Contrôle de l'émergence dans les systèmes d'agents cognitifs autonomes

Caroline Chopinaud*, — Patrick Taillibert* — Amal El Fallah Seghrouchni****

**Thales Systèmes Aéroportés*
1 boulevard Jean Moulin F-78852 Elancourt
{caroline.chopinaud,patrick.taillibert}@fr.thalesgroup.com

***LIP6*
8 rue du Capitaine Scott F-75015 Paris
amal.elfallah@lip6.fr

RÉSUMÉ. *Pouvoir garantir qu'un système multiagent ne va pas générer des comportements indésirables est primordial dans le contexte d'applications critiques (systèmes embarqués ou temps réels). L'émergence de comportement due à la mise en relation d'agents autonomes peut engendrer l'apparition de situations incompatibles avec la bonne exécution du système. L'utilisation de méthodes classiques de validation, pour vérifier le comportement d'un système multiagent, n'est pas suffisante si l'on souhaite garantir que, une fois mis en condition réelle, le système ne produira pas un mauvais comportement. Nous proposons une approche complémentaire d'auto-surveillance et d'auto-régulation permettant aux agents de contrôler leur propre comportement. Ce papier présente la génération automatique d'agents auto-contrôlés utilisant la technique de l'observateur afin de vérifier que leur comportement respecte un ensemble de loi tout au long de leur exécution.*

ABSTRACT. *Being able to ensure that a multiagent system will not generate undesirable behaviors is of prime importance within the context of critical application (embedded systems or real-time systems). Emergence of behaviors from the agents interaction can generate situations being inconsistent with the system execution. The use of classic techniques to validate a mutiagent system does not guarantee it against the occurence of undesirable behavior during its execution in real condition. We propose an additional approach of dynamic self-monitoring and self-regulation allowing the agents to control their own behavior. This paper presents the automatic generation of self-controlled agents using the observer approach in order to verify that their behavior respect a set of law throughout their execution.*

MOTS-CLÉS : *lois, observateur, résistance aux pannes, vérification, autonomie, émergence, contrôle*

KEYWORDS: *laws, observer, fault tolerance, verification, autonomy, emergence, control*

1. Introduction

Le comportement d'un système multiagent émerge de la mise en relation des agents le constituant. Ce comportement peut être prévu et correspondre aux attentes des concepteurs ou il peut être inattendu. Ce dernier type de comportement n'est pas toujours appréciable pour le fonctionnement de l'application. Des comportements émergents inattendus peuvent faire échouer le système, ce sont des comportements indésirables. L'autonomie est une caractéristique essentielle des agents cognitifs. Nous considérerons l'autonomie comme la capacité d'un agent à prendre seul ses décisions sans l'aide d'une autre entité [BAR 99]. Du point de vue du développeur, cela signifie que l'implémentation d'un agent nécessite de prendre en considération que le comportement des agents ne peut être prédit avec certitude. Cette vision accroît la possibilité d'apparition de comportements inattendus et indésirables lors de l'exécution du système.

Le fait de voir surgir des comportements indésirables potentiellement critiques pour la bonne exécution du SMA peut poser problème dans le cadre d'applications embarquées ou temps réel et lever des objections quant à son utilisation. Il est donc essentiel de pouvoir garantir qu'un système multiagent ne suivra pas des comportements indésirables, néfastes pour son exécution. L'objectif de notre approche est de fournir un mécanisme permettant de détecter et/ou empêcher l'apparition de tels comportements, qu'il soit dû à l'émergence naturelle de comportement ou à des erreurs d'implémentation au sein des agents. Une idée aurait pu être d'utiliser les méthodes classiques de validation de systèmes, telles que les tests, le model checking et la démonstration automatique, en les appliquant aux systèmes multiagents. Mais ces techniques ne sont pas en mesure de détecter toutes les erreurs possibles du système. Le model checking travaillant sur une abstraction du système et de son environnement d'exécution ne peut que détecter les erreurs de ce modèle ; la démonstration automatique est lourde et complexe ; les tests ne peuvent être exhaustifs [SCH 99]. Ainsi, pour détecter les erreurs restantes et les comportements indésirables, il est intéressant d'effectuer une vérification du système en cours d'exécution. Cette vérification[1] consiste en la surveillance et la régulation dynamique du système dans le but d'empêcher son échec.

