Les effets de bord des agents ne sont pas que le fruit du hasard

Bruno Mermet, Gaële Simon

Les effets de bord des agents ne sont pas que le fruit du hasard

Bruno Mermet Gaële Simon

GREYC - CNRS
UMR 6072

Prenom.Nom@univ-lehavre.fr

Résumé

Une des spécificités des SMA est l'apparition d'effets de bord au niveau du système : des propriétés inattendues apparaîssent. On parle alors de propriétés de vivacité. Suivant les cas, il peut s'agir de propriétés intéressantes ou au contraire, de défauts du système. Dans tous les cas, il peut être intéressant de prouver l'apparition de telles propriétés, mais il n'existe pas de système de preuve adapté. GDT4MAS est une méthodologie qui permet de spécifier des SMA et de les prouver. Cependant, elle est actuellement plus particulièrement orienté vers la preuve de propriétés d'invariance. Dans cet article, nous proposons un mécanisme de preuve permettant d'enrichir GDT4MAS pour prouver l'apparition d'effets de bord. Plus précisément, nous montrons comme il est possible d'ajouter à GDT4MAS un mécanisme de preuve adapté à des propriétés de type "leads-to", un cas particulier de propriété de vivacité.

Mots clef

SMA, preuve, propriétés de vivacité, GDT4MAS

Abstract

Side effects are an important characteristic of MAS, and proving them is an interesting issue. But there is no system dedicated to this kind of proof. The GDT4MAS framework allows to specify and prove the correctness of multiagent systems. However, 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 address 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.

Keywords

SMA, proof, liveness properties, GDT4MAS

1 Introduction

Lors de l'exécution d'un SMA, on observe souvent l'apparition de propriétés inattendues au niveau du système. Ces propriétés peuvent présenter un intérêt certain (on parlera alors par exemple de propriété *émergente*) ou, au contraire, s'avérer néfastes. Dès lors, il peut devenir intéressant de prouver l'apparition de telles propriétés. Mais à notre connaissance, aucun système adapté à la preuve de ces propriétés dans les SMA n'existe.

Prouver la correction d'un système multi-agents est un problème difficile abordé par un certain nombre de travaux depuis plusieurs années [11]. Parmi eux, l'approche GDT4MAS [9] a des caractéristiques intéressantes. En particulier, la génération d'obligations de preuve est entièrement automatisable, tout en étant applicable à des systèmes de taille conséquente. Elle repose en effet sur la logique du premier ordre et des techniques de preuve de théorèmes plutôt que sur une logique propositionnelle et des techniques de model-checking.

Cependant, si cette approche est bien adaptée à la vérification de propriétés d'invariance (aussi appelées propriétés de sûreté) et à la preuve de la correction des agents (i.e. vérifier que les agents vont bien satisfaire leur but principal), elle ne permet pas à l'heure actuelle de prouver qu'un agent vérifie des propriétés de vivacité qui ne sont pas des conséquences directes de la résolution de leur but principal. Ces propriétés sont pourtant essentielles lorsque l'on s'intéresse à la dynamique d'un système multi-agents. Elles peuvent notamment aider dans la mise au point du comportement d'un agent en mettant en évidence qu'au delà de la résolution de sa tâche, dans certains contextes, il établit des propriétés souhaitées ou non que l'on peut voir comme des effets de bord.

Dans cet article, nous proposons donc d'ajouter à cette approche un nouveau système de preuve dédié à la preuve de propriétés *leads-to*, un cas particulier bien connu et assez général de propriétés de vivacité. Dans le paragraphe suivant, nous introduisons brièvement la notion de propriété de vivacité. Dans la partie 3, nous rappelons les principaux concepts de l'approche GDT4MAS. Le nouveau système de preuve proposé est décrit dans la partie 4. Il est partiellement illustré dans la section 5. Enfin, la section 6 fournit

une brève comparaison avec d'autres travaux.

2 Propriétés

Lorsqu'il est question de vérification formelle, plusieurs types de propriétés peuvent être considérées. Dans cette partie, nous en présentons 2 types : les propriétés d'invariance et les propriétés de vivacité.

2.1 Propriétés d'invariance

Les propriétés d'invariance spécifient un ensemble d'états que le système doit satisfaire à tout moment. Elles sont souvent présentées comme des propriétés spécifiant que "rien de mauvais ne peut arriver". Ces propriétés sont pour la plupart des propriétés de sureté. Dans les systèmes de vérification formelle comme la méthode B [1], utilisés pour le développant de systèmes critiques, ces propriétés sont les seules à être prises en comptes. En logique temporelle, une propriété d'invariance est spécifiée ainsi : $\Box(IP)$.

