Les artifacts de calcul

Une solution aux délibérations longues

Cédric Dinont[*,**] — **Emmanuel Druon**[**]
Philippe Mathieu[*] — **Patrick Taillibert**[***]

[*] *SMAC/LIFL - UMR CNRS 8022 - 59655 Villeneuve-d'Ascq Cedex*
{cedric.dinont,philippe.mathieu}@lifl.fr

[**] *ISEN - 41 Bd Vauban - 59046 Lille Cedex*
{cedric.dinont,emmanuel.druon}@isen.fr

[***] *Thales Aerospace Division - 2 avenue Gay Lussac - 78990 Elancourt*
patrick.taillibert@fr.thalesgroup.com

RÉSUMÉ. *Dans une optique industrielle, nous voulons pouvoir intégrer dans les SMA du code hérité et assurer un fonctionnement correct même lorsque celui-ci ne dispose pas de bonnes propriétés temporelles. Nous focalisons notre étude sur le cas le plus défavorable où les agents doivent mettre en œuvre des algorithmes qui ont des durées d'exécution longues et difficiles à prévoir. Nous montrons pourquoi il est utile dans ce cadre de disposer d'entités de premier plan que les agents peuvent utiliser comme outils pour atteindre leurs buts. Initialement développée comme moyen d'encapsuler et de faciliter l'utilisation des mécanismes de coordination, la notion d'artifact issue des travaux de Omicini, Ricci et Viroli [OMI 04] peut être étendue pour fournir aux agents des outils de calcul qu'ils pourront apprendre à utiliser grâce à un langage de description de leur mode d'emploi. Ces idées sont illustrées sur une application développée à Thales Aerospace.*

ABSTRACT. *We want to be able to integrate legacy code in MAS and to ensure a correct operation even if it does not have good temporal properties. We focus our study on the most unfavourable case where agents use algorithms with long and hard to estimate execution times. We show why it is then useful to have first-class entities that can be used by agents as tools to achieve their goals. Initially developed as a way to encapsulate and to facilitate the use of coordination mechanisms, the concept of artifact due to Omicini, Ricci and Viroli [OMI 04] can be extended to provide computational tools to agents. Agents will be able to learn how to use them thanks to a description language of their instructions. These ideas are illustrated on an application developed in Thales Aerospace.*

MOTS-CLÉS : *Programmation de SMA, artifact, interaction, interface d'usage.*

KEYWORDS: *MAS design, artifact, interaction, usage interface.*

1. Introduction

Nous posons le problème de la programmation de SMA dans lesquels les actions des agents sont des délibérations mettant en jeu des algorithmes issus du domaine de l'Intelligence Artificielle. Ces algorithmes peuvent avoir des temps d'exécution longs, ainsi que de mauvaises propriétés temporelles.

Pour la bonne marche d'un SMA, les agents doivent de préférence avoir un comportement extraverti : il doivent prendre régulièrement connaissance des messages qu'ils ont reçus et des changements dans l'environnement. L'exécution par les agents d'algorithmes ayant des durées d'exécution longues va à l'encontre de ce principe. Les agents deviennent plutôt introvertis : ils se remettent en phase avec leur environnement une fois leurs délibérations terminées. En se comportant de cette manière, un agent risque par exemple de continuer un long calcul malgré le fait qu'un autre agent lui ait envoyé un message lui indiquant que le calcul est devenu inutile. La durée d'exécution d'un algorithme étant une notion relative, nous considérerons qu'un algorithme a une durée d'exécution longue lorsqu'elle dépasse la durée séparant deux événements qu'un agent doit prendre en compte : pour conserver un comportement extraverti à l'agent, le programmeur ne peut plus dans ce cas négliger le temps pris par l'exécution de l'algorithme considéré. Il devra disposer d'un moyen pour prendre en compte le nouvel événement, tout en laissant l'algorithme continuer son exécution.

De plus, lorsque les applications sont développées dans un cadre industriel, l'architecture utilisée doit permettre la réutilisation de code. La solution retenue devra donc permettre l'intégration de code hérité, celui-ci pouvant avoir de mauvaises propriétés temporelles. Par mauvaises propriétés temporelles, nous entendons que les algorithmes utilisés peuvent avoir des durées d'exécution hautement variables en fonction des données en entrée et que ces durées sont difficiles à prévoir.