Nous nous plaçons donc dans le contexte du monitoring logiciel (software) consistant à observer et comprendre le comportement d'un programme au cours de son exécution. Le programme est instrumenté en insérant des sondes logicielles dans le code pour détecter certains événements qui seront récupéré par un monitor [SOU 02]. Pour surveiller le comportement d'un système multiagent il faut donc insérer, dans le programme des agents, des détecteurs d'événements. Bien qu'il soit possible d'effectuer cette instrumentation manuellement cela peut s'avérer complexe : l'insertion de sondes dans un programme prend du temps, est difficile et encline à l'erreur [MAN 95]. Lorsque plusieurs agents sont concernés, les difficultés s'accroissent. Dès lors, il est intéressant de prendre en considération l'existence de techniques d'automatisation de

1. Nous emploierons par la suite le terme de contrôle.

l'instrumentation des programmes. Il en existe de deux sortes : *(1)* le développeur utilise un métalangage [LUM 90] ou une librairie de routines [HUA 95] pour insérer de façon transparente les sondes ; *(2)* l'insertion est effectuée par le compilateur à partir des spécifications des événements intéressants [LIA 92]. C'est cette dernière vision qui nous semble la plus intéressante pour instrumenter le programme des agents et ainsi réduire le travail du programmeur dans la mise en place du contrôle.

Enfin, nous pensons que les agents sont les mieux placés pour effectuer le contrôle de leur comportement. L'idée est donc de fournir aux agents les moyens nécessaires pour surveiller leur comportement et, grâce à leur capacité de raisonnement, ils vont pouvoir le réguler dans le but de se soustraire à un comportement indésirable. Ainsi, notre approche consiste à générer des agents capables d'effectuer le contrôle de leur propre comportement par instrumentation automatique de leur code, à partir de la description d'exigences. Dans la section 2 nous présenterons ce que l'on entend par contrôle d'agents autonomes. Dans la section 3, nous décrirons les différents étapes nécessaires à la mise en place d'agents auto-contrôlés. Enfin la section 4 présentera le framework SCAAR et illustrera son fonctionnement sur un exemple.

2. Contrôle d'agents autonomes

2.1. *Vérification de comportement*

Construire un modèle d'un SMA et de ses agents est généralement complexe (possible explosion du nombre d'état) compte tenu de la difficulté à cerner leur comportement dans leur globalité (indéterminisme, caractère distribué, communications asynchrones). Pour ne pas modéliser le comportement du système dans sa globalité, nous proposons d'utiliser les exigences associées aux agents et aux systèmes pour décrire les comportements ou les états désirables et indésirables. Les exigences considérées sont celles significatives ou critiques pour l'application. Elles représentent ce que l'on appellera les **lois** du système. Un agent capable d'auto-contrôle vérifie que les lois sont respectées tout au long de son exécution. Mais la surveillance du bon respect des lois ne suffit pas, quand un agent détecte la transgression d'une loi, il va réguler son comportement, à partir d'information de transgression, pour éviter de suivre un comportement indésirable. Afin que les agents puissent avoir conscience de la violation d'une loi et pour leur permettre de déduire le comportement à suivre suite à cette violation, nous supposons que les agents connaissent les lois. Soit le développeur des agents aura tout mis en oeuvre pour respecter les exigences et par conséquent il va, par construction, vérifier que les lois sont respectées, soit les agents ont une représentation des lois qu'ils vont tenter de respecter lors de leur exécution. La vérification du comportement des agents consiste alors à s'assurer que les lois sont bien respectées tout au long de l'exécution du système et ainsi éviter l'apparition de comportements indésirables. Ces derniers pouvant être dus soit à une erreur de programmation des agents, soit à leur exécution dans un contexte multiagent.