2.2 Propriétés de vivacité

A l'inverse des propriétés d'invariance, les propriétés de vivacité spécifient comment le système est susceptible de modifier son état. Elles sont souvent présentées comme des propriétés spécifiant que "quelque chose de bien doit arriver". Il y a beaucoup de types de propriétés de vivacité. Dans cet article, nous nous focalisons sur un type assez général de telles propriétés : les propriétés de type *leads-to*. Le lecteur pourra trouver des détails supplémentaires par exemple dans [3, 8].
Une propriété de type *leads-to* est une propriété qui spécifie qu'un prédicat LT finira par devenir vrai dès lors qu'un autre prédicat P aura été vrai auparavant: $\Box(P \rightarrow \Diamond LT)$
L'intérêt de ce type de propriété est que l'on peut aussi les utiliser pour spécifier des situations non souhaitées qui, si elles sont prouvées, peuvent permettre de détecter des effets de bord gênants dans le comportement d'un agent.

3 Approche GDT4MAS et notations

3.1 Concepts principaux

Dans l'approche GDT4MAS, le système multi-agents est décrit par un environnement, principalement spécifié par un ensemble de variables, un invariant (noté $i_\mathcal{E}$ dans la suite) et une population d'agents évoluant dans l'environnement. Chaque agent est défini comme une instance de type d'agent.
Dans cette partie, nous donnons quelques définitions informelles des notions et notations utiles à la compréhension de l'article. Ces définitions se bornent à introduire uniquement les éléments nécessaires à la compréhension de cet article, c'est pourquoi elles sont appelées *définitions simplifiées*. Nous invitons le lecteur à se référer à de précédentes publications pour avoir plus de précisions sur GDT4MAS et le système de preuve associé [9, 10].

Notation 3.1 (variable primée et non primée) *La valeur d'une variable v peut être considérée dans 2 états : celui avant un événement et celui après cet événement. L'événement peut être l'exécution d'une action par un agent, la résolution d'un but par un agent etc. La valeur de la variable dans le premier état est notée v tandis que la valeur de la variable dans le second état est notée v'. Par exemple, l'action consistant à augmenter la valeur de v de 1 est spécifiée par la postcondition $v' = v + 1$.*

Notation 3.2 (Invariant) *Soit A un agent situé dans un environnement $\mathcal{E}$. On note :*

- *i_A l'invariant sur les variables de l'agent;*
- *$i_\mathcal{E}$ l'invariant sur les variables d'environnement;*
- *$i_{\mathcal{E}A}$ la conjonction de i_A et de $i_\mathcal{E}$.*

Définition simplifiée 3.1 (Type d'agent) *Un type d'agent t est décrit par un nom ($name_t$), un ensemble de variables internes ($VarI_t$), un invariant (i_A) et un comportement (b_t) défini par un GDT.*

Définition simplifiée 3.2 (GDT) *Un GDT (*Goal Decomposition Tree*) décrit le comportement des agents instance d'un type d'agent. Chaque noeud de cet arbre correspond à un but de l'agent. La structure de l'arbre repose sur la décomposition de chaque but en sous-buts par des opérateurs de décomposition. Un prédicat appelé Triggering Context (TC) est associé à chaque GDT. Un agent exécutera le comportement spécifié par le GDT à chaque fois que le TC est vrai.*

Définition simplifiée 3.3 (but) *Un but G d'un GDT est notamment décrit par un nom ($name_G$), une condition de satisfaction (sc_G), une gpf (gpf_G), une décomposition et un label ns (ns_G). La condition de satisfaction est un prédicat spécifiant les propriétés devant être vérifiées lorsque le but réussit. La gpf (*Guaranted Property in case of Failure*) est un prédicat spécifiant les propriétés qui sont vérifiées lorsque le but échoue. Le label ns spécifie si le but réussit toujours (NS) ou pas (NNS). Dans la suite, lorsque ce label vaut NS (resp. NNS), on parlera de but NS (resp. NNS).*

Définition simplifiée 3.4 (Décomposition d'un but) *Un but est soit un but feuille soit un but intermédiaire. Dans le cas d'un but feuille, une action est associée au but. L'exécution du but déclenche l'exécution de l'action. Un but intermédiaire est décomposé en sous-buts en utilisant un opérateur de décomposition.*

Une liste d'opérateurs de décomposition est décrite dans [11]. Dans cet article, nous n'utiliserons que 2 opérateurs, *SeqAnd* (*et* séquentiel) et *SeqOr* (*ou* séquentiel), éventuellement dans leur version synchronisée (*SyncSeqAnd* et *SyncSeqOr*), permettant de verrouiller des variables d'environnement.