Nous présentons en détail dans les paragraphes suivants quelles sont les contraintes apportées par les délibérations longues et le code hérité dans nos applications. Nous discutons dans la section 2 des différentes possibilités qui s'offrent à nous pour programmer de tels systèmes. Nous décrivons dans la section 3 la solution que nous proposons, basée sur la notion d'artifact issues des travaux d'Omicini, Ricci et Viroli. Nous l'illustrons sur une application dans la section 4.

1.1. *Délibérations longues*

Les problèmes auxquels s'attaque la recherche en Intelligence Artificielle sont la plupart du temps d'une complexité combinatoire très élevée. Les recherches dans ce domaine tentent de fournir des algorithmes les plus efficaces possibles pour résoudre en des temps raisonnables des problèmes de taille toujours plus importante. Cela passe souvent par l'élaboration d'heuristiques pour lesquelles on peut prouver que l'on obtient de bons résultats avec des données qui ont certaines propriétés. Ainsi, à l'exécution, il est possible, en inspectant les données d'entrée, de déterminer quelle heu-

ristique lancer. Il arrive parfois que l'on ne sache pas déterminer quelle heuristique choisir. Dans ce cas, il est possible d'en lancer une au hasard et si au bout d'un certain temps elle ne donne pas de réponse, on la stoppe et on essaye avec la suivante. On peut également lancer plusieurs heuristiques en parallèle et arrêter l'ensemble dès qu'une heuristique a trouvé une solution. La différence entre la complexité moyenne et la complexité au pire des cas étant souvent très importante, on ne peut pas souvent prédire le temps que va mettre un algorithme pour donner un résultat ou même s'il donnera un résultat.

Ce passage quasi-obligé par des heuristiques qui donnent des résultats plus ou moins bons en fonction des données en entrée est un frein à l'utilisation des résultats issus de la recherche en IA dans les applications réelles, où l'on veut pouvoir garantir que l'application ne gèlera pas en attente d'une réponse qui ne viendra jamais. Si l'on veut pouvoir utiliser une plus grande part des algorithmes issus des recherches en IA, il faut proposer des moyens d'améliorer les propriétés temporelles des systèmes ou tout du moins des moyens permettant d'assurer un fonctionnement temporel correct malgré des algorithmes ayant de mauvaises propriétés temporelles. Dans les deux cas, les agents doivent disposer de moyens pour raisonner sur les temps d'exécution et les ressources processeur utilisées par leurs tâches calculatoires.

La première voie de recherche est donc celle qui tente d'améliorer les algorithmes pour qu'ils aient les propriétés temporelles que l'on souhaite. C'est notamment ce qui est visé par les travaux sur l'algorithmie anytime [ZIL 99, GAR 96]. Un algorithme anytime est un algorithme incrémental pour lequel on dispose d'une fonction donnant la qualité de la réponse en fonction du temps. Disposer d'un algorithme anytime est le cas le plus favorable pour qu'un agent puisse raisonner sur le temps : l'agent pourra déterminer le temps pendant lequel il devra exécuter l'algorithme pour avoir une solution de qualité donnée. Il pourra également déterminer de quelle qualité il devra se contenter s'il dispose d'un temps contraint pour trouver une solution. Les recherches dans ce domaine ont donné quelques résultats intéressants, notamment dans le domaine médical [VIN 01]. Il faut cependant constater que trouver un algorithme anytime pour un problème donné n'est pas chose aisée. On ne peut parfois pas trouver de fonction de qualité en fonction du temps, mais disposer quand même d'un algorithme incrémental. On pourra dans ce cas faire moins de raisonnements temporels, mais l'utilisation de ressources processeur supplémentaires sera toujours bénéfique pour la solution finale.

Le cas le plus fréquent reste cependant celui où on ne dispose même pas de la propriété d'incrémentalité ; il faut donc se tourner vers la seconde voie de recherche qui consiste à prendre tel quels les algorithmes existants et à concevoir un niveau méta de monitoring/contrôle/décision. Celui-ci sera chargé de gérer l'exécution de plusieurs heuristiques en parallèle, de décider de la durée des contrats de temps alloués à chaque algorithme, de déterminer et modifier les paramètres de l'algorithme, de décider de continuer ou d'arrêter en fonction de la qualité de la solution ou de la rapidité de convergence. L'arrêt d'un calcul non terminé peut dans ce cas apparaître comme un gâchis de temps processeur. Pourtant, il y a des circonstances où l'agent devra prendre

une telle décision : par exemple si le calcul est devenu inutile ou si l'agent s'aperçoit que le temps de réponse est trop long, éventuellement à cause d'une boucle infinie apparue dans le calcul.

