Publié dans les Actes des Troisièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents, pp. 3–14, Chambéry-St-Badolph, France, mars, 1995.

FONDEMENTS D'UNE APPROCHE LOGIQUE A LA PROGRAMMATION D'AGENTS*

Yves Lespérance, Hector J. Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter et Richard B. Scherl†

Department of Computer Science
University of Toronto
Toronto, ON, M5S 1A4 Canada
{lesperan,hector,fl,marcu,reiter,scherl}@cs.toronto.edu

Résumé

Cette communication présente une approche à la programmation haut-niveau d'agents basée sur une théorie logique de l'action hautement développée. L'usager fournit une spécification des actions primitives des agents (préconditions et effets) de même que des aspects pertinents de leur environnement dans un formalisme logique. Il peut ensuite spécifier des comportements pour les agents dans un langage de programmation où l'on peut faire référence aux conditions en vigueur dans l'environnement. Lorsqu'une implantation des actions primitives est fournie, l'exécution des programmes dans un environnement réel est possible; autrement on peut quand même faire une exécution simulée. L'interpréteur maintient automatiquement un modèle du monde pour chacun des agents. La théorie sur laquelle l'approche est basée inclus une solution au problème de frame, permet aux agents d'avoir une connaissance incomplète du monde, et traite les actions de perception. Ce fondement théorique permet aussi la vérification des programmes. L'approche est présentée à l'aide d'un exemple d'application en cours de développement — un système multi-agent d'aide à l'organisation de réunions. Les travaux en cours sur l'implantation et les nombreux problèmes théoriques qui demeurent sont aussi décrits.

1 Introduction

La notion d'*agent* informatique est en vogue en ce moment (voir [16] pour une collection d'articles récents). Ce genre d'agent semble être une bonne façon de livrer des services aux usagers de façon conviviale sur les systèmes informatiques en réseau. Des application typiques sont la recherche d'information sur l'Internet, l'automation d'activités communes des usagers (ex. organisation de réunions), les interfaces à l'usager intelligents, l'intégration d'outils logiciels hétérogènes, la robotique intelligente, la modélisation de processus industriels ou organisationnels, etc. Le terme ≪ agent ≫ est utilisé de façons très différentes; il convient donc de tenter de clarifier le sens que nous lui donnons.

*Cette recherche est supportée financièrement par le Centre de recherche sur la technologie de l'information (Ontario, Canada), l'Institut pour la robotique et les systèmes intelligents (Canada) et le Conseil de recherche en sciences naturelles et en génie (Canada). Levesque et Reiter sont membres de l'Institut canadien pour la recherche avancée.

†Addresse courante: Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102 USA

Pour nous, un agent est toute entité active dont le comportement est utilement décrit à l'aide de notions mentales telles que les connaissances, les buts, les intentions, etc. (Ceci corresponds largement à l'usage commun en intelligence artificielle (IA), par opposition à la notion plus courante de scripts qui peuvent être exécutés sur des machines distantes.) De plus, nous porterons notre attention plus spécifiquement sur l'approche au développement d'applications qui consiste à concevoir un système comme une collection d'agents en interaction.

On peut voir la programmation orientée agent [19] comme une généralisation de la programmation orientée objet. Mais la notion d'agent est beaucoup plus complexe que celle d'objet. Conséquemment, il est essentiel de baser des outils pour la modélisation et la conception d'agents sur des fondements théoriques solides. Depuis quelque temps déjà, nous travaillons sur une théorie logique des agents et sur des outils de programmation basés sur celle-ci. Le cadre théorique comprends une formalisation de l'action qui incorpore une solution au problème de frame, débarrassant ainsi le concepteur de l'obligation de spécifier les aspects du monde qui restent inchangés à la suite d'une action. Le traitement comprends aussi un modèle des connaissances des agents et de leur capacité d'acquérir des connaissances nouvelles en accomplissant des actions perceptuelles, par exemple en inspectant leur environnement à l'aide d'une caméra ou de capteurs sonar, ou en entrant en interaction avec un usager ou un autre agent. Finalement, nous avons aussi un traitement des actions et processus complexes qui hérite de la solution au problème de frame pour les actions simples. La théorie possède des propriétés computationnelles intéressantes.

L'ensemble d'expressions d'actions complexes défini peut être vu comme un langage de programmation d'agents. Lorsqu'une spécification des actions de base des agents (préconditions et effets) est fournie, le concepteur peut spécifier des comportement complexes pour les agents dans ce langage de programmation. Cette spécification peut se faire à un très haut niveau d'abstraction car le langage permet de référer aux conditions en vigueur dans l'environnement. Lorsqu'une implantation des actions primitives est fournie, l'exécution des programmes dans un environnement réel est possible; autrement une exécution simulée peut être faite. L'interpréteur maintient automatiquement un modèle du monde pour les agents sur la base de la spécification donnée.

Jusqu'à maintenant, nos travaux sur la théorie et l'implantation d'agents avaient surtout porté sur des agents uniques, principalement pour des applications en robotique. Dans cette communication, nous proposons une généralisation de notre traitement aux systèmes multi-agents. Le traitement proposé est toutefois préliminaire et nous identifions plusieurs problèmes qui restent à résoudre avant d'obtenir une théorie et une implantation tout à fait satisfaisantes.

L'approche est présentée à l'aide d'un exemple — un système multi-agent d'aide à l'organisation de réunions. Chaque usager a un agent ≪ secrétaire ≫ qui a une certaine connaissance de son agenda. Lorsqu'un usager veut organiser une réunion, il crée un nouvel agent ≪ organisateur ≫ qui contacte les agents secrétaires des participants et tente d'obtenir leur accord sur une période de temps précise. L'exemple est une version très simplifiée d'un tel système. Pour être vraiment utile, un agent secrétaire devrait avoir une certaine connaissance des préférences de l'usager au niveau des temps de réunion, savoir quand et comment communiquer avec lui lorsqu'une requête est reçue, peut-être connaître la structure hiérarchique de l'organisation, etc. Les agents organisateurs devraient avoir des stratégies d'ordonnancement et de négociation robustes. Notre formalisation de l'application inclus un module générique de communication entre agents utilisable pour d'autres applications. Chaque agent a une file de messages et des actes de communication abstraits sont définis (ex. $\text{INFORME}(agent, \phi)$, $\text{REQUÊTE}(agent, action)$, $\text{QUESTIONSI}(agent, \phi)$, etc.).

