor on the shelf). It means that if there are already 2 resources in the environment, one of them should be consumed before the producer can put a new one in the environment. This constraint for the producer is formalized by its *triggering context*: $stockPro > 0 \wedge stockE < 2$. With such a triggering context, the producer can be activated only if it still has resources and if it can store its production in the environment. Moreover the following invariant is associated to the environment: $stockE > -1 \wedge stocke < 3$. This invariant determines the set of legal values for the *stockE* variable, according to the constraint presented above. The following invariant is associated to the Producer type: $stockPro > -1$. Indeed, the number of internal resources cannot be negative.

The triggering context of the Consumer type is simpler: $stockE > 1$. In other words, the consumer cannot act if there is not at least 2 resources available in the environment. The invariant associated to this agent type is similar to the invariant of the Producer: it specifies that the number of its internal resources is a natural number: $stockCons > -1$.

4.1.2 Proof failure analysis

When proof obligations are generated from the case study presented above, a PVS theory made of 18 theorems is produced, where each theorem corresponds to a proof obligation that must be proven to ensure that the specifications of both agent types are correct. These proof obligations cannot be listed here, but we can recall that, if proven, they guarantee that:

- each goal decomposition is correct (achieving the plan associated to a goal achieves this goal);
- for each leaf goal:
  - The execution context of the goal implies the precondition of the action associated to the goal;
  - The postcondition of the action associated to the goal implies the satisfaction condition of the goal;
  - the environment invariant and the agent invariant are preserved by the execution of the action associated to the goal.

When PVS tries to prove the theory generated from the GDT4MAS specification, there are 2 proof failures (for two theorems, PVS says that the proof is “unfinished”). In interactive mode, PVS provides to the user the last sequent of the proof branch that it cannot verify. The two unproved theorems are the following:

- `SCproducerTypeA` : this proof obligation is required to prove that the decomposition of goal A is correct, that it is to say that the success of the execution of the plan associated to A achieves A.

- `PostInvProducerTypeC` : this proof obligation is required to verify that the execution of the action associated to goal C preserves the invariants of the agent and of the environment.

We introduce now a few notations concerning the variables that are used in the PVS specification. These notations are used to represent the value of a variable in several states of the agent.

- v_{-2} (v_2) : value of v just before B;
- v_{-1} (v_1) : value of v just after B;
- v_0 (v0) : value of v just before C;
- v_1 (v1) : value of v just after C.

For instance, in theorem `SCproducerTypeA`, $stockE'$ is represented by $stockE1$ and $stockE$ is represented by $stockE\_2$. Indeed, in our execution model, the state *just before A* is considered to be the same state as the state *just before B*.

First Proof failure The theorem `SCtproducerTypeA` generated by the plateform is the following:

```
SCproducerTypeA: THEOREM
((true)&(stockPro_2>0)&(stockE_2<2)&(stockPro_2>-1)&
(stockE_2>-1)&(stockE_2<3)&(stockPro_1 = stockPro_2-1)&
(stockE_1 = stockE_2)&(stockPro_1 = stockPro0)&
(stockPro_1>-1)&(stockE_1>-1)&(stockE_1<3)&(stockPro0>-1)&
(stockE0>-1)&(stockE0<3)&(stockE1 = stockE0+1)&
(stockPro1 = stockPro0)&(stockPro1>-1)&
(stockE1>-1)&(stockE1<3))
=>
(stockPro1 = stockPro_2 - 1)&(stockE1 = stockE_2 + 1)
```

We can notice that the right-hand side of the “implies” corresponds to the satisfaction condition of goal A with the states introduced above: `stockE1` corresponds to à `stockE'`, `stockE_2` corresponds to `stockE`, `stockPro1` corresponds to `stockPro'` and `stockPro_2` corresponds to `stockPro`.

The sequent that PVS cannot prove when it tries to prove this theorem is the following:

```
{-1}  (stockPro_2 > 0)
{-2}  (stockE_2 < 2)
{-3}  (stockPro_2 > -1)
{-4}  (stockE_2 < 3)
{-5}  (stockPro_1 = stockPro0)
{-6}  (stockE_1 = stockE_2)
{-7}  (stockPro_2 - 1 = stockPro0)
{-8}  (stockE_2 > -1)
{-9}  (stockE0 > -1)
{-10} (stockE0 < 3)
{-11} (stockE1 = 1 + stockE0)
{-12} (stockPro1 = stockPro0)
{-13} (stockPro0 > -1)
{-14} (1 + stockE0 > -1)
{-15} (1 + stockE0 < 3)
  |-------
{1}   stockE0 = stockE_2
```

The first thing we can notice is that the predicate with the *stockPro* variable that was part of the right-hand side of the “implies” in the initial theorem is missing in the consequent of the sequent. This means that this part of the theorem has been proven and that the problem only concerns the part of the satifaction condition of goal A dealing with *stockE*. Thus, we can conclude that the prover cannot prove that $stockE0 = stockE\_2$ with the hypotheses in the left-hand side of the theorem.