Nous considérons dans la suite qu'il n'y a pas d'interactions directes entre les différents algorithmes mis en œuvre par les agents : nous ne nous intéressons pas aux cas où les différents algorithmes auraient besoin de données calculées par les autres et donc auraient besoin d'interagir directement. Les cas qui nous intéressent sont les suivants :

– les agents réalisent des tâches de calcul indépendantes,

– les agents réalisent des tâches de calcul qui dépendent les unes des autres, mais les dépendances sont gérées dans le plan de l'agent,

– les agents utilisent différentes heuristiques pour résoudre un unique problème, mais les heuristiques fonctionnent indépendamment les unes des autres.

1.2. *Code hérité*

Le paradigme agent apparaît comme un moyen de réduire la complexité structurelle des systèmes. En concevant une application comme un ensemble d'entités indépendantes lors de l'exécution, on réduit le couplage entre les différents éléments du système et donc la programmation des entités du système se fait de manière plus indépendante également. Cependant, un paradigme tel que celui d'agent ne peut être facilement adopté qu'à la condition que l'on puisse réutiliser la base de code dont on dispose dans de nouvelles applications ou dans la refonte d'une application avec le paradigme agent.

Le fait de considérer l'utilisation de code hérité n'est pas qu'une problématique industrielle car nous avons vu dans le paragraphe précédent que la plupart du temps il n'est pas envisageable de modifier l'algorithme dont on dispose pour qu'il ait de meilleures propriétés temporelles. Il faut donc se résoudre à devoir intégrer dans nos systèmes des boîtes noires pour lesquelles on ne peut toucher au code, soit parce qu'on ne peut le faire (code hérité), soit parce que qu'on se refuse à le faire (algorithme complexe). On peut cependant dans tous les cas disposer des propriétés des algorithmes mis en œuvre (anytime, incrémental, interruptible, ...). Nous avons donc en réalité des boîtes grises à intégrer aux systèmes comme des outils de calcul pour les agents.

Le concept de réutilisabilité peut également être vu non pas du point de vue du programmeur, mais du point de vue agent : les agents devraient pouvoir dynamiquement apprendre à utiliser de nouveaux outils de calcul disponibles dans le SMA pour atteindre leurs buts. Pour cela, les outils de calcul devraient disposer d'un mode d'emploi qui indique aux agents comment les utiliser.

2. Quelle solution ?

Nous voulons que les agents conservent un comportement extraverti tout en ayant des tâches calculatoires longues. De plus, pour les raisons citées dans la section précédente, nous ne pouvons modifier le code qui réalise les tâches calculatoires. En

particulier, nous ne pouvons pas intégrer dans ce code un appel régulier à la méthode de traitement des nouveaux messages de la boîte aux lettres de l'agent. Il faut donc disposer de plusieurs fils d'exécution qui s'exécutent en parallèle (threads ou processus) :

– un qui intègre le comportement extraverti de l'agent (traitement des messages, vérification des changements dans l'environnement),

– un autre pour chaque tâche de calcul longue.

La question à laquelle il faut répondre maintenant est : sous quelle forme doit-on faire apparaître le second type de fil d'exécution dans le SMA ? Faut-il le faire apparaître comme une partie des agents : un objet, un composant, une couche ? Ou comme un type d'agent particulier ? Ou encore comme une nouvelle entité de premier plan dans les SMA ?

2.1. *Une partie de l'agent*

Considérons tout d'abord que les fils d'exécution qui réalisent les tâches longues soient des parties de l'agent.

On pourrait modéliser un agent comme un assemblage d'objets ou de composants. L'approche par composant semble séduisante car elle permet l'utilisation de code hérité et elle apporte la réutilisabilité dans différents contextes. De plus, les communautés composant et agent tentent de combiner leurs méthodes pour allier les avantages des deux approches : autonomie des agents et maturité des composants (par ex. : [LAC 06]). Cependant, les spécifications qui accompagnent un composant portent principalement sur les noms et paramètres des appels de méthode, sans donner un "mode d'emploi" qui indiquerait à l'agent quel enchaînement utiliser pour atteindre ses buts. L'agent devrait également pouvoir déduire de ce mode d'emploi les propriétés temporelles du composant qu'il manipule, ce qui n'est pas possible avec les spécifications très bas niveau utilisées avec les composants.