2.2. *Niveau de description des lois*

Nous souhaitons que la personne en charge du choix des lois ne soit pas forcément le développeur. Généralement les exigences sont fournies par un client et le développeur se charge de les prendre en compte dans l'implantation du système. Le client est alors le mieux placé pour exprimer ce qu'il est important de vérifier dans l'exécution du système, sans pour autant connaître l'implémentation des agents. Ainsi, nous supposons que les lois sont exprimées en Français par le client et qu'un expert se charge de les traduire dans un langage de description permettant d'en déduire le contrôle à appliquer aux agents. Il s'avère alors indispensable de pouvoir exprimer les lois à un niveau d'abstraction compréhensible par un client et permettant de facilement effectuer la traduction dans le langage de description des lois. De plus, par souci de généralité, nous souhaitons fournir un mécanisme de contrôle applicable à plusieurs modèles d'agents (tel que des agents BDI ou CLAIM). En effet, les agents d'une même application peuvent être construit suivant différents modèles et dans ce cas, un loi doit malgré tout être appliquable à ces agents. Le niveau de description des lois doit permettre d'inclure différents types de modèles d'agents. Aussi les lois doivent-elles porter sur des **concepts** généraux représentant les modèles d'agents utilisés et l'application. Pour ce faire, le développeur du modèle fournit un ensemble de concepts représentant les spécificités du modèle et le développeur du système fournit les concepts caractéristiques de l'application. C'est à partir de ces concepts que l'expert pourra décrire les lois exprimées par le client.

2.3. *Application du contrôle*

L'application du contrôle se divise en cinq étapes :

(1) Le développeur du modèle fournit une description des concepts et de leur représentation dans le programme.
(2) Le client fournit l'ensemble des lois qu'il souhaite voir être respectées par le système.
(3) Le développeur du SMA implante les agents à partir des exigences. Il fournit aux agents la possibilité de déduire le comportement à suivre suite à la violation d'une loi. Il décrit aussi les concepts représentatifs de l'application.
(4) Un expert écrit les lois exprimées en (2) à l'aide d'un langage de description fourni. Les lois portent sur les concepts décrits en (1) et en (2).
(5) La génération automatique des agents auto-contrôlés à partir de la description des lois (phase 4), des concepts utilisés dans les lois et de leur représentation dans le programme (phase 1) et du code de comportement des agents (phase 3).

Nous pouvons regrouper ces 5 étapes en 3 grandes parties :

- La description du système regroupant les étapes 1 et 3.
- La description des lois regroupant les étapes 2 et 4.
- La génération des agents auto-controlés correspondant à l'étape 5.

3. Génération d'agents auto-contrôlés

3.1. *Description du système*

Nous avons vu que les lois portent sur des concepts représentatifs du modèle et du système. Ces concepts permettent de décrire les spécificités des agents. Nous proposons une ontologie de base à partir de laquelle les développeurs du modèle et du système vont pourvoir ajouter leurs concepts en étendant les concepts existants. Un ensemble de concepts est proposé par D.N. Lam et K.S. Barber [LAM 04] dans le cadre d'une méthodologie de vérification du comportement des agents. Ces concepts sont : *But*, *Croyance*, *Intention*, *Action*, *Evénement*, *Message*. Nous gardons une partie de ces concepts (*But*, *Action*, *Message*) et nous ajoutons des concepts qui nous semblent plus caractéristiques des agents : *Objet*, *Connaissance*, *Plan*, *Création d'agent*, *Envoi de message*, *Réception de message*, *Migration*. Enfin, comme les lois peuvent être posées au niveau du SMA, nous ajoutons le concept d'*Agent*. Chaque concept a un ensemble d'attributs et de méthodes[2] permettant d'exprimer des tests dans les lois. La figure 1 représente l'ontologie des concepts de base que nous proposons. Les concepts sont répartis en deux catégories : les **états** (représentant les caractéristiques internes de l'agent) et les **actions**.

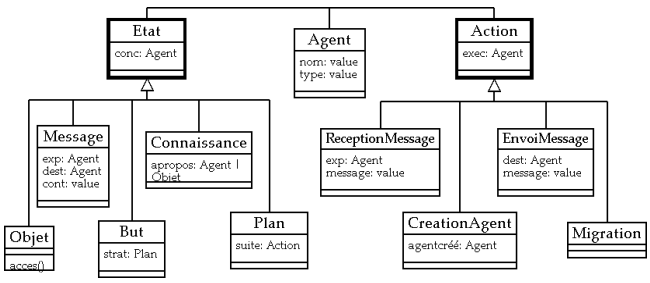


Figure 1. *Ontologie des concepts d'agent (UML))*

Les lois portant sur des concepts abstraits, il est nécessaire de connaître leur signification dans le programme des agents sous contrôle pour vérifier si la loi est respectée. Le concepteur du modèle se charge de décrire les liens entre les concepts abstraits représentant le modèle et l'implémentation du modèle. Cette description est à faire une seule fois par modèle d'agent utilisé et elle peut être validée pour garantir que toute loi utilisant ces concepts sera effectivement prise en compte.