Dans la prochaine section, nous faisons un bref compte rendu de notre théorie de l'action. Puis, nous présentons notre traitement des actions complexes et expliquons comment il peut être vu comme un langage de programmation d'agents. La section 4 développe un ensemble d'outils simples pour la communication entre agents et la section 5 complète notre spécification de l'application d'organisation de réunions. Dans les sections subséquentes, nous discutons diverses questions d'architecture soulevées par l'implantation de notre formalisme et décrivons le statut de l'implantation et les expériences d'utilisation en cours. Nous terminons en résumant les caractéristiques importantes de notre approche et en discutant les problèmes qui restent à régler.

2 Une théorie de l'action

Notre approche est basée sur une version généralisée du calcul situationnel [12], un dialecte du calcul des prédicats qui permet de représenter un monde en évolution. Le monde est conçu comme se trouvant dans un certain état (ou situation). Cet état ne peut changer que sous l'effet d'une action accomplie par un agent. Le terme $do(agt, act, s)$ représente l'état qui résulte de l'accomplissement de l'action act par l'agent agt dans l'état s. Par exemple, la formule $\text{SUR}(\text{B}_1, \text{B}_2, do(\text{AGT}, \text{DÉPOSESUR}(\text{B}_1, \text{B}_2), s))$ pourrait signifier que B_1 est sur B_2 dans l'état qui suit l'accomplissement de l'action DÉPOSESUR par AGT dans l'état s. Les prédicats et fonctions dont la valeur peut changer d'état en état (et dont le dernier argument est un état) sont appelés *fluents*.

On spécifie une action en énonçant tout d'abord les conditions dans lesquelles elle peut être accomplie au moyen d'un *axiome de préconditions*. Par exemple,

$$\begin{aligned} &Poss(agt, \text{AJOUTEACTIVITÉ}(usager, période, activité, responsable), s) \equiv \\ &\quad agt = \text{SECRÉTAIRE}(usager) \land \\ &\quad \neg\exists activité_1, responsable_1\ \text{AGENDA}(usager, période, activité_1, responsable_1, s) \end{aligned} \tag{1}$$

signifie qu'un agent peut ajouter une activité à l'agenda d'un usager ssi il est le secrétaire de cet usager et aucune activité n'est prévue pour cette période. Puis, on spécifie comment l'action affecte l'état du monde au moyen *d'axiomes d'effets*, par exemple:

$$\begin{aligned} &Poss(agt, \text{AJOUTEACTIVITÉ}(usager, période, activité, responsable), s) \supset \\ &\quad \text{AGENDA}(usager, période, activité, responsable, \\ &\qquad do(agt, \text{AJOUTEACTIVITÉ}(usager, période, activité, responsable), s)) \end{aligned} \tag{2}$$

Ces axiomes ne sont cependant pas suffisants pour le raisonnement sur le changement. Il est normalement nécessaire de leur adjoindre des axiomes de frame spécifiant les propriétés qui demeurent inchangées sous l'effet de l'action. Le notoire problème de frame [12] tient au fait que le nombre de ces axiomes est de l'ordre du produit du nombre d'actions et du nombre de fluents. Notre approche incorpore un traitement du problème de frame dû à Reiter [15] (inspiré de travaux antérieurs de Pednault [14], Schubert [18] et Haas [4]). L'idée de base est de réunir les axiomes d'effets pertinents à un fluent donné et de supposer que ceux-ci spécifient toutes les façons dont le fluent pourrait changer de valeur. Une transformation syntactique permet alors obtenir un *axiome d'état successeur* pour le fluent, par exemple:

$$\begin{array}{l} Poss(agt, act, s) \supset \\ [\text{AGENDA}(agt', période, activ, resp, do(agt, act, s)) \equiv \\ \ act = \text{AJOUTEACTIVITÉ}(agt', période, activ, resp) \vee \\ \ \text{AGENDA}(agt', période, activ, resp, s) \wedge act \neq \text{ENLÈVEACTIVITÉ}(agt', période)]. \end{array} \tag{3}$$

Ceci dit qu'une activité est à l'agenda d'un usager dans l'état suivant l'accomplissement de l'action act par agt dans l'état s ssi l'action est d'ajouter cette activité à l'agenda ou bien si l'activité était déjà à l'agenda de l'usager et l'action n'est pas de l'enlever. Ce traitement évite la prolifération du nombre d'axiomes car il ne nécessite qu'un axiome d'état successeur par fluent et un axiome de préconditions par action.[1]

Scherl et Levesque [17] ont généralisé ce traitement aux actions de perception. Ces actions ont des effets sur l'état mental de l'agent et non sur le monde extérieur. Par exemple, après avoir accompli l'action INSPECTEMESS, un agent sait si sa file de messages est vide et lorsqu'elle ne l'est pas, sait aussi quel message se trouve en tête de file:

$$\begin{array}{l} Poss(agt, \text{INSPECTEMESS}, s) \supset \\ \quad \mathbf{KWhether}(agt, \text{VIDE}(\text{FILEMESS}(agt)), do(agt, \text{INSPECTEMESS}, s)) \wedge \\ \quad [\neg\text{VIDE}(\text{FILEMESS}(agt, s)) \supset \\ \quad \exists m \mathbf{Know}(agt, \text{TÊTE}(\text{FILEMESS}(agt)) = m, do(agt, \text{INSPECTEMESS}, s))] \end{array}$$

KWhether(agt, ϕ, s) tient lieu de **Know**$(agt, \phi, s) \vee$ **Know**$(agt, \neg\phi, s)$, et **Know**(agt, ϕ, s) est aussi une abréviation que nous définissons ci-dessous. La connaissance est représentée en adaptant le modèle des monde possibles au calcul situationnel (à la façon de Moore [13]). $K(agt, s', s)$ représente le fait que dans l'état s, l'agent agt croit que l'état du monde pourrait être s'. **Know**(agt, ϕ, s) est une abréviation de la formule $\forall s'(K(agt, s', s) \supset \phi(s'))$. Pour plus de clarté, nous employons parfois la pseudo-variable now pour représenter l'état lié par le **Know**; ainsi **Know**$(agt, \phi(\mathsf{now}), s)$ abrévie $\forall s'(K(agt, s', s) \supset \phi(s'))$.