Le problème de la réactivité, au sens de la prise en compte *à temps* des changements dans l'environnement, tout en ayant des raisonnements à long terme, a déjà été traité. La littérature propose ainsi de construire un agent en plusieurs couches. Les deux principaux modèles proposés sont InteRRaP [MUL 93] et Touring Machines [FER 92]. Le modèle de Touring Machines utilise trois couches pour les comportements réactifs, de planification et de résolution de conflits. Le comportement de l'agent est cyclique. A chaque top d'horloge, chaque couche propose une action en fonction des données fournies par les capteurs de l'agent. Un module à base de règles est chargé d'inhiber les entrées ou les sorties de chacun des blocs pour qu'une seule action soit finalement sélectionnée. Dans InteRRaP, il y a également trois couches : la première pour les comportements de type réflexe connus, la seconde pour les comportements nécessitant de la planification et la troisième pour les comportements mettant en œuvre la coopération d'autres agents. Lorsqu'un agent perçoit un changement dans l'environnement, une réaction est d'abord cherchée dans la première couche. Si au-

cune réaction n'est prévue dans cette couche, la seconde couche tente d'en trouver une en planifiant un ensemble de réactions de la première couche. En dernier recours, la main est donnée à la dernière couche.

D'autres modèles ont été également proposés pour combiner des capacités cognitives et réactives comme [GUE 96] ou [BUS 94]. Tous ces modèles permettent de trouver un compromis entre les réactions de type réflexe et les réactions délibératives, mais ils ne sont pas adaptés lorsque les actions que l'agent réalise sont elles-mêmes des délibérations longues sur lesquelles les agents doivent raisonner, ce qui est notre cas.

Les modèles proposés (à composants ou à couches) ne conviennent pas à nos besoins. On peut donc se demander si nous devons proposer un nouveau modèle d'agent (à couche ou autre) qui réponde à nos attentes ? En réalité, les problèmes que nous voulons résoudre (délibérations longues et utilisation de code hérité) sont suffisamment généraux pour que l'on veuille proposer une solution indépendante du modèle d'agent choisi. Il faudrait donc plutôt utiliser d'autres objets de premier plan du SMA pour exécuter les tâches calculatoires.

2.2. *Un second agent*

S'il vaut mieux que les délibérations longues soient exécutées par une autre entité de premier plan, considérons qu'elles le sont pas un autre agent. C'est ce que nous avions proposé dans [DIN 06b]. Nous avions défini deux classes d'agents : une au comportement extraverti qui ne peut s'engager dans des traitements longs et une seconde au comportement introverti qui ne se remet en phase avec son environnement qu'après avoir terminé l'exécution de ses tâches. Les agents extravertis devaient déléguer leurs tâches longues aux agents introvertis. De plus, pour gérer des dates limites sur les tâches, l'exécution des agents introvertis était commandée par les agents extravertis. Un agent extraverti négociait des tranches de temps pour des tâches calculatoires réalisées par des agents introvertis auprès d'un Agent de Services Temporels (AST). L'utilisation de l'AST permettait de garantir que les tâches se terminaient avant leur date limite. Les agents extravertis avaient la charge de démarrer et de mettre en pause les agents introvertis qui travaillaient pour eux.

Cette solution, correcte, pose cependant plusieurs problèmes conceptuels dûs principalement au fait que l'on attribue généralement aux agents la propriété d'autonomie. Le concept d'autonomie étant très vaste et notre problématique étant de type génie logiciel, nous en utilisons une définition opérationnelle qui nous permet de manipuler le concept d'autonomie de manière concrète dans nos applications : un agent A est considéré comme étant autonome par rapport à un agent B si l'agent B ne peut prédire à coup sûr les réactions de l'agent A. Un agent ne dispose donc que d'un modèle approximatif d'un autre agent. S'il disposait d'un modèle précis, l'autre agent ne pourrait plus être considéré comme autonome par rapport à lui. Or, nous avons décrit précédemment nos tâches calculatoires comme des boîtes grises : un algorithme vu comme

une boîte noire non modifiable accompagné de ses caractéristiques. Un algorithme n'est pas vu ici comme une entité autonome et ses caractéristiques sont un modèle précis sur lequel un agent peut compter pour effectuer ses raisonnements.

Lorsque l'on programme un agent, il faut prendre en compte l'autonomie des autres agents. Dans notre cas, cette idée s'exprime par le fait que l'on devrait se refuser à stopper et redémarrer un autre agent, celui-ci perdant son autonomie par rapport à l'agent qui le commande. Or, nos agents introvertis sont commandés par les agents extravertis. D'un point de vue conceptuel, le fil d'exécution qui exécute nos tâches calculatoires se détache donc de l'idée que l'on se fait du concept d'agent.