Définition simplifiée 3.5 (Action) *Une action α est spécifiée par un nom ($name_\alpha$), une précondition (pre_α), une*

postcondition ($post_\alpha$), un label ns (ns_α) et une gpf (gpf_α). La précondition est un prédicat spécifiant les propriétés devant être vérifiées pour que l'action puisse s'exécuter. La postcondition spécifie ce que l'action fait. Le label ns spcifie si l'action réussit toujours ou non. La gpf spécifie les propriétés qui sont malgré tout vérifiées quand l'action échoue.

Dans GDT4MAS, un certain nombre de transformeurs de prédicats sont utilisés. Nous rappelons ici la définition du seul transformeur qui sera utilisé par la suite.

Notation 3.3 (Primer) *Soit f une formule (prédicat ou expression). Si f contient au moins une variable primée alors $pr(f) = f$. Sinon $pr(f)$ est un prédicat obtenu à partir de f dans lequel chaque variable non indicée est primée.*
Exemples: $pr((x = x_0)) \equiv (x' = x_0)$ et $pr((x = x')) \equiv (x = x')$.

3.2 Principes généraux des mécanismes de preuve de GDT4MAS

Le système de preuve de GDT4MAS repose sur la définition de schémas de preuve. Un schéma de preuve permet de générer des obligations de preuve, prouvables automatiquement par un prouveur comme PVS [13]. Dans la version actuelle, les obligations de preuve générées permettent de garantir la correction des agents ainsi que la préservation d'invariants par le SMA. Certains schémas de preuve reposent sur l'utilisation des contextes des buts. Ceux-ci sont calculés automatiquement par des règles d'inférence appliquées sur le GDT en partant du but racine. Intuitivement, le contexte C_G d'un but G est un prédicat résumant l'état dans lequel le but G va être exécuté.

4 Un nouveau mécanisme de preuve dédié à la preuve de propriétés de type leads-to

Dans cette partie, nous présentons le nouveau mécanisme de preuve que nous proposons pour vérifier que des propriétés de type leads-to sont vérifiées par des agents. Il est important de noter que ce mécanisme n'est applicable que pour des GDT donc la correction a été préalablement prouvée. On ne s'intéresse dans cet article qu'à des propriétés qui peuvent être établies par un seul agent sans que d'autres agents soient nécessaires.
Dans cette partie, on considère une propriété *leads-to* L définie ainsi : $L \equiv \Box(P_L \rightarrow \Diamond Q_L)$
La manière classique de prouver qu'une propriété *leads-to* est vérifiée par une spécification est de lui associer un *variant* et un *temoin* [3].
Informellement, un variant exprime une notion de progrès vers l'établissement de Q_L. Si l'on parvient à prouver qu'un agent fait décroitre un variant et que, lorsque le variant atteint sa borne, Q_L est vérifiée, alors la propriété *leads-to* L est prouvée. Un témoin est une propriété qui représente le fait que P_L est devenue vrai et que, par conséquent, Q_L devra être établie.
Dans cet article, nous proposons d'adapter ce principe général à la preuve de propriétés *leads-to* engendrées par le comportement d'un agent.

4.1 Définitions et notations

Définition 4.1 (Variant) *Un variant est une séquence décroissante définie par une structure bien fondée. On note V_L le variant utilisé pour prouver une propriété L.*

Bien sûr, cette définition nécessite de définir ce qu'est une structure bien fondée :

Définition 4.2 (Structure bien fondée) *Une structure bien fondée $(S, <)$ est définie par un ensemble S avec une relation d'ordre $<$ tels que toute séquence décroissante dans S a une borne inférieure. Par exemple, $(\mathbb{N}, <)$ est une structure bien fondée.*

Corollaire 4.1 *Un variant a une borne inférieure. On note V_{0_L} la borne inférieure d'un variant V_L.*