Scherl et Levesque montrent comment obtenir un axiome d'état successeur pour le fluent K dans le cas d'un agent unique. Dans les domaines multi-agents, le problème est plus complexe car un agent n'est pas nécessairement au courant des actions des autres agents (actions exogènes). Pour l'application d'organisation de réunions, l'axiome d'état successeur qui suit est suffisant. Notez que $s < s'$ ssi s' est le résultat de l'accomplissement d'une séquence d'actions dans l'état s et chacune des

[1] La présente discussion ignore les problèmes de ramification et de qualification; un traitement compatible avec notre approche est proposé par Lin et Reiter [10].

actions de la séquence est possible dans l'état où elle a lieu; $s \leq s'$ tient lieu de $s < s' \vee s = s'$.

$$
\begin{array}{l}
Poss(acteur, act, s) \supset \\
[K(sujet, s'', do(acteur, act, s)) \equiv \\
\ \exists s' (K(sujet, s', s) \wedge \\
\quad (acteur \neq sujet \supset Exo(sujet, s', s'')) \wedge \\
\quad (acteur = sujet \supset \exists s^* (Exo(sujet, s', s^*) \wedge \\
\qquad s'' = do(sujet, act, s^*) \wedge Poss(sujet, act, s^*) \wedge \\
\qquad [act = \text{INSPECTEMESS} \supset \\
\qquad\quad (\text{VIDE}(\text{FILEMESS}(sujet, s^*)) \equiv \text{VIDE}(\text{FILEMESS}(sujet, s))) \wedge \\
\qquad\quad (\neg \text{VIDE}(\text{FILEMESS}(sujet, s^*)) \supset \\
\qquad\quad\ \text{TÊTE}(\text{FILEMESS}(sujet, s^*)) = \text{TÊTE}(\text{FILEMESS}(sujet, s)))])))] \\
\text{where } Exo(agt, s, s') \overset{def}{=} s \leq s' \wedge \\
\qquad\qquad\qquad \forall s^* \forall agt' \forall act (s < do(agt', act, s^*) \leq s' \supset agt' \neq agt)
\end{array}
\tag{4}
$$

Voici ce que dit cet axiome. Quand une action act est accomplie par un agent $acteur$ autre que le $sujet$, l'axiome dit que dans la situation résultante, $sujet$ considère possible toute situation qui résulte d'un nombre arbitraire d'actions exogènes se produisant dans une situation qui était considérée comme possible. Donc, $sujet$ admet la possibilité d'un nombre arbitraire d'actions exogènes et perd toutes les connaissances qu'il pouvait avoir sur des fluents qui pourraient être affectés par ces actions exogènes.

Lorsque act est une action non-perceptuelle (ex. AJOUTEACTIVITÉ(u, p, a, o)) accomplie par $sujet$ lui-même, l'axiome énonce que dans la situation résultante, $sujet$ considère possible toute situation produite par son accomplissement de act précédé d'un nombre arbitraire d'actions exogènes se produisant dans une situation qui était considérée comme possible. Ainsi, $sujet$ apprends qu'il vient d'accomplir act, mais perds toutes les connaissances qu'il pouvait avoir sur des fluents qui pourraient être affectés par les actions exogènes.

Finalement, lorsque act est une action perceptuelle accomplie par $sujet$ (c'est à dire INSPECTEMESS), nous obtenons la même chose que précédemment plus le fait que l'agent apprends la valeur des fluents associés à l'action perceptuelle — dans le cas qui nous concerne, si sa file de messages est vide, et si tel n'est pas le cas, quel est le message en tête de file.[2] Ceci se généralise facilement à un nombre arbitraire d'actions perceptuelles.

En général, permettre un nombre arbitraire d'actions exogènes à chaque transition comme nous le faisons ici ne laisserait sans doute que trop peu de connaissances aux agents. Mais notre domaine est nettement partitionné: l'agenda d'un usager ne peut être mis à jour que par l'agent secrétaire de celui-ci. Donc malgré tout, les secrétaires connaissent toujours l'agenda de leur usager. Dans d'autres circonstances, il pourrait être approprié de supposer que toutes les actions sont publiques,

[2] Pour voir comment on obtient ceci, notez que nous exigeons que $\text{VIDE}(\text{FILEMESS}(sujet, s^*)) \equiv \text{VIDE}(\text{FILEMESS}(sujet, s))$. Ainsi, $\text{VIDE}(\text{FILEMESS}(sujet))$ aura la même valeur de vérité dans tous les s^* tels que $K(sujet, do(sujet, \text{INSPECTEMESS}, s^*), do(sujet, \text{INSPECTEMESS}, s))$. On notera alors que pour toute situation s, $\text{VIDE}(\text{FILEMESS}(sujet))$ est vrai à $do(sujet, \text{INSPECTEMESS}, s)$ ssi il est vrai à s. Donc, $\text{VIDE}(\text{FILEMESS}(sujet))$ a la même valeur de vérité dans toutes les situations s'' telles que $K(sujet, s'', do(sujet, \text{INSPECTEMESS}, s))$, et on a **KWhether**$(sujet, \text{VIDE}(\text{FILEMESS}(sujet, \mathsf{now})), do(sujet, \text{INSPECTEMESS}, s))$.

c'est à dire connues de tous les agents; ce cas est similaire à celui d'un agent unique et s'axiomatise de façon analogue. Dans d'autres cas, il semble préférable pour les agents de supposer qu'aucun événement exogène ne se produit et de réviser leurs croyances lorsqu'une contradiction apparaît; une formalisation de cette approche est en cours.