2.3. *Une nouvelle entité ?*

L'introduction d'un nouveau type d'entités dans les SMA est délicate. Il faut identifier de manière précise un nouveau concept suffisamment différent de celui d'agent, ce qui est difficile tant le concept d'agent est vaste. Il est d'ailleurs très souvent possible de s'en sortir en identifiant plusieurs classes d'agents différentes, comme ce qui a été proposé dans le paragraphe précédent. Ferber avait notamment proposé dans [FER 95] une classification des agents en fonction de leur type de comportement qui est pertinente dans notre cas. Elle est rappelée dans la figure 1.

Relation au monde / Conduites	Agents cognitifs	Agents réactifs
Téléonomiques	Agents intentionnels	Agents pulsionnels
Réflexes	Agents "modules"	Agents tropiques

Figure 1. *Les différents types d'agents définis par Ferber*

Nous ne nous intéressons qu'à la colonne correspondant aux agents cognitifs. Les agents extravertis décrits dans le paragraphe précédent sont des agents intentionnels car ils ont un comportement téléonomique (leur comportement est dirigé par des buts *explicites*). Au contraire, les agents introvertis précédemment cités entrent plutôt dans la catégorie des agents "modules" : ils sont utilisés par les agents extravertis comme des sous-traitants dociles qui exécutent de manière réflexe les travaux qu'on leur donne. Ferber concède que ces agents se situent à la limite du concept d'agent. Nous pouvons nous poser la question de savoir ce que sont des agents "modules" qui ne seraient pas autonomes, comme nos boîtes grises. Faut-il toujours les considérer comme des agents ou y-a-t'il un bénéfice à les considérer comme un nouveau type d'entité ? Les remarques du paragraphe précédent sur le manque d'autonomie, ainsi que le fait que l'on doive disposer d'un modèle précis des entités qui réalisent les tâches calculatoires, sont des arguments suffisamment forts pour se permettre d'introduire un nouveau type d'entité de premier plan dans les SMA. Celle-ci ne sera pas autonome, pourra être utilisée par les agents comme outil de calcul pour atteindre leurs buts et disposera d'un mode d'emploi sur lequel l'agent pourra compter.

3. Les artifacts

Il est étonnant de remarquer que la plupart des travaux sur les agents cognitifs ont en fait porté sur les agents intentionnels et que peu de travaux ont été faits dans le sens de ce que Ferber nomme les agents “modules”. Une équipe de Bologne a cependant récemment été dans ce sens en travaillant sur les mécanismes de coordination dans les SMA et en proposant d’utiliser pour cela de nouvelles entités de premier plan comme outils de coordination, qu’ils nomment artifacts [VIR 04, OMI 04].

Nous présentons dans les paragraphes suivants le concept d’artifact, tel qu’il a été défini par l’équipe de Bologne, ainsi que les extensions que nous introduisons pour étendre le concept de base aux artifacts de calcul.

3.1. *Le concept d’artifact*

Si un agent est vu comme un système dirigé ou orienté par des buts, un artifact est une entité qu’un agent *utilise* pour atteindre ses buts. Un artifact est caractérisé par :

– *une interface d’usage :* définie comme un ensemble d’opérations. Les opérations sont de deux types : l’exécution d’une action et la perception de la terminaison d’une action,

– *sa fonction :* la description du service rendu par l’artifact,

– *une spécification de son comportement :* la description (formelle) du comportement interne de l’artifact,

– *un ensemble d’instructions opératoires :* la description (formelle) de la manière dont les agents peuvent utiliser l’artifact.

Les instructions opératoires sont exprimées à l’aide d’un formalisme à base d’algèbre de processus introduit dans [VIR 04]. Nous allons maintenant décrire succinctement ce formalisme. Le lecteur intéressé par les fondements théoriques pourra trouver plus d’informations sur les algèbres de processus dans [FOK 00]. La définition d’une instruction I est :

$$I ::= 0 \mid !\alpha \mid ?\pi \mid I; I \mid I + I \mid (I||I) \mid \mathcal{D}(t_1, \ldots, t_n)$$

Une instruction I peut être une instruction atomique : le comportement nul (0), l’exécution de l’action α ($!\alpha$), et la perception π de la terminaison d’une action ($?\pi$). Une instruction peut également être structurée grâce à différents opérateurs : “;” pour la composition séquentielle de deux instructions, “+” pour le choix et “||” pour la composition parallèle. Enfin, pour permettre la récursion, une instruction peut être l’invocation $\mathcal{D}(t_1, \ldots, t_n)$ d’une autre instruction opératoire.