Définition 4.3 (Témoin) *Soit $L \equiv \Box(P_L \rightarrow \Diamond Q_L)$ une propriété leads-to. Un témoin W_L est une propriété qui doit être vraie lorsque P_L devient vrai et, qui reste vraie jusqu'à ce que Q_L soit vrai.*

4.2 Détails du processus de preuve

Grâce au variant et au témoin associés à la propriété *leads-to*, prouver qu'un agent d'un type T établit cette propriété consiste à prouver que :

1. Le variant est bien un variant :
   - quand le variant a atteint sa borne inférieure, Q_L est vrai;
   - à partir du moment où P_L est devenue vrai et jusqu'à ce que Q_L devienne vrai, un agent de type T doit exécuter son GDT;
   - aucun autre agent ne fait croître le variant.
2. Le témoin W_L est bien un témoin :
   - W_L est vrai quand P_L est vrai;
   - quand W_L est vrai, il reste vrai jusqu'à ce que Q_L devienne vrai.
3. Un agent de type T progresse i.e. quand l'agent exécute son GDT, il fait décroitre le variant ou il vérifie la propriété Q_L.

Dans la suite de cette partie, nous détaillons ces étapes.

4.3 Le variant choisi est bien un variant !

Afin de prouver que V_L est un variant, on doit prouver que, quand il a atteint sa borne inférieure, la propriété à laquelle il est associé est vérifiée. Cela nécessite donc d'ajouter l'obligation de preuve générée par le schéma de preuve :

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

De plus, il faut prouver qu'à partir du moment où P_L est devenue vrai, et jusqu'à ce que Q_L devienne vrai, l'agent de type T va bien exécuter son GDT. Ceci est vérifié en prouvant la propriété obtenue par le schéma de preuve suivant dans lequel TC est le *triggering context* du GDT :

$$i_{\mathcal{E}A} \wedge W_L \wedge \neg Q_L \rightarrow TC_A \quad (2)$$

Enfin, il faut également prouver qu'à partir du moment ou P_L est vérifié, aucun autre agent ne fait croitre le variant tant que Q_L n'est pas vérifié. Par conséquent, pour chaque agent A du système et pour chaque action α utilisée dans un but feuille G de son GDT, il faut vérifier que :

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

4.4 La propriété témoin est bien un témoin !

Comme expliqué auparavant, le témoin W_L associé à la propriété *leads-to* L doit vérifier les propriétés suivantes :

- **Initialisation** W_L doit être vrai quand P_L est vrai;

  Cet aspect doit être vérifié en prouvant la propriété suivante :

  $$i_{\mathcal{E}A} \wedge P_L \rightarrow W_L \quad (4)$$

- **Finalisation** W_L reste vrai jusqu'à ce que Q_L devienne vrai.

  Pour chaque autre agent du système, on doit doit vérifier qu'à chaque fois qu'il exécute une action (réussissant ou échouant), si le témoin est vrai avant l'exécution de l'action, alors il est toujours vrai après sauf si Q_L est devenu vrai. Donc, pour chaque action α associée à un but feuille G de chaque agent A, il faut prouver la propriété suivante :

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

4.5 L'agent progresse

Afin de montrer que chaque exécution de son GDT par un agent du type T constitue un progrès vers l'établissement de la propriété Q_L, il faut prouver que l'exécution du but principal du GDT engendre bien un tel progrès. En effet, l'exécution du but principal d'un GDT implique une exécution particulière de ce GDT. Il faut donc montrer que le but racine du GDT de l'agent est un *but de progrès*.

Définition 4.4 (But de progrès) *On appelle* but de progrès *un but dont l'exécution soit fait décroître le variant soit établit la propriété Q_L. Pour une propriété* leads-to L*, on associe à chaque but G un booléen pg_{L_G} qui est vrai si et seulement si G est un but de progrès pour L.*

Déterminer si un but est un but de progrès peut être mis en oeuvre via des règles d'inférence reposant sur la structure du GDT. Il faut pour cela commencer par déterminer quels buts feuille sont des buts de progrès. Ensuite, ces informations peuvent être propagées par des règles d'inférence associées à chaque opérateur de décomposition pour déterminer quels buts intermédiaires sont des buts de progrès.

Cependant, comme l'exécution du GDT dépend de la réussite ou non des différents but exécutés, il faut également déterminer dans quel cas un but donné est un but de progrès. On va pour cela distinguer les *buts de progrès en cas de succès* des *buts de progrès en cas d'échec*.