2. Les méthodes ne sont pas décrites sur la figure

3.2. *Description des lois*

Nous distinguons deux types de lois : celles représentant les états ou les comportements interdits d'un agent. Elles permettent la détection de situations où un événement ne devant jamais apparaît va survenir ; celles représentant des états ou des comportements obligatoires pour un agent. Elles permettent la détection de situations où un événement attendu n'a pas lieu. Comme dans de nombreux travaux sur les normes [V´ 04], nos représentons les lois à l'aide de la logique déontique [WRI 51]. Cette dernière permet d'exprimer, entre autre, des interdictions et des obligations sur des propositions. Le premier type de loi correspond donc à une interdiction et le second à une obligation. Les lois exprimées en Francais par le client sont traduites par l'expert dans un langage permettant de décrire des interdictions, des obligations, du temps et un ordre entre des événements. Les événements utilisés dans les lois sont relatifs aux deux catégories de concepts que nous avons vu en 3.1, c'est-à-dire les changement d'état d'un agent et l'exécution d'une action. Les lois d'interdiction portant sur des actions empêchent l'exécution de cette action. En revanche, une loi d'interdiction sur un état ne peut que détecter l'apparition de cet état. Nous découpons les lois en trois parties :

- **AGENTS_CONCERNES(CA)** : contient la déclaration des agents concernés par la loi. Ce sont aussi bien les agents auxquels la loi sera appliquées que les agents utilisés pour décrire le contexte d'application de la loi.
- **ASSERTIONS_DEONTIQUE(DA)** : décrit les événements obligatoires (OBLI) ou interdits (FORB). C'est un ensemble d'associations entre un agent et un concept. Ce concept peut être une action (do) ou un état (have). Il est possible de définir des conditions sur ces concepts (and).
- **CONDITIONS_APPLICATION (APC)** : décrit les conditions sur le contexte d'application de la loi. C'est une expression décrivant quand l'assertion déontique doit être respectée relativement à un ensemble d'événement et/ou au temps.

La syntaxe du langage utilisé pour décrire les lois est, à ce jour, la suivante :

LAW := (CA)((DA APC)<**elapse_time**(TIME)>)
CA := *agent* : AGENT |*agent* : AGENT **and** PROP
DA := DEO (EVENT) |DEO (EVENT **and** PROP)
APC := QUA (EVENT) |QUA (EVENT **and** PROP)
DEO := **FORB** |**OBLI**
EVENT := *agent* **do** ACTION |*agent* **have** STATE
QUA := **AFTER** |**AND**
TIME := **sup**(*seconde*) |**inf**(*seconde*) |**equal**(*seconde*)
PROP := *des propriétés sur les paramètres d'un concept*
AGENT := *un concept d'agent*
ACTION := *un concept d'action*
STATE := *un concept d'état*

3.3. *Génération du contrôle*

A partir des concepts représentants le système, des lois décrites pour contrôler les agents et des programmes de comportement, les agents auto-contrôlés sont automatiquement générés. Il est donc nécessaire d'instrumenter le code des agents pour leur permettre de détecter les événement décrit dans les lois et analyser l'enchaînement de ces événements pour vérifier si la loi est respectée. Le processus de génération correspond alors à l'ajout des deux composantes suivantes au sein des agents :

- Des **points de contrôle** permettant la détection d'événements particuliers concernant le comportement ou l'état d'un agent lors de son exécution.
- Une **partie contrôle** qui réagit aux détections d'événements effectuées par les points de contrôle, analyse le comportement de l'agent et vérifie si les lois sont respectées.

Pour mettre en place ces deux composants nous allons utiliser deux techniques connues : les **observateurs**, pour vérifier le respect des lois et le **tissage** pour insérer les points de contrôle dans le code des agents.

La technique de l'observateur [DIA 94] est une approche qui consiste à faire évoluer un programme et son modèle en parallèle et de les comparer. Le modèle est lié au programme du système sous contrôle par des points de contrôle. Des propriétés concernant le programme sont modélisées, par exemple, par des réseaux de Petri dont les transitions sont liées aux programmes grâce aux points de contrôle. Lorsque l'exécution du programme arrive sur les points de contrôle, un contrôleur s'assure que les jetons sont en place au bon moment et les fait évoluer en conséquence. Lorsque l'exécution du système ne correspond pas au modèle, une exception est levée. Nous utilisons les observateurs pour vérifier que le comportement des agents correspond aux lois qui lui sont attribuées. Chaque lois est représentée par un réseau de Petri dans la partie contrôle des agents, dont les transitions sont liées aux points de contrôle insérer dans le programme des agents.