Ce langage dispose d’un certain nombre de règles de congruence pour exprimer la commutativité et la transitivité des différents opérateurs, ainsi qu’une sémantique opérationnelle qui permet de décrire l’évolution du mode d’emploi en fonction des

actions décidées par l'agent et des perception qu'il reçoit de l'artifact. N'étant pas rappelées ici, le lecteur intéressé pourra trouver tous les détails dans [VIR 04].

3.2. *L'interface d'usage des artifacts de calcul*

Nous avons proposé dans [DIN 06a] une extension du concept d'artifact aux artifacts de calcul. Un artifact de calcul générique est un artifact auquel un agent peut donner un travail de calcul à effectuer sur des données, l'artifact renvoyant soit un résultat soit un rapport d'erreur en fonction des données qui lui ont été présentées. Nous définissons l'interface d'usage minimale d'un artifact de calcul ainsi : les actions possibles sont `start(input_data)`, `pause`, `restart`, `stop` et les perceptions possibles sont `finished(result)` et `failure(error_info)`. Ce modèle de base peut être étendu en fonction des capacités propres des artifacts de calcul réellement utilisés dans les applications. Par exemple, si nous voulons que les agents aient la possibilité de demander un état de la progression d'un calcul, une action `query_progress_report` et une perception `progress_report` seront ajoutées.

Notons qu'au niveau de l'implémentation, les artifacts de calcul peuvent prendre la forme de threads ou de processus. Le seul prérequis est de pouvoir réaliser les opérations `pause`, `restart` et `stop` à tout moment depuis l'extérieur de l'artifact visé. L'utilisation de threads sera faite avec prudence car les opérations précédentes ne sont généralement pas sûres : un deadlock peut apparaître si on utilise ces opérations en parallèle avec des verrous et de la mémoire partagée entre les différents threads. Ces problèmes disparaissent si, comme dit précédemment, on prend l'hypothèse de calculs indépendants : le partage de mémoire et la coordination entre les différents fils d'exécution n'est pas nécessaire.

Il n'est pas toujours possible de décrire le fonctionnement d'un artifact de calcul juste en ajoutant de nouvelles actions et perceptions. Par exemple, pour décrire le fonctionnement d'un algorithme à contrat, il est nécessaire d'introduire des extensions au langage de base. Pour cela, nous utiliserons les extensions suivantes dont la sémantique détaillée est donnée dans [VIR 04] et [DIN 06a] :

$$I ::= \ldots \mid \tau(T) \mid W(T)$$

$\tau(T)$ représente un *timeout* de T secondes (une durée maximale avant qu'un événement ne survienne), tandis que $W(T)$ représente un temps d'attente de T secondes nécessaire avant de pouvoir poursuivre. Un exemple d'utilisation de ces extensions est présenté dans la section suivante.

4. Application

Les artifacts ont été utilisés dans diverses applications développées chez Thales Aerospace. Cela concerne par exemple la planification de mission ou encore la coordination d'un ensemble de drones. Il apparaît même que tous les SMA que nous avons

développés à ce jour ont mis en évidence la nécessité de faire cohabiter des agents cognitifs autonomes et des agents qui, bien que s'exécutant en parallèle et ne dépendant des autres que par des échanges de messages, ne pouvaient en aucun cas être considérés comme autonomes, la spécification de leur comportement étant connue lors de la programmation des autres agents. A la différence des exemples traités par l'équipe de Bologne, nos artifacts concernaient moins la coordination que la mise en œuvre d'algorithmes complexes tels que ceux issus des recherches en Intelligence Artificielle (propagation de contraintes, recherche heuristique, ...) que nous ne voulions pas voir traités directement par les agents afin de ne pas les bloquer si la durée des traitements devenait prohibitive, ce qui se produit assez souvent avec de tels algorithmes.