3 Actions complexes et CONGOLOG

Les actions traitées jusqu'ici sont primitives et déterministes. Toute approche à la programmation d'agent requiert des actions complexes. De plus, comme nous voulons modéliser les systèmes multi-agents, nous devons aussi traiter les processus concurrents.

Nous introduisons ce type d'action et processus dans le formalisme en tant qu'abréviations (macros). La formalisation résultante est beaucoup plus simple que les alternatives et les actions complexes héritent automatiquement de la solution au problème de frame. Nous définirons donc une abréviation $Do(\delta, s, s')$ où δ est une expression d'action/processus complexe, qui signifie que l'accomplissement de δ dans l'état s mène à l'état (pas nécessairement unique) s'. Il est relativement simple de donner une définition inductive de $Do(\delta, s, s')$ lorsque δ ne contient pas d'actions concurrentes, ce que nous faisons dans [5]; toutefois cette approche simple ne se généralise pas aux actions concurrentes.

Ici, nous développerons une approche différente qui traite les actions concurrentes par entrelacement. Nous dirons qu'une exécution concurrente de δ_1 et δ_2, représentée par $(\delta_1 \parallel \delta_2)$, a lieu entre s et s' ssi un entrelacement quelconque de δ_1 et δ_2 a lieu entre s et s'. Il nous faut donc capturer l'idée d'une exécution d'un programme δ sur un intervalle $[s, s']$, tout en permettant la présence d'autres actions dans l'intervalle. Do est définie par induction sur la structure de son premier argument comme suit:

- *Actions primitives:*

$$Do(\mathbf{by}(agt, act), s, s') \stackrel{def}{=} \exists s^*(s < do(agt, act, s^*) \leq s').$$

- *Actions de test:*

$$Do(\phi?, s, s') \stackrel{def}{=} \exists s^*(s \leq s^* \leq s' \wedge \phi(s^*)).$$

- *Séquences:*

$$Do([\delta_1; \delta_2], s, s') \stackrel{def}{=} \exists s^*(Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s')).$$

- *Actions concurrentes:*

$$Do((\delta_1 \parallel \delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \wedge Do(\delta_2, s, s').$$

- *Choix d'action non-déterministe:*

$$Do((\delta_1 \mid \delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s').$$

- *Choix d'argument non-déterministe:*

$$Do(\pi x\, \delta(x), s, s') \stackrel{def}{=} \exists x\, Do(\delta(x), s, s').$$

- *Itération non-déterministe:*

$$\begin{aligned} Do(\delta^*, s, s') \stackrel{def}{=} \forall P\{ & \\ & \forall s_1[P(s_1, s_1)] \quad \wedge \\ & \forall s_1, s_2, s_3[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \ \supset P(s_1, s_3)]\,\} \\ & \qquad \supset \ P(s, s'). \end{aligned}$$

Cette définition a toutefois un léger problème: elle ne garantit pas que toutes les actions ayant eu lieu dans l'intervalle sont le résultat de l'exécution d'une partie du programme. Dans [7], nous présentons une version plus complexe qui solutionne ce problème. Cette formalisation s'inspire du traitement standard des actions concurrentes en logique dynamique [3].

Les conditionnelles et les boucles peuvent être définis en termes des expressions précédentes de la façon suivante:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{def}{=} [\phi?; \delta_1] | [\neg\phi?; \delta_2],$$

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \stackrel{def}{=} [[\phi?; \delta]^*; \neg\phi?].$$

Les procédures et la récursion peuvent aussi être traitées.

Pour plus de commodité, l'expression **by**(agt, δ) est employée avec toute action complexe δ, représentant alors δ avec chaque instance d'une action primitive act remplacée par **by**(agt, act); par exemple,

$$\begin{aligned} &\textbf{by}(agt, \textbf{while } \phi \textbf{ do } [act_1; act_2] \textbf{ endWhile}) \stackrel{def}{=} \\ &\quad \textbf{while } \phi \textbf{ do } [\textbf{by}(agt, act_1); \textbf{by}(agt, act_2)] \textbf{ endWhile}. \end{aligned}$$

Un traitement des interruptions et des processus concurrents à priorités différentes a aussi été développé. De plus, nous expérimentons avec diverses techniques permettant de représenter et raisonner sur les actions de longue durée (ex. remplir une baignoire) pour lesquelles le modèle d'entrelacement est inapproprié

Cet ensemble d'expressions d'actions complexes forme un langage de programmation adapté aux applications en IA distribuée, robotique, et simulation. Nous appelons la portion séquentielle GOLOG (alGOl en LOGique), et le langage entier CONGOLOG (CONcurrent GOLOG). Les prochaines sections contiennent plusieurs exemples.

CONGOLOG se distingue des langages courants de plusieurs façons. Il a une sémantique formelle basée sur le calcul situationnel. Ses actions complexes se décomposent en actions primitives qui réfèrent normalement à des actions externes aux agents (ex. ramasser un objet ou questionner un autre agent, par opposition à des actions qui n'affectent que l'état interne de la machine comme une affectation à un registre). Ceci permet de spécifier les comportements désirés pour les agents à un très haut niveau d'abstraction. Leur évaluation se fait à l'aide d'un démonstrateur. L'usager fournit: les axiomes de préconditions, un par action, les axiomes d'état successeur, un par fluent, une spécification de l'état initial du monde, et un programme CONGOLOG spécifiant le comportement des agents du système. En gros, pour exécuter un programme, on pose le problème $Axiomes \models \exists s Do(programme, S_0, s)$ au démonstrateur, et si une preuve (constructive) est obtenue, on en extrait une affectation pour la variable $s = do(agt_n, act_n, \ldots do(agt_2, act_2, do(agt_1, act_1, S_0)) \ldots)$, puis on envoie la séquence d'action $\langle agt_1, act_1\rangle, \langle agt_2, act_2\rangle, \ldots, \langle agt_n, act_n\rangle$ au module d'exécution d'action primitives. Il est à noter que l'interpréteur CONGOLOG maintient automatiquement un modèle du monde pour les agents. Dans cette approche, l'emphase porte sur la programmation à haut niveau plutôt que sur la synthèse de plans au temps d'exécution. Mais les plans partiels sont admis, le non-déterminisme permettant souvent d'inférer les détails manquants. Par exemple, le plan **while** $\exists b\,$SURTABLE(b) **do** $\pi b\,$ENLEVE(b) laisse aux soins de l'interpréteur CONGOLOG de trouver une séquence d'actions légales qui permet de débarrasser la table.