Indeed, in the hypotheses of the theorems generated by our proof systems, relations between variables concern either variables in the same state (this is for instance the case for invariant properties) or variables in two consecutive states. So, if the sequent above cannot be proven, this ought to be because at least one hypothesis among $stockE\_2 = stockE\_1$ or $stockE\_1 = stockE0$ is missing. But we can observe that the second one is present in the hypotheses of the sequent (number $\{-6\}$). So, the only missing predicate is $stockE\_1 = stockE0$, meaning that the value of *stockE* is not modified between the end of the execution of goal B and the beginning of the execution of goal C. Indeed, between these two states, another agent in the system might modify the value of variable *stockE*. The only way to prevent this is to lock the variable, using a synchronized operator (`SyncSeqAnd` in our case). With such a lock, no other agent can modify *stockE* during the execution of the decomposition of goal A. As *stockE* is an environment variable, at any time, any agent can modify it (this is not the case for agent variables, that can only be modified by the owner agent). So, in the GDT of the Producer type, the `SeqAnd` operator must be replaced by a `SyncSeqAnd` operator locking variable *stockE*.

Second proof failure The theorem `PostinvtypeProducteurC` generated by our plateform is the following:

```
((true)&(stockPro_2>0)&(stockE_2<2)&(stockPro_2>-1)&
(stockE_2>-1)&(stockE_2<3)&(stockPro_1 = stockPro_2-1)&
(stockE_1 = stockE_2)&(stockPro_1 = stockPro0)&
(stockPro_1>-1)&(stockE_1>-1)&(stockE_1<3)&(stockPro0>-1)&
(stockE0 -1)&(stockE0<3)&(stockE1 = stockE0+1)&
(stockPro1 = stockPro0))
=>
(stockPro1 > -1)&((stockE1 > -1)&(stockE1 < 3))
```

This theorem aims at demonstrating that the invariant of the agent is preserved by the execution of the action associated to goal C: this explains the part $stockPro1 > -1$ in the right-hand side of the "implication". It also aims at demonstrating that the invariant predicate of the environment is preserved by the execution of this action. This explains the part $stockE1 > -1 \wedge stockE1 < 3$.

The sequent that PVS cannot prove is the following:

```
{-1}  (stockPro_2 > 0)
{-2}  (stockE_2 < 2)
{-3}  (stockPro_2 > -1)
{-4}  (stockE_2 < 3)
{-5}  (stockPro_1 = stockPro0)
{-6}  (stockE_1 = stockE_2)
{-7}  (stockPro_2 - 1 = stockPro0)
{-8}  (stockPro0 > -1)
{-9}  (stockE_2 > -1)
{-10} (stockE0 > -1)
{-11} (stockE0 < 3)
{-12} (stockE1 = 1 + stockE0)
{-13} (stockPro1 = stockPro0)
   |-------
{1}   (1 + stockE0 < 3)
```

When we compare it with the initial theorem, it appears that the prover cannot prove a part of the environment invariant, more precisely the fact that the value of $stockE$ is always less than 3 because hypothesis $stockE\_1 = stockE0$ is missing. This is the same problem that we encountered for the first proof failure. It means that this proof failure is also a consequence of not having used a synchronized operator locking variable $stockE$.

So, this proof failure has allowed us to detect the same design error that the first proof failure. However, it does not correspond to the same bug. Indeed, the fact that the prover cannot verify that $stockE0 < 2$ shows that it is not guaranted that the value of $stockE$ is less than 2 before the execution of C. As a consequence, there are situations where, after the execution of C, the value of $stockE$ is greater than 2, which is forbidden by the environment invariant. This corresponds to a new bug that may occur during the system execution. An interesting property with this proof failure is that the bug cannot occur in a system with a single producer in the system. But if we consider a MAS with several producers, the problem will probably occur. For instance, consider a system with 2 producers p1 and p2, each having a $stockPro$ variable initialized to 5, an environment with the variable $stockE$ initialized to 0, and a single consumer whose variable $stockCons$ is initialized to 0. Now, supppose that the execution of the MAS leads to the following interleaving of agents p1 and p2:

```
...
p1 (stockPro of p1 takes the value of 4)
p1 (stockE takes the value of 1)
p1 (stockPro of p1 takes the value of 3)
p2 (stockPro of p2 takes the value of 4)
p2 (stockE takes the value of 2)
p1 (stockE takes the value of 3)
...
```

The execution trace above leads to the bug pointed out by the proof failure. This trace has indeed been executed by our plateform and, thanks to the visualisation of the variables of the system, we have observed the bug: the value of $stockE$ indeed reaches the value of 3 during this execution.