Définition 4.5 (But de progrès en cas de succès (spg)) *On appelle* but de progrès en cas de succès *un but qui soit fait décroitre le variant soit établit la propriété Q_L lorsqu'il est exécuté et qu'il réussit. Pour une propriété leads-to L, on associe à chaque but G un booléen spg_{L_G} qui est vrai si et seulement si G est un but de progrès en cas de succès.*

Définition 4.6 (But de progrès en cas d'échec (fpg)) *On appelle* but de progrès en cas d'échec *un but qui soit fait décroitre le variant soit établit la propriété Q_L lorsqu'il est exécuté et qu'il échoue. Pour une propriété leads-to L, on associe à chaque but G un booléen fpg_{L_G} qui est vrai si et seulement si G est un but de progrès en cas d'échec.*

Corollaire 4.2 *Un but est un but de progrès si et seulement si c'est à la fois un but de progrès en cas de succès et un but de progrès en cas d'échec. Pour chaque but G, on a : $pg_{L_G} = spg_{L_G} \wedge fpg_{L_G}$.*

Dans les paragraphes suivants, on commence par présenter comment on peut déterminer les buts feuille spg et fpg. On présente ensuite deux conditions suffisantes aidant à déterminer si les autres buts sont spg et fpg, l'une de ces conditions reposant sur une inférence à partir des statuts (spg et fpg) des buts feuille. Pour finir, nous décrivons les schémas de preuve associés aux buts feuille non spg et non fpg.

Déterminer l'ensemble des buts feuille fpg et spg. Pour déterminer si un but feuille est un but spg, il faut vérifier que lorsque ce but réussit (et que par conséquent sa condition de satisfaction est vérifiée), soit il fait décroitre le variant soit la propriété Q_L est vraie. Bien sûr, on ne doit prendre en compte que les exécutions de ce but dans le contexte où P_L est devenu vrai ce qui est représenté par le fait que W_L est vrai. Par conséquent, voici la propriété devant être vérifiée par tout but feuille spg G :

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

De la même manière, un but feuille G est un but fpg si et seulement s'il vérifie la propriété suivante :

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

Notons que, d'après cette dernière propriété, comme les buts *NS* ont leur gpf à $false$, ils sont donc *fpg*.

Inférence des propriétés spg et fpg pour les buts non feuille. Une première façon de garantir qu'un but non feuille est spg/fpg consiste à montrer que sa décomposition l'implique. Ici, nous illustrons ce processus pour les opérateurs SeqAnd, SyncSeqAnd, SeqOr et SyncSeqOr.

SeqAnd et SyncSeqAnd Soit G un but décomposé en G_1 $SeqAnd$ G_2. Afin de déterminer si G est spg, il faut considérer tous les contextes d'exécution dans lesquels ce but peut réussir. Bien sûr, G est exécuté avec succès lorsque G_1 et G_2 sont exécutés avec succès. Mais nous rappelons (voir [11]) qu'en raison de la prise en compte d'effets de bord éventuels, G peut réussir même si G_1 a échoué ou encore si ce dernier a réussi, conduisant selon la sémantique du SeqAnd, à l'exécution de G_2 et que ce dernier a échoué. Aussi, pour s'assurer que G est un but spg, il doit vérifier fpg_{G_1}, $spg_{G_1} \vee spg_{G_2}$ et $spg_{G_1} \vee fpg_{G_2}$. Cela peut se ramener à la règle d'inférence suivante :

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

Pour déterminer si G est un but fpg, on doit prendre en compte les 2 contextes d'exécution dans lesquels il peut échouer à savoir : lorsque le premier sous-but échoue ou lorsque le second sous-but échoue après que le premier ait réussi. D'où la seconde règle d'inférence :

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

SeqOr et SyncSeqOr Soit G un but non-paresseux décomposé en G_1 $SeqOr$ G_2. Le but G peut réussir dans 3 cas : le but G_1 réussit, le but G_1 échoue puis le but G_2 réussit ou encore le but G_1 échoue puis le but G_2 échoue mais, en raison d'effets de bord, le but G réussit malgré tout. Cela permet de déduire la règle d'inférence suivante :