Le tissage est une part importante de la programmation par aspects. Ce type de programmation consiste en la modularisation de structures transverses (des aspects) qui vont être injectées par tissage dans l'application [WAM 03]. Ce tissage se fait grâce à la définition de points de jonction situés soit au niveau des méthodes, soit au niveau des flux d'exécution. Nous utilisons le tissage pour injecter dans le programme des agents les points de contrôle associés aux événements attendus dans les lois et relier l'observateur à l'éxecution du programme. Le tissage des événements se fait grâce aux liens fournis entre les concepts de l'application et les programmes des agents.

La génération des observateurs et la déduction des points de contrôle à insérer dans le programme des agents se fait à partir des lois définies dans le langage que nous avons présentée dans la section 3.2. Cette génération se fait en deux étapes : *(1)* partant de la loi on déduit une formule en logique déontique ; *(2)* de cette formule, on génère un réseau de Petri représentant la loi. La formule déontique se déduit de la loi en utilisant le tableau 1 de correspondance entre les mots-clés du langage et les opérateurs logiques. En utilisant les axiomes de la logique déontique, on obtient

LANGAGE	LOGIQUE
FORB (EV1) AFTER (EV2)	O(EV1(t1) ∧ inf(t1,t2) ⊃ ¬EV2(t2))
OBLI (EV1) AFTER (EV2)	O(EV1(t1) ∧ inf(t1,t2) ⊃ EV2(t2))
FORB (EV1) AND (EV2)	O((EV1 ⊃ ¬EV2) ∧ (EV2 ⊃ ¬EV1))
OBLI (EV1) AND (EV2)	O(EV1 ∧ EV2)
EVENT and prop	EVENT(prop)
ag do ACTION	ACTION(ag)
ag have STA	STA(ag) ∨ ¬STA(ag)
elapse_time(SEC)	time(t1,t2,SEC)

Tableau 1. *Correspondance entre les mots clés du langage et les expressions logiques.*

une formule permettant de mettre en avant les événements nécessaires pour vérifier le respect de la loi. Des points de contrôle permettant de détecter ces événements sont tissés dans le programme des agents, tandis que la structure de la formule déontique permet de déduire le réseau de Petri correspondant. Un ensemble de réseau de Petri est déduit de la formule déontique à l'aide du tableau 2. Par composition des différents mini-réseaux, on créé automatiquement un réseau de Petri final représentant la loi. Nous verrons dans la partie suivante un exemple de génération de réseau de Petri à partir d'une loi.

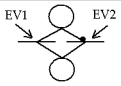
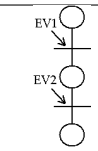
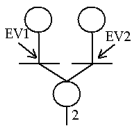
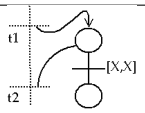
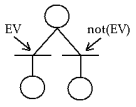
LOGIQUE	RES PETRI	LOGIQUE	RES PETRI
F		O	
EVENT		EV1 ∧¬EV2	
EV1 ⊃ EV2		EV1 ∧ EV2	
time(t1,t2,X)		EVENT ∨ ¬EVENT	

Tableau 2. *Traduction des expressions logiques en réseau de Petri*

Un observateur d'une loi ne doit pas être activé durant toute l'exécution. Tant que la première transition du réseau de Petri n'est pas activée les points de contrôle associées aux autres transitions du réseau ne doivent pas réagir lorsque le programme passe par eux. En effet, une interdiction ou une obligation n'a de signification que

sous certaines conditions. Donc, si l'observateur est toujours activé, il va nécessairement générer des exceptions, même si le comportement de l'agent est correct. Une solution aurait pu être de déduire des réseaux de Petri plus complexe permettant de ne pas générer ces exceptions hors-contextes, mais cette déduction n'est pas aisée si l'on souhaite pouvoir exprimer assez simplement les lois. Nous faisons donc en sorte que les observateurs soient activés en même temps que la ou les premières transitions du réseau (nous représenterons graphiquement cette activation par un triangle) et désactivé au niveau du passage des dernières transitions (la désactivation sera représentée par un carré). Une fois les observateurs générés et les points de contrôle insérés dans le code des agents, les agents auto-contrôlés sont générés et prêts à être exécutés. Un agent est divisé en deux parties, une partie représentant son comportement réel et une partie contrôle qui va s'assurer que la partie réelle respecte les lois qui lui sont attribuées. La partie contrôle contient les représentations des lois sous forme de réseau de Petri dont les transitions sont reliées aux points de contrôle contenu dans le code de comportement de l'agent. Lorsqu'une transition est tirée, si elle correspond à la transition d'activation de une ou plusieurs lois, les observateurs correspondants seront activés et en attente des événements suivants, leur permettant de vérifier si la loi est respectée. Dans le cas de la violation d'une loi, la partie comportement de l'agent est prévenue de cette transgression et elle va déduire la régulation à effectuer dans ce contexte de violation.