Nos travaux se rapprochent de ceux de Shoham et son équipe sur le langage de programmation AGENT-0 [19]. Mais il n'est pas facile de faire une comparaison systématique car les différences entre CONGOLOG et AGENT-0 sont nombreuses. Ce dernier inclus un modèle des engagements et capacités des agents et un ensemble d'actions de communications simples; ses agents ont tous une architecture générique à base de règles; il y a aussi une horloge globale et les croyances sont toutes de propositions datées. Toutefois, il n'y a pas de mise à jour automatique des connaissances des agents sur la base d'une spécification des actions primitives comme en CONGOLOG et seulement quelques types d'actions complexes sont traités. Il semble aussi qu'il y ait moins d'emphase sur l'obtention d'une spécification formelle complète du système.

4 Communication en CONGOLOG

Les applications multi-agents nécessitent généralement des outils pour la communication entre agents. Un choix populaire est le langage de communication KQML [2] et les outils associés. Toutefois selon Cohen et Levesque [1], la définition de KQML souffre de nombreuses lacunes, en particulier l'absence de sémantique formelle. Comme un de nos objectifs est de démontrer l'utilité de CONGOLOG pour divers types de tâches, nous choisissons ici de définir nos propres outils de communications simples. On peut voir la spécification comme un module générique qu'on inclus dans des applications spécifiques. Nous procéderons comme suit: tout d'abord, un ensemble d'actions primitives d'échange de messages sera spécifié; par la suite, des actes de communications abstraits seront définis sur la base de ces primitives. Chaque agent agt a une file de messages dénotée par le fluent fonctionnel FILEMESS(agt, s). Trois types d'actions primitives opèrent sur ces files de messages:

- ENVOIEMESS$(destinataire, mess)$: envoyer le message $mess$ à l'agent $destinataire$, ce qui a pour effet d'ajouter la paire $\langle expéditeur, mess\rangle$ à la file du destinataire;
- INSPECTEMESS: percevoir quelle paire expéditeur-message est à la tête de sa file; et
- ENLÈVEMESS: enlever la paire expéditeur-message à la tête de sa file.

Les préconditions de ces actions sont les suivantes: ENLÈVEMESS est possible pour un agent ssi sa file de messages n'est pas vide:

$$Poss(agt, \text{ENLÈVEMESS}, s) \equiv \neg \text{VIDE}(\text{FILEMESS}(agt, s)); \tag{5}$$

ENVOIEMESS$(dest, mess)$ et INSPECTEMESS sont toujours possibles (les axiomes sont omis).

Les effets physiques de ces actions sont tels que décrits ci-dessus, ce qui donne l'axiome d'état successeur suivant pour le fluent FILEMESS:

$$
\begin{aligned}
&Poss(agt, act, s) \supset \\
&[\text{FILEMESS}(agt', do(agt, act, s)) = q \equiv \\
&\ \exists m(act = \text{ENVOIEMESS}(agt', m) \wedge q = \text{CONCATÈNE}(\text{FILEMESS}(agt', s), [\langle agt, m\rangle])) \\
&\ \vee act = \text{ENLÈVEMESS} \wedge agt = agt' \wedge q = \text{RESTE}(\text{FILEMESS}(agt', s)) \\
&\ \vee q = \text{FILEMESS}(agt', s) \wedge \\
&\qquad \forall m(act \neq \text{ENVOIEMESS}(agt', m)) \wedge (act \neq \text{ENLÈVEMESS} \vee agt \neq agt')].
\end{aligned}
\tag{6}
$$

Sur la base de ces primitives, nous pouvons maintenant définir des actes de communication abstraits utiles:

proc INFORME(agt, ϕ)
  Know(ϕ)?; ENVOIEMESS$(agt, \ulcorner\text{INFORME}(\phi)\urcorner)$
end

proc INFORMESI(agt, ϕ)
  INFORME(agt, ϕ) | INFORME$(agt, \neg\phi)$ | INFORME$(agt, \neg$**KWhether**$(\phi))$
end

proc REQUÊTE(agt, δ)
  ENVOIEMESS$(agt, \ulcorner\text{REQUÊTE}(\delta)\urcorner)$
end

proc QUESTIONSI(agt, ϕ)
  REQUÊTE$(agt,$ INFORMESI$(\mathsf{self}, \phi))$
end

Notez que la pseudo-variable self réfère à l'agent qui exécute le programme. Les définitions utilisent aussi la citation. Il y a plusieurs traitements bien connus de cet élément. Un traitement compatible avec notre théorie est en cours de développement.