Parmi ces applications, l'une d'entre elles mérite une mention particulière. Elle concerne la localisation de bateaux à partir d'avions d'observation par la seule utilisation de la direction relative du radar de veille du bateau observé. Cette méthode permet à l'avion de rester discret et donc d'éviter que le bateau prenne conscience du repérage auquel il est soumis. Elle peut être utilisée pour le repérage de pétroliers opérant un dégazage illicite et bien entendu également dans de nombreuses applications de nature militaire. Le principe est simple : grâce à un capteur spécifique (analogue à un goniomètre) l'avion mesure, à intervalles réguliers correspondant à la période du radar du bateau, la direction de ce dernier par rapport à l'avion. Les angles mesurés sont imprécis et peuvent même manquer en raison des conditions ambiantes difficiles prévalant dans ce type d'application. Si le bateau était immobile, une simple triangulation permettrait très rapidement de le localiser. Mais ce n'est bien sûr pas le cas. Néanmoins le calcul n'est possible que si la vitesse du bateau est notablement inférieure à celle de l'avion (un facteur 10 par exemple) et si la trajectoire de l'avion est correctement choisie, conditions en général faciles à satisfaire. Le calcul est effectué par un algorithme de propagation d'intervalles sur les domaines continus [LHO 93] qui exploite la différence de vitesse entre bateaux et avion pour fournir une localisation qui est progressivement réduite au fur et à mesure que les données arrivent. La propagation d'intervalle est particulièrement bien adaptée à ce genre de calcul car d'une part elle permet d'injecter de nouvelles contraintes de manière incrémentale (contraintes établies à partir des mesures reçues), et d'autre part de prendre en compte des contraintes non linéaires (en l'occurrence trigonométriques).

La gestion des temps de traitement est primordiale pour un agent utilisant ce type d'algorithme. En effet, la durée des calculs peut changer de manière considérable car des phénomènes de convergence lente peuvent apparaître lors du processus de propagation [LHO 94]. Pour conserver son autonomie, ne serait-ce que pour pouvoir décider d'abandonner la poursuite d'un bateau donné si celle-ci devient inutile, l'agent doit donc se protéger de blocages éventuels. La création d'un artifact de calcul est donc apparue comme une bonne solution car elle permettait de découpler la partie cognitive de l'agent de celle chargée d'effectuer les calculs, laissant à l'agent la possibilité d'une délibération pour décider d'arrêter un calcul devenu trop long ou de prendre en compte ou non une mesure nouvellement arrivée. L'artifact peut bien sûr être implémenté comme un agent ne possédant pas de capacité de décision propres ; c'est d'ailleurs ainsi que nous avions procédé initialement. Cependant, le type d'interac-

tion régissant les rapports entre l'artifact et les autres agents est différent des rapports entre agents. Ainsi, comme nous l'avons montré dans [DIN 06a], il doit être possible de suspendre puis relancer l'exécution de l'artifact afin de pouvoir exploiter au mieux les ressources offertes par la machine hôte, ce qui n'est pas souhaitable pour un agent réellement autonome devant pouvoir réagir aux changements de son environnement. Par ailleurs, le comportement des artifacts est connu à l'avance et la programmation de la délibération de l'agent peut donc en tenir compte, ce qui la rend notablement plus aisée.

Dans l'application de localisation de bateaux, nous utilisons un propagateur de contraintes sur intervalles que nous avons développé et mis en œuvre depuis de nombreuses années (Interlog [BOT 93]). Il est possible de lui spécifier un contrat de temps à chaque fois qu'une nouvelle contrainte est introduite. Si le point fixe n'a pas été atteint lorsque le contrat de temps expire, Interlog s'arrête et retourne un ensemble d'intervalles spécifiant la localisation obtenue à ce stade du traitement. L'agent utilisant l'artifact peut alors décider de relancer la propagation pour un nouveau contrat de temps afin d'améliorer la précision de la localisation ou, au contraire, d'introduire les contraintes correspondant à une mesure nouvellement arrivée. Pour prendre une telle décision, il estime l'amélioration apportée à la localisation par la réduction de sa surface obtenue au cours d'un contrat de temps.

Dans l'article [DIN 06a], nous avons montré comment le contrôle d'un tel artifact et l'utilisation d'un artifact de coordination permettait à la fois de garantir que l'agent était en mesure de décider de son comportement lors de l'arrivée d'une nouvelle mesure (contrat de temps terminé) et une meilleure utilisation de la puissance de calcul. Dans le présent article, nous nous plaçons dans le cas où il n'y a pas d'artifact de coordination et donc où l'agent gère lui même ses relations avec l'artifact de calcul. Pour décrire ce dernier, nous utiliserons le formalisme proposé dans [OMI 04] et rappelé succinctement dans la section précédente.