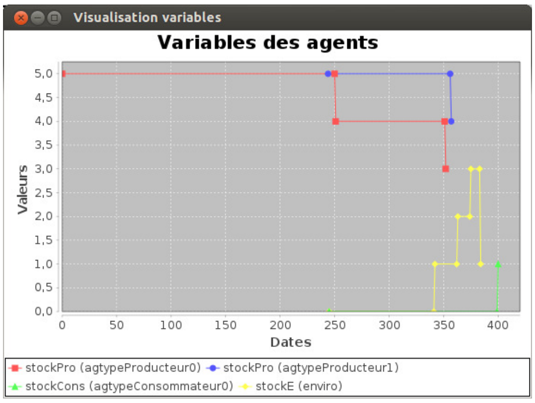

Figure 3: Values of the variables in the MAS over the time

Review of the proof failures analysis After the anlysis of these 2 proof failures, we can, at first, notice that the structure of the proof obligations generated by our proof system provides a first way to rapidly identify which part of the system behaviour is involved in the problem (the name of the generated theorem helps in this task). We have also shown that the 2 studied proof failures help to identify a design error in the specification of the Producer type. The first proof failure has helped to find an hypothesis that was lacking in the theorem, and that led us to find how to fix the problem by using a synchronized operator in order to lock variable *stockE* during the execution of the decomposition of goal A. This problem is a consequence of a synchronisation problem between the agents (producer/consumer in the first case, producer/producer in the second case), identified thanks to the proof failures.

Moreover, each of these proof failures pointed out 2 different bugs (Satisfaction condition of goal A not established in some cases, non compliance with the bounds of *stockE* in other cases) generated by the same error in the design. We can notice that that they would have been difficult to identify without our proof system. Namely, the bug associated to the second proof failure cannot occur in a system with less than two producers, and even in a system with 2 producers or more, the bug can be observed only during specific executions. So, using a proof system leads to an important time saving in bug identification.

We can notice that, if we modify the specification of the Producer agent type using a `SyncSeqAnd` operator instead of a `SeqAnd` operator to decompose goal A, the proof of the system is now performed successfully by PVS.

This small case study shows that proof failures can be used to detect undesired behaviours during the execution (bugs associated to each proof failure) and to determine errors in the specification linked to these bugs (here, the lack of lock on an environment variable by the producer agent).

5 Conclusion and perspectives

In this article, we have presented a promising way to use proof failures in the tuning of MAS. In particular, we have shown that such a technique highlights bugs that appears in few executions, because they can depend on the interleaving of the actions of the agents. That makes these bugs hard to detect and to correct with standard debugging techniques because they are hard to reproduce. Of course, research must continue with other kinds of proof failures to validate the technique in a more general way. We also aim at developing a semi-automatic usage of proof failures, because it seems that standard patterns of proof failure emerge. In the longer term, we should be able to propose a taxonomy of proof failures, associating to each kind of proof failure the potential causes and the potential required patches.

REFERENCES

Dastani, M. and Meyer, J.-J. C. (2010). *Specification and Verification of Multi-agent Systems*, chapter Correctness of Multi-Agent Programs: A Hybrid Approach. Springer.

Dennis, L. A. and Nogueira, P. (2005). What can be learned from failed proofs of non-theorems. Technical report, Oxford University Computer Laboratory.

Drogoul, A., Ferrand, N., and Müller, J.-P. (2004). Emergence : l'articulation du local au global. *ARAGO*, 29:105–135.

Dung N. Lam, K. S. B. (2005). Automated Interpretation of Agent Behaviour. In *AOIS*, pages 1–15.

Kaufmann, M. and Moore, J. (2008). Proof Search Debugging Tools in ACL2. http://www.cs.utexas.edu/users/moore/-publications/acl2-papers.html.

Lam, D. N. and Barber, K. S. (2005). Comprehending agent software. In *AAMAS*, pages 586–593.

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

Mermet, B. and Simon, G. (2013). A new proof system to verify gdt agents. In *IDC*, pages 181–187.

Miles, S., Winikoff, M., Cranefield, S., Nguyen, C., Perini, A., Tonella, P., Harman, M., and Luck, M. Why testing autonomous agents is hard and what can be done about it. AOSE Technical Forum 2010 Working Paper.

Nguyen, C., Perini, A., Bernon, C., Pavón, J., and Thangarajah, J. (2009). Testing in Multi-Agent Systems. In *AOSE*, pages 180–190.

Nguyen, C., Perini, A., and Tonella, P. (2008). Ontology-based test generation for multiagent systems. In *AAMAS*, pages 1315–1320.

Nguyen, C. D., Perini, A., and Tonella, P. (2010). Goal-oriented testing for MASs. *IJAOSE*, 4(1):79–109.

Owre, S., Shankar, N., and Rushby, J. (1992). Pvs: A prototype verification system. In *CADE 11*.

Serrano, E., Gómez-Sanz, J., Botía, J., and Pavón, J. (2009). ntelligent data analysis applied to debug complex software systems. *Neurocomputing*, 72(13-15):2785–2795.

Tiryaki, A., Öztuna, S., Dikenelli, O., and Erdur, R. (2006). SUNIT: A Unit Testing Framework for Test Driven Development of Multi-Agent Systems. In *Agent Oriented Software Engineering (AOSE)*, pages 156–173.

Vigueras, G. and Botía, J. (2007). Tracking Causality by Visualization of Multi-Agent Interactions Using Causality Graphs. In *PROMAS*, pages 190–204.

Zhang, Z., Thangarajah, J., and Padgham, L. (2009). Model based testing for agent systems. In *AAMAS'09*, pages 1333–1334.