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

Le but G est susceptible d'échouer uniquement si G_1 et G_2 échouent. Ce qui permet de déduire la propriété suivante permettant de déterminer si le but G est un but fpg :

$$fpg_{L_{G_1}} \vee fpg_{L_{G_2}} \rightarrow fpg_{L_G} \quad (11)$$

Utilisation des conditions de satisfaction des buts pour déterminer s'ils sont spg L'utilisation des règles d'inférence 8 et 10 pour déterminer si un but est spg sont certes suffisantes, mais pas forcément nécessaires. Notamment, si la condition de satisfaction d'un but non feuille permet d'établir la décroissance du variant (ou l'établissement de la propriété Q_L), le but peut également être considéré comme spg. Aussi, pour tout but intermédiaire G pour lequel les règles d'inférence décrites précédemment n'ont pas permis d'inférer qu'il est spg, nous vérifierons également si la propriété 6 est vérifiée.

buts feuille non fpg et non spg. Lorsqu'un un but G n'est pas un but spg, il faut prouver que ce but ne fait pas croitre le variant lorsqu'il réussit. Donc, pour chaque but non spg, il faut prouver la formule suivante :

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

De la même façon, pour chaque but G qui n'est pas fpg, il faut prouver la formule suivante :

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

5 Application sur un petit exemple

Nous avons choisi ici volontairement un exemple très simple afin de pouvoir facilement illustrer l'essentiel des principes de la preuve. Toutefois, faute de place, les détails ne sont pas donnés directement dans cet article, mais peuvent être trouvés en ligne [2]. On considère donc un SMA dont un seul agent a modifie le variant. L'environnement du système est défini par deux variables x et d et par l'invariant suivant : $i_{\mathcal{E}} \triangleq x \in \mathbb{N} \wedge d \in \mathbb{B} \wedge (d \leftrightarrow (x > 0 \wedge x \leq 10))$

L'agent a a un comportement décrit par un GDT donné dans la figure 1. Chaque but est représenté par son nom (de A à E) et sa condition de satisfaction simplifiée. La simplification des conditions de satisfaction consiste à ne pas faire figurer les formules spécifiant que les autres variables que celles manipulées lors de la résolution du but ne sont pas modifiées. Par exemple, la condition de satisfaction complète du noeud D est $y' = 2 \wedge x' = x \wedge d' = d$.

Voici, de plus, le triggering context du GDT précédent, l'invariant de l'agent et les *gpf* des noeuds NNS :

TC_a	$\triangleq$	d	gpf_E	$\triangleq$	$x' = x$
i_a	$\triangleq$	$(y \in \mathbb{N})$	gpf_B	$\triangleq$	$x' = x$

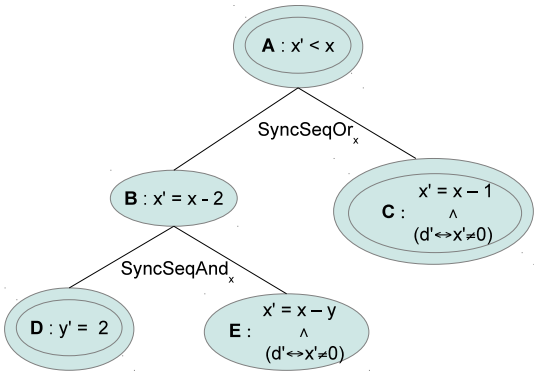

Figure 1: GDT de l'exemple

La propriété leads-to que nous avons prouvée est :

$$L \triangleq \Box(x = 10 \rightarrow \Diamond x = 0) \quad (14)$$

Pour prouver cette propriété, et en utilisant les notations utilisées dans la partie précédente, nous posons :

P_L	$\triangleq$	$(x = 10)$	Q_L	$\triangleq$	$(x = 0)$
V_L	$\triangleq$	(x)	V_{0_L}	$\triangleq$	(0)
W_L	$\triangleq$	(d)			

Comme cela est détaillé dans [2], nous avons notamment établi que les buts feuilles C et E sont spg et que les buts feuille C et D sont fpg. B est identifié comme but spg non par inférence, mais grâce à la règle 6. Par inférence, nous déduisons que A, but NS est spg. Donc le GDT de l'agent étudié fait bien décroître le variant. L'application des autres schémas de preuve de l'article est illustrée dans [2].

6 Discussion

Plusieurs langages de spécification formelle dédiés aux SMA ont été proposés. Cependant, ils ne sont le plus souvent pas conçus pour la preuve. C'est par exemple le cas de 2APL [4] qui est plus un langage de programmation Metatem [6] fournit au développeur la possibilité de spécifier des propriétés. Le système controle ensuite à l'exécution que ces propriétés ne sont pas violées. Cependant il s'agit d'un processus de preuve par construction. Cela signifie que c'est uniquement à l'exécution que la preuve est construite et donc, si les conditions initiales changent, une nouvelle preuve reposant sur une exécution à partir du nouvel état initial doit être construite.