Il faut tout d'abord décrire l'interface d'usage qui spécifie les actions et perceptions en direction et provenant de l'artifact. Dans notre application de localisation de bateaux, les actions possibles sur l'artifact de calcul sont :

– `add_constraint(C)` : introduction d'une contrainte `C`,

– `propagate`, `propagate(T)` : lancement de la propagation sans ou avec un contrat de temps `T`,

– `pause` : suspension du traitement,

– `restart` : relance du traitement,

– `stop` : arrêt (normal) de l'artifact.

Par ailleurs, tout artifact peut être interrompu définitivement (destruction du thread ou du processus) mais cette action, au même titre que la création initiale, ne fait pas à proprement parler de l'interface d'usage. Les perceptions possibles sont :

– `syntax_error` : perception d'une erreur de syntaxe si la contrainte introduite par l'action `add_constraint` est incorrecte,

– `success` : perception du succès de l'introduction d'une nouvelle contrainte,

– `result(E, I)` : perception fournie à la fin du contrat de temps et représentant l'état du calcul (`E`) et l'ensemble des intervalles de localisation (`I`) comprenant les coordonnées cartésiennes de la zone et ses coordonnées polaires relatives à la dernière position de l'avion.

L'élément essentiel nécessaire à la programmation de l'agent utilisant un artifact est le mode d'emploi de l'artifact (les instructions opératoires). Deux situations ont été considérées. La première correspond à une utilisation de l'artifact de propagation de contraintes sans contrat de temps. En cas de convergence lente, l'agent n'est pas bloqué mais le calcul continue jusqu'à son terme, qui peut être très lointain. L'agent a néanmoins la possibilité de "tuer" définitivement l'artifact. Le mode d'emploi correspondant est le suivant :

```
OI1 := !stop +
       (!add_constraint(C) ;
        ((?syntax_error ; OI1) +
         (?success ; (OI1 + (!propagate ; ?result(E,I); OI1)))))
```

Lorsque l'artifact est au repos, l'agent peut le stopper ou ajouter une nouvelle contrainte. Après l'ajout d'une contrainte, il peut percevoir qu'il y avait une erreur de syntaxe dans sa contrainte et dans ce cas, il reboucle sur la même instruction opératoire pour arrêter l'artifact ou lui reproposer une nouvelle contrainte. Dans le cas où la nouvelle contrainte est correcte, l'agent peut ajouter de nouvelles contraintes ou lancer le calcul. Il est important de remarquer qu'en disposant de cette instruction opératoire, l'agent peut déterminer qu'une fois le calcul lancé par l'action `propagate`, il recevra obligatoirement un résultat, mais dans un temps indéterminé.

La seconde situation correspond au cas où la résolution s'accompagne d'un contrat de temps au bout duquel l'algorithme renvoie une réponse même si le point fixe n'a pas été atteint. A cette réponse est associée une information sur l'état du calcul afin que l'agent puisse décider de la suite à donner. Cet état peut prendre trois valeurs :

– `disponible` : le dernier calcul demandé est terminé,

– `en cours` : le calcul n'est pas terminé, mais un minimum de temps processeur a été utilisé pour garantir une qualité minimum de la solution,

– `saturé` : dans le cas contraire.

Le mode d'emploi devient alors le suivant :

```
OI2 := !stop +
       (WORK_OI +
        (!add_constraint(C) ;
         (?syntax_error + ? success)) ; OI2)

WORK_OI := !propagate(T) ; (τ(T) + ?result(E,I))
```

Contrairement au premier cas, l'agent peut relancer le calcul, même sans avoir ajouté de contrainte. Précédemment, cette possibilité n'était pas nécessaire car le résultat obtenu était toujours le point fixe et donc relancer le calcul ne servirait à rien. Notons qu'en analysant ce nouveau mode d'emploi, l'agent peut déterminer que s'il lance la résolution avec un contrat de `T` secondes, il aura le résultat dans au maximum `T` secondes.

La dernière caractéristique de l'artifact est sa fonction. Cela est spécialement utile dans un système ouvert où il est nécessaire pour un agent de découvrir les artifacts susceptibles de l'aider à résoudre son problème. Ce n'est pas le type d'utilisation dont nous avons eu besoin dans l'application de localisation de bateaux et nous n'avons donc pas eu à spécifier la fonction de nos artifacts.

L'approche agent/artifact que nous avons utilisée dans cette application a permis de réduire considérablement la difficulté de la programmation des agents et donc d'autoriser une délibération beaucoup plus poussée que lorsque le calcul était partie intégrante de l'agent. La prise en compte des contraintes temporelles s'en est trouvée également