4. SCAAR : un framework pour la génération d'agents autonomes auto-contrôlés

4.1. *Présentation du framework*

Nous allons implanter la génération automatique des agents auto-contrôlés dans un framework : SCAAR (Self-Controlled Autonomous Agent geneRator). Ce framework va donc permettre la génération d'agents en mesure d'effectuer leur propre contrôle. Il mettra en place le processus de génération que nous avons decrit dans la section précédente et fournira l'ensemble des outils nécessaires à la description du système et des lois. Le framework se divisera en quatre parties :

- **Ontologie** : l'ensemble des concepts représentant les modèles d'agent utilisés et le système. Le framework fournit les concepts de base à partir desquels seront décrits les concepts spécifiques. Le framework fournit un langage simple pour exprimer ces concepts.
- **Lois** : l'ensemble des propriétés sur les états et les comportements des agents que ces derniers doivent respecter. Le framework fournit le langage de description des lois.
- **Lien au modèle** : l'ensemble des descriptions des liens entre les concepts abstraits et l'implémentation du modèle. Le framework fournit un langage simple pour exprimer ces liens.
- **Générateur** : la création de l'agent final à partir des éléments précédents. Le framework se charge de la génération automatique de ces agents capables d'auto-contrôle.

4.2. *Illustration du processus automatique de génération*

Nous nous plaçons dans le cadre d'un SMA simple de résolution de problème, constitué de trois types d'agents : un agent A, servant d'interface entre un utilisateur et le reste du système ; un agent B, en charge de la gestion des problèmes transmis par l'agent A ; un agent C, résolvant les problèmes envoyés par l'agent B. Le modèle d'agent utilisé est un réseau de Petri. La figure 2 représente les comportements des trois types d'agent.

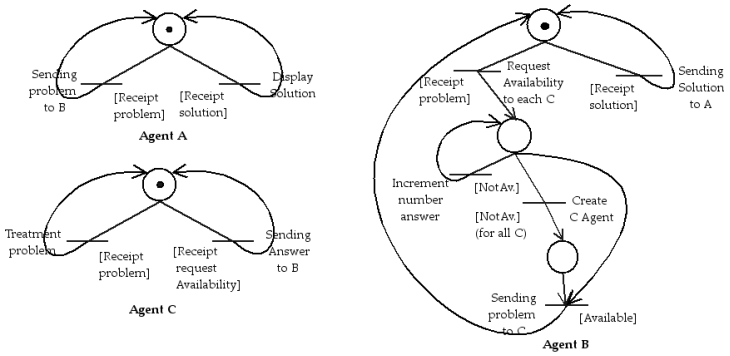


Figure 2. *Le comportement des agents.*

4.2.1. *Description du système*

Les concepts que l'on peut mettre en avant pour représenter le modèle d'agent sont : *ReceptionMessage* (au niveau de l'état du réseau), *Action* (au niveau des transitions) et *Agent*, qui sont des concepts de base, et *MessageRecu* qui est une extension du concept de base *Message*. Les concepts représentatifs de l'application sont : *EnvoiMessage* et *CreationAgent*, qui sont des concepts de base, *AffichageSolution*, *TraitementProbleme*, *Incremente* qui sont des extensions du concept d'*Action*. Enfin nous avons le concept d'*EtatAgent* qui est une extension du concept de *Connaissance* et qui a un paramètre valeur : disponible ou non.