La plupart des travaux traitant de vérification de SMA reposent sur des techniques de model-checking. Un des travaux les plus récents dans ce domaine est la définition de AJPF [5], un model-checker reposant sur JPF [12] et AIL (*Agent Infrastructure Layer*). C'est, à notre connaissance, la seule approche proposant de vérifier des propriétés de type leads-to pour des SMA. Un premier inconvénient de l'approche est le temps pris par la preuve qui peut durer plusieurs heures pour un système simple. Bien sûr, un model-checker plus optimisé comme Spin [7] permettrait de réduire de manière importante le temps nécessaire à la preuve. Néanmoins ce type d'approche reste plus adapté à des systèmes de petite taille. De plus cette approche a un autre inconvénient important. Bien qu'elle permette de prouver une propriété telle que celle dont nous avons fait la preuve dans la partie 5, elle ne permet par contre pas de prouver des propriétés dont la partie gauche (ie. $x = 10$ dans notre exemple) caractérise un nombre infini d'états. Par exemple, si l'on s'intéresse à prouver la propriété leads-to suivante : $\Box(x \geq 10 \rightarrow \Diamond x = 0)$, une approche de type model-checking va échouer tandis que le processus de preuve qui a été présenté dans cet article pourrait être appliqué de la même manière que dans l'exemple.

Notre approche, quant à elle, génère un nombre d'obligations de preuve qui est linéaire par rapport au produit du nombre de propriétés par la somme du nombre de noeuds pour chacun des types d'agents du système. En effet, nous générons environ 2 obligations de preuve par noeud pour chaque propriété d'invariance. Pour les propriétés de vivacité, nous générons environ $6nf$ obligations de preuve, où nf est le nombre de noeuds feuille présents dans l'ensemble des GDT des différents types d'agent. Nous gardons donc une complexité globalement linéaire en le nombre de types d'agents.

7 Conclusion et perspectives

Dans cet article, nous avons montré que GDT4MAS, au départ principalement dédié à la preuve de propriétés d'invariance sur des SMA, peut être étendu pour la preuve de propriété de vivacité de type leads-to. Nous avons ici considéré seulement le cas des propriétés de vivacité établies par un seul agent. Bien sûr, d'autres propriétés plus générales comme les propriétés de vivacité établies collectivement par un ensemble d'agents du système doivent être envisagées. Ceci constitue une perspective à court terme d'évolution supplémentaire de l'approche GDT4MAS. De plus, comme c'est le cas dans les systèmes de vérification classiques, le système de preuve proposé dans cet article permet uniquement de prouver des propriétés leads-to de type *P leads-to Q* pour lesquelles il y a un progrès continu vers l'établissement Q une fois que P est devenu vrai. Dans un SMA où les agents peuvent être complètement autonomes, il faut aussi considérer des propriétés pour lesquelles le progrès n'est pas continu. Ceci constitue une perspective importante à long terme de notre travail.

References

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

[2] B. Mermet and G. Simon. Preuve de propriétés de vivacité sur un petit exemple de GDT. https://mermet.users.greyc.fr/GDT/RFIA/preuve.pdf.

[3] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[4] M. Dastani. 2APL: a practical agent programming language. *Journal of Autonomous Agents and Multi-Agent Systems*, 16:214–248, 2008.

[5] L. Dennis, M. Fisher, M. Webster, and R. Bordini. Model checking agent programming languages. *Automated Software Engineering*, 19(1):5–63, 2012.

[6] M. Fisher. A survey of concurrent METATEM – the language and its applications. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic - Proceedings of the First Intemational Conference (LNAI Volume 827)*, pages 480–505. Springer-Verlag: Heidelberg, Germany, 1994.

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

[8] L. Lamport. The syntax and semantics of tla$^+$. Part 1: Definitions and Modules, June 1996.

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

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

[11] B. Mermet, G. Simon, A. Saval, and B. Zanuttini. Specifying, verifying and implementing a MAS: A case study. In *Post-Proc. of ProMAS'07*, number 4908 in Lecture Notes in Artificial Intelligence, pages 172–189. Springer, 2007.

[12] NASA. Java Path Finder. http://babelfish.arc.nasa.gov/trac/jpf.

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