On peut démontrer que l'acte de communication abstrait INFORME a les caractéristiques attendues. Définissons d'abord SINCÈRE$(expéditeur, destinataire, s)$, qui signifiera que jusqu'à l'état s, chaque message INFORME envoyé par $expéditeur$ à $destinataire$ fut envoyé sincèrement:

$$
\begin{aligned}
&\text{SINCÈRE}(exp, dest, s) \stackrel{def}{=} \\
&\quad \forall s'[do(exp, \text{ENVOIEMESS}(dest, \ulcorner\text{INFORME}(\phi)\urcorner), s') \leq s \supset \textbf{Know}(exp, \phi, s')].
\end{aligned}
$$

Il est alors possible de montrer que lorsqu'un agent a envoyé un message INFORME(ϕ) à quelqu'un et que le destinataire a inspecté ses messages, ce dernier saura alors qu'à un certain moment dans le passé, l'expéditeur savait que ϕ, à condition que le destinataire sache que sa file de messages est vide initialement et que l'expéditeur est sincère envers lui durant cette période:

Proposition

$\mathbf{Know}(dest, \text{VIDE}(\text{FILEMESS}(dest, \mathsf{now})), S_0) \wedge$
$\mathbf{Know}(dest, \text{SINCÈRE}(exp, dest, \mathsf{now}),$
$\quad do(dest, \text{INSPECTEMESS}, do(exp, \text{ENVOIEMESS}(dest, \ulcorner \text{INFORME}(\phi) \urcorner), S_0))) \supset$
$\mathbf{Know}(dest, \exists s'[s' \leq \mathsf{now} \wedge \mathbf{Know}(exp, \phi, s')],$
$\quad do(dest, \text{INSPECTEMESS}, do(exp, \text{ENVOIEMESS}(dest, \ulcorner \text{INFORME}(\phi) \urcorner), S_0)))$

Présentement, il est impossible de prouver des résultats généraux intéressants à propos de REQUÊTE à cause de l'absence de formalisation des buts et des intentions. Nous comptons remédier bientôt à cette lacune. La prochaine section montrera que les outils de communication spécifiés ci-dessus suffisent pour développer des applications intéressantes. Nous travaillons à étoffer cette spécification et l'étendre à d'autres types d'actes de communication.

5 L'organisation de réunions en CONGOLOG

Avant de pouvoir définir notre exemple de système d'organisation de réunions, nous devons d'abord compléter la spécification des actions primitives qui manipulent l'agenda des usagers. L'axiome de préconditions pour AJOUTEACTIVITÉ a déjà été donné (1). Quant à l'action d'enlever une activité de l'agenda d'un usager, nous la considérons possible ssi l'agent accomplissant l'action est le secrétaire de l'usager et une activité est présentement à l'agenda de l'usager pour la période:

$$\begin{aligned} &Poss(agt, \text{ENLÈVEACTIVITÉ}(usager, période), s) \equiv \\ &\qquad agt = \text{SECRÉTAIRE}(usager) \wedge \\ &\qquad \exists activ, resp\ \text{AGENDA}(usager, période, activ, resp, s) \end{aligned} \tag{7}$$

Les effets de ces actions sur le fluent AGENDA sont capturés par l'axiome d'état successeur donné plus tôt (3).

Nous sommes maintenant prêts à utiliser CONGOLOG pour définir le comportement de nos agents. Nous commençons par les agents ≪ organisateurs de réunions ≫. Ceux-ci exécuterons la procédure apparaissant à la figure 1. La procédure utilise l'abréviation suivante:

$$\begin{aligned} &\text{CONSENTIRÉUNION}(participant, requérant, période, responsable, s) \stackrel{def}{=} \\ &\quad \exists s', s''(s'' \leq s \wedge Do(\text{SECRÉTAIRE}(participant), [\\ &\qquad \text{TÊTE}(\text{FILEMESS}(\text{SECRÉTAIRE}(participant))) = \langle requérant, \ulcorner \text{REQUÊTE}(\\ &\qquad\qquad \text{AJOUTEACTIVITÉ}(participant, période, \text{RÉUNION}, responsable) \urcorner \rangle ?; \\ &\qquad \text{INSPECTEMESS}; \\ &\qquad \text{AJOUTEACTIVITÉ}(participant, période, \text{RÉUNION}, responsable)], \\ &\quad s', s'')). \end{aligned}$$

Ainsi, nous considérons un participant comme ayant donné son accord à une requête de réunion ssi dans une situation antérieure, le secrétaire du participant a perçu la requête d'ajouter la réunion à son agenda et a alors immédiatement donné suite à la requête.[3]

[3] Ceci est une façon un peu simpliste de modéliser le consentement à une requête. On pourrait parler de la plus récente

proc ORGANISERÉUNION$(responsable, Participant, période)$
  for $p : Participant(p)$ **do**
    REQUETE(SECRÉTAIRE(p), AJOUTEACTIVITÉ$(p, période$, RÉUNION, $responsable))$;
    QUESTIONSI(SECRÉTAIRE(p), CONSENTIRÉUNION$(p$, self, $période, responsable))$
  endFor;
  while $\neg$**KWhether**(self, $\forall p[Participant(p) \supset$
          CONSENTIRÉUNION$(p$, self, $période, responsable)]$, now) **do**
    INSPECTEMESS;
    if $\neg$VIDE(FILEMESS(self)) **then** ENLÈVEMESS **endIf**
  endWhile;
  INFORMESI$(responsable,$
      $\forall p[Participant(p) \supset$ CONSENTIRÉUNION$(p$, self, $période, responsable)])$;
  if $\neg\forall p[Participant(p) \supset$ CONSENTIRÉUNION$(p$, self, $période, responsable)]$ **then**
    % libérer participants de leurs engagements
    for $p : Participant(p)$ **do**
      REQUETE(SECRÉTAIRE(p), ENLÈVEACTIVITÉ$(p, période))$
    endFor
  endIf
endProc

Figure 1: Procédure exécutée par les agents ≪ organisateurs de réunions ≫.

Pendant ce temps, les agents ≪ secrétaires ≫ exécutent la procédure apparaissant à la figure 2. Notez que la procédure ne traite pas le cas où un usager voudrait se libérer d'un engagement.

Pour mettre en oeuvre un système d'organisation de réunions, on pourrait par exemple, soumettre à l'interpréteur CONGOLOG le programme suivant:

by(SECRÉTAIRE$_1$, GÈREAGENDA(USAGER$_1$)) $\|$
by(SECRÉTAIRE$_2$, GÈREAGENDA(USAGER$_2$)) $\|$
by(SECRÉTAIRE$_3$, GÈREAGENDA(USAGER$_3$)) $\|$
by(ORGANISATEUR$_1$, ORGANISERÉUNION(USAGER$_1$, {USAGER$_1$, USAGER$_3$}, MIDI)) $\|$
by(ORGANISATEUR$_2$, ORGANISERÉUNION(USAGER$_2$, {USAGER$_2$, USAGER$_3$}, MIDI)).