Pour les concepts représentatifs du modèle il faut fournir les liens vers l'implémentation du modèle. Les agents sont programmés en Prolog. Nous allons prendre pour exemple le concept de *EnvoiMessage*. Ce concept correspond, dans l'implémentation du modèle d'agent à la clause *sendMessage* dont le premier argument correspond au message et le second au destinataire. La description du lien pour ce concept est :

```
link((EnvoiMessage == sendMessage\2),
    (EnvoiMessage.message == argument(1)),
    (EnvoiMessage.dest == argument(2)))
```

4.2.2. *Desctiption des lois*

Le client exprime les lois de ce système en Français (pour simplifier nous allons supposer qu'il connaît l'agentification de l'application). Prenons pour exemple la loi suivante :

L1 : "un agent A ne doit pas envoyer un message à un agent B à une cadence supérieure à un message par seconde".

En utilisant les concepts définis précédemment, l'expert va traduire la loi dans le langage défini à la section 3.2 :

(L1) (agA : Agent **and** type = typeA) (agB : Agent **and** type = typeB)
(**FORB** (agA **do** EnvoiMessage **and** receiver = agB)
AFTER (agA **do** EnvoiMessage **and** receiver = agB)) **elapse_time**(inf(1))

4.2.3. *Génération automatique*

A partir de cette loi, le framework génère automatiquement l'observateur correspondant. En utilisant la table 1, le framework déduit la formule logique suivante :

$$O(agent(agA, typeA) \wedge agent(agB, typeB) \wedge EnvoiMessage(agA, agB, x, t1) \wedge time(t1, t2, inf(1)) \supset \neg(agent(agA, typeA) \wedge agent(agB, typeB) \wedge EnvoiMessage(agA, agB, y, t2)))$$

En appliquant les axiomes de la logique déontique on obtient :

$$(O(agent(agA, typeA) \wedge agent(agB, typeB) \wedge EnvoiMessage(agA, agB, x, t1)) \wedge O(time(t1, t2, inf(1)))) \supset F(agent(agA, typeA) \wedge agent(agB, typeB) \wedge EnvoiMessage(agA, agB, y, t2))$$

A partir de cette formule, le framework déduit automatiquement le réseau de Petri représentant la loi à partir du tableau 2 (*cf* 3). La figure 3 est un exemple du processus de déduction du réseau de Petri associé à la loi L1. Le réseau obtenu est utilisé par la partie contrôle des agents générés pour vérifier que leur comportement respecte la loi correspondante.

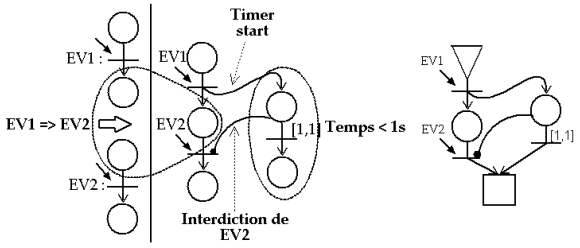


Figure 3. *Génération du réseau de Petri pour L1 - Représentation du réseau final.*

5. Travaux Similaires

Dans les travaux de M.S.Feather [FEA 98] il est également question de comparer l'exécution d'un SMA à ses spécifications. Dans cette approche, un unique monitor se charge de récupérer les événements envoyés par les agents et un réconciliateur va, en cas de violation, non pas remettre le système dans un état tel qu'il respecterait ses spécifications, mais modifier les spécifications pour qu'elles soient en accord avec le nouveau comportement du système. Ici, il n'est pas question de spécifications primordiales pour le bon fonctionnement du système, les auteurs ne cherchent donc pas à empêcher les mauvais comportements mais à faire en sorte que le système et les spécifications s'adaptent pour rester en accord, tout au long de l'exécution.

D.N.Lam et K.S.Barber [LAM 04] propose une méthodologie, la "Tracing Method" pour tester et expliquer le comportement des agents. L'objectif de cette méthodologie est de s'assurer qu'un agent exécute une action pour de bonnes raisons et si une action inattendue apparaît, d'aider à expliquer pourquoi l'agent a décider d'exécuter cette action. Les similitudes avec notre approche viennent de l'ontologie d'agent que les auteurs proposent pour pouvoir facilement comparer des modèles de spécifications (diagramme d'état, de séquences...) au comportement réel des agents. Notre approche se distingue de la Tracing Method du fait de l'automatisation que nous proposons au niveau de l'instrumentation du code des agents, mais aussi au niveau de la détection des incohérences entre le comportement attendu et le comportement observé. Enfin, notre approche consiste à embarquer le contrôle au sein des agents pour permettre une détection dynamique des erreurs de comportement et non une étude post-mortem des traces de programme.

Enfin nous citerons les récents travaux de R.Paes [PAE 05]. Dans le cadre des systèmes multiagents ouverts, les auteurs proposent d'utiliser des lois pour contrôler l'émergence de mauvais comportements. Si les idées de base s'avèrent être les mêmes, le contrôle qu'ils appliquent se fait sur les messages échangés entre les agents et non sur le comportement des agents dans sa globalité. Ils proposent de mettre en place un médiateur qui reçoit les messages, applique les lois sur ces messages et les transmet au destinataire. Ici, il est plus spécifiquement question de surveiller les interactions entre les agents à l'aide d'une entité extérieure.

6. Conclusion

Nous avons présenter le principe de contrôle d'agent autonome. En particulier, nous avons décrit les mécanismes de la génération automatique d'agent en mesure d'effectuer la vérification de leur propre comportement. Cette vérification s'effectue à l'aide de spécification de lois que les agents doivent respecter tout au long de leur exécution. L'intérêt de notre approche tient en la possibilité de définir des lois à un niveau permettant de les appliquer à des agents construits sur des modèles différents et ce par une personne extérieure à l'implémentation du système. Un autre point réside dans la génération automatique des observateurs associés aux agents permettant

à ces derniers de surveiller leur propre comportement. Nous proposons un framework fournissant les moyens nécessaires pour générer des agents auto-contrôlés : un langage pour décrire les lois, un ensemble de concepts à partir desquels il est possible de décrire l'application sous contrôle et un mécanisme de génération automatique du contrôle à appliquer aux agents. La prochaine étape de nos travaux portera sur l'application de lois concernant plusieurs agents.

7. Bibliographie

[BAR 99] BARBER K., MARTIN C., « Agent Autonomy : Specification, Measurement and Dynamic Adjustment », *Proc. of the Autonomy Control Software workshop at Autonomous Agents'99*, May 1999, p. 8–15.

[DIA 94] DIAZ M., JUANOLE G., COURTIAT J.-P., « Observer-A Concept for Formal On-Line Validation of Distributed Systems », *IEEE Trans. Softw. Eng.*, vol. 20, n° 12, 1994, p. 900–913, IEEE Press.

[FEA 98] FEATHER M., FICKAS S., VAN LAMSWEERDE A., PONSARD C., « Reconciling System Requirements and Runtime Behavior », *Proceedings of IWSSD9*, Isobe, Japan, 1998.

[HUA 95] HUANG Y., KINTALA C., « Software Fault Tolerance in the Application Layer », *Software Fault Tolerance*, 1995.

[LAM 04] LAM D., BARBER K., « Debugging Agent Behavior in an Implemented Agent System », *Proceedings of PROMAS'04*, New York City, July 20 2004, p. 45–56.

[LIA 92] LIAO Y., COHEN D., « A Specificational Approach to High Level Program Monitoring and Measuring », *IEEE Trans. Software Engineering*, vol. 18, n° 11, 1992.

[LUM 90] LUMPP J., CASAVANT T., SIEGLE H., MARINESCU D., « Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems », *Proceedings of the 10th International Conference on Distributed Systems*, June 1990, p. 476–483.

[MAN 95] MANSOURI-SAMANI M., « Monitoring of Distributed Sytems », PhD thesis, University of London, London, UK, 1995.

[PAE 05] PAES R., GUSTAVO CARVALHO, CARLOS LUCENA, ALENCAR P., ALMEIDA H., SILVA V., « Specifying Laws in Open Multi-Agent Systems », *Agents, Norms and Institutions for Regulated Multiagent Systems - ANIREM*, Utrecht, July 2005.

[SCH 99] SCHNOEBELEN P., *Vérification de Logiciels : Techniques et Outils du Model-Checking*, Vuibert édition, 1999.

[SOU 02] DE SOUSA DIAS M., RICHARDSON D., « Issues on Software Monitoring », rapport, July 2002, Department of Information and Computer Science, University of California.

[V´ 04] VÁZQUEZ-SALCEDA J., ALDEWERLD H., DIGNUM F., « Implementing Norms in Multiagent Systems », *Proceedings of MATES'04*, Erfurt, Germany, September, 29–30 2004.

[WAM 03] WAMPLER D., « The future of aspect oriented programming », 2003, White Paper, available on *http ://www.aspectprogramming.com*.

[WRI 51] VON WRIGHT G., « Deontic Logic », *Mind*, vol. 60, n° 237, 1951, p. 1–15.