Ici, les organisateurs tenteront tous deux d'obtenir l'accord d'USAGER$_3$ pour une réunion à midi; il y aura donc deux types de séquences d'exécution, dépendant de qui obtient cet accord.

instance de la requête perçue, et si on avait un modèle des intentions, on pourrait éviter d'exiger une réaction immédiate de l'agent recevant la requête.

```
proc GÈREAGENDA(usager)
   while True do
      INSPECTEMESS;
      if ¬VIDE(FILEMESS(self)) then
         % si message est INFORME(φ), rien à faire
         if TÊTE(FILEMESS(self)) = ⟨interrogateur, ⌜QUESTIONSI(φ)⌝⟩ then
            INFORMESI(interrogateur, φ) endIf;
         if TÊTE(FILEMESS(self)) = ⟨requérant,
                        ⌜REQUETE(AJOUTEACTIVITÉ(usager, période, activ, resp))⌝⟩
             ∧ AUTORISÉAJOUACTIV(resp)
             ∧ Poss(self, AJOUTEACTIVITÉ(usager, période, activ, resp), now)
            then AJOUTEACTIVITÉ(usager, période, activ, resp) endIf;
         if TÊTE(FILEMESS(self)) = ⟨requérant, ⌜REQUETE(ENLÈVEACTIVITÉ(usager, période))⌝⟩
             ∧ Poss(self, ENLÈVEACTIVITÉ(usager, période), now)
             ∧ ∃activ AGENDA(usager, période, activ, PROPRIÉTAIRE(requérant), now)
            then ENLÈVEACTIVITÉ(usager, période) endIf;
         ENLÈVEMESS
      endIf
   endWhile
endProc
```

Figure 2: Procédure exécutée par les agents ≪ secrétaires ≫.

6 Questions d'architecture

CONGOLOG doit maintenir un modèle du monde pour chacun des agents, de façon à pouvoir évaluer les tests contenus dans leurs programmes. Il peut aussi être nécessaire de modéliser certains aspects de l'environnement des agents, par exemple pour une exécution simulée. Dans nos travaux antérieurs, nous avons identifié diverses techniques intéressantes pour effectuer ce raisonnement. Dans [15, 17], nous proposons une méthode pour établir si une proposition donné (requête) est vraie après une séquence d'actions. La méthode applique une forme de régression à la requête donnée pour obtenir une requête équivalente qui ne mentionne que la situation initiale; la requête résultante peut alors être traitée par une méthode de démonstration non-temporelle ordinaire. L'approche est plus efficace que la démonstration directe et est très générale; elle traite les cas de connaissances incomplètes de la situation initiale, de même que les actions à effets dépendants du contexte. Elle a été prouvée adéquate et complète. La régression peut aussi être employée pour incorporer les informations acquises par perception au modèle du monde: les informations acquises sont régressées jusqu'à ce qu'elles ne mentionnent que la situation initiale et le résultat est ajouté à la théorie de la situation initiale.

Un problème associé à la méthode de régression est que son coût augmente à mesure que l'on s'éloigne de la situation initiale. Il est clair qu'en général, on doit effectuer une mise à jour des connaissances des agents avec l'évolution du système. Cependant, ainsi que Lin et Reiter [8, 9] l'ont montré, la progression d'une base de connaissances (BC) n'est pas toujours représentable en logique du premier ordre. Ils identifient malgré tout certains cas spéciaux intéressants où la progression d'une BC est représentable en logique du premier ordre et facilement calculable, en particulier, le cas des BCs où les énoncés dépendants de l'état sont uniquement des littéraux et où les actions n'ont pas d'effets dépendants du contexte. Notez que l'hypothèse du monde clos n'est pas requise et que les connaissances des agents peuvent donc être incomplètes.

Les actions exogènes sont problématiques pour ces deux techniques, car alors, l'agent ne sait pas quelle séquence d'action a eu lieu. Notre implantation traite présentement certains de ces cas en se rabattant sur la perception lorsque la régression est impossible. Une idée intéressante serait de précalculer les effets de séquences arbitraires d'actions exogènes sur les fluents. Des travaux sont en cours pour obtenir un traitement général.

Nous travaillons aussi a généraliser la méthode de progression. Dans certains cas, on pourrait peut-être partitionner la BC et traiter certains fluents par régression et d'autres par progression. Les agents pourraient aussi tenter de réduire leurs BC à la forme requise pour la progression en accomplissant des actions de perception. Finalement, dans certains cas, il pourrait être nécessaire de se contenter d'une approximation facilement calculable à la progression (c'est à dire, quelque chose de plus faible que la postcondition la plus forte de la BC pour l'action).

Le non-déterminisme est un autre aspect du langage qui soulève des questions. Présentement, notre implantation explore l'espace des actions possibles (profondeur d'abord) jusqu'à ce qu'une action perceptuelle (ou la fin du programme) soit atteinte avant de faire un choix d'action. Il est clair que ceci n'est pas toujours la meilleure stratégie. Peut-être vaudrait-il mieux laisser au programmeur le soin d'indiquer jusqu'où la recherche doit être faite à l'aide d'une primitive de contrôle similaire au ≪ ! ≫ de PROLOG? De plus, ne pourrait-on pas le cas échéant poursuivre la recherche même au delà des action perceptuelles en considérant les divers résultats possibles?

Une spécification formelle du processus d'interprétation de CONGOLOG pour les programmes multi-agents est en cours de développement. Une telle spécification faciliterait l'évaluation de différentes implantations. Notre formalisation de la notion de ≪ capacité d'un agent à exécuter un programme ≫ [6], qui traite les préconditions de connaissances des actions, représente un pas important dans cette direction.

7 Implantation et expérimentation

Un interpréteur GOLOG prototype a été implantés en PROLOG et des expériences impliquant divers types d'applications sont en cours. La plus avancée concerne une application en robotique, la livraison de courrier dans un environnement simple [5]. Le contrôleur de haut niveau du robot programmé en GOLOG est interfacé à un logiciel robotique accomplissant les tâches de base. Le système fonctionne présentement en mode simulation; des expériences avec un véritable robot sont projetées.

Deux versions de l'interpréteur CONGOLOG sont aussi en cours d'implantation. Une version monolithique supporte l'exécution concurrente d'agents sur un seul processeur. L'autre version supporte l'exécution véritablement distribuée d'agents, la communication entre agents étant implantée au moyen d'outils TCP/IP [11]. L'application d'organisation de réunions est en cours d'implantation.

8 Conclusion

Nous croyons que le manque de robustesse des systèmes intelligents présentement disponibles tient en bonne partie à l'absence de spécification adéquate de la tâche à accomplir. En conséquence, nous tentons de développer des outils ayant une base théorique solide. Les résultats obtenus jusqu'à maintenant représentent un pas important vers cet objectif. Toutefois, il reste beaucoup à faire, autant sur les questions théoriques que l'implantation. Tel que mentionné précédemment, nous étudions diverses approches pour implanter le mécanisme de raisonnement sur le changement et les effets des actions perceptuelles. Nous examinons aussi comment le non-déterminisme peut être employé pour permettre l'exécution de programmes $\ll$ incomplets $\gg$ et comment la synthèse de plans au temps d'exécution pourrait être incorporée au modèle. Aussi à l'étude sont des questions telles que le traitement de l'incertitude et la révision des croyances, de même que les représentations relative à l'agent (indexicales) pour la robotique.

Notre formalisme est relativement limité quant au support offert aux applications multi-agents. Il serait certainement utile d'avoir un véritable modèle des engagements entre agents, avec un mécanisme d'implantation adéquat. Un traitement d'états mentaux tels que les intentions, les buts, les désirs, les intentions conjointes, et de notions comme la capacité pourrait aussi être utile (voir [6] pour un traitement de la capacité dans notre formalisme). De plus, est-il nécessaire de modéliser et implanter les attitudes itérées (ex. les connaissances de A sur les connaissances de B) et les connaissances mutuelles? Par rapport à la communication, jusqu'où devrait-on aller dans la modélisation des caractéristiques de la communication humaine (ex. tromperie, ironie) dans un ensemble d'outils pour la programmation d'agents? Quels types d'actes de communication sont nécessaires? En ce qui a trait à la coordination, quel type d'infrastructure devrait-on offrir? Est-il possible de réunir un ensemble de politiques de coordination génériques? En fin de compte bien sur, l'utilité de l'approche proposée devra être évaluée empiriquement.

References

[1] Philip R. Cohen et Hector J. Levesque. Communicative actions for artificial agents. Soumis à la First International Conference on Multiagent Systems (ICMAS'95), Stanford, CA, juin 1995.

[2] ARPA Knowledge Sharing Initiative External Interfaces Working Group. Specification of the KQML agent-communication language. Working Paper, juin 1993.

[3] Robert Goldblatt. *Logics of Time and Computation*. CSLI Lecture Notes No. 7. Center for the Study of Language and Information, Stanford University, Stanford, CA, 2ème. édition, 1987.

[4] Andrew R. Haas. The case for domain-specific frame axioms. Dans F.M. Brown, éditeur, *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, pages 343–348, Lawrence, KA, avril 1987. Morgan Kaufmann Publishing.

[5] Yves Lespérance, Hector J. Levesque, F. Lin, Daniel Marcu, Raymond Reiter et Richard B. Scherl. A logical approach to high-level robot programming – a progress report. Dans Benjamin

Kuipers, éditeur, *Control of the Physical World by Intelligent Agents, Papers from the 1994 AAAI Fall Symposium*, pages 109–119, New Orleans, LA, novembre 1994.

[6] Yves Lespérance, Hector J. Levesque, Fangzhen Lin et Richard B. Scherl. Ability and knowing how in the situation calculus. Soumis à IJCAI-95, 1995.

[7] Hector J. Levesque. Concurrency in the situation calculus. En préparation, 1995.

[8] Fangzhen Lin et Raymond Reiter. How to progress a database (and why) I. logical foundations. Dans Jon Doyle, Erik Sandewall et Pietro Torasso, éditeurs, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, pages 425–436, Bonn, Allemagne, 1994. Morgan Kaufmann Publishing.

[9] Fangzhen Lin et Raymond Reiter. How to progress a database II: The STRIPS connection. Rapport technique, Department of Computer Science, University of Toronto, 1995. A paraître.

[10] Fangzhen Lin et Raymond Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994.

[11] D. Marcu, Y. Lespérance, H. Levesque, F. Lin, R. Reiter et R. Scherl. Distributed software agents and communication in the situation calculus. Soumis au Workshop on Intelligent Computer Communication, Cluj-Napoca, Roumanie, juin 1995.

[12] John McCarthy et Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. Dans B. Meltzer et D. Michie, éditeurs, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, Edinburgh, UK, 1979.

[13] Robert C. Moore. A formal theory of knowledge and action. Dans J. R. Hobbs et Robert C. Moore, éditeurs, *Formal Theories of the Common Sense World*, pages 319–358. Ablex Publishing, Norwood, NJ, 1985.

[14] E. P. D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. Dans R.J. Brachman, H.J. Levesque et R. Reiter, éditeurs, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, ON, mai 1989. Morgan Kaufmann Publishing.

[15] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. Dans Vladimir Lifschitz, éditeur, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[16] D. Riecken (éditeur). Communications of the ACM **37** (7), numéro spécial sur les agents intelligents, juillet 1994.

[17] Richard B. Scherl et Hector J. Levesque. The frame problem and knowledge-producing actions. Dans *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 689–695, Washington, DC, juillet 1993. AAAI Press/The MIT Press.

[18] L.K. Schubert. Monotonic solution to the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. Dans H.E. Kyberg, R.P. Loui et G.N. Carlson, éditeurs, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer Academic Press, Boston, MA, 1990.

[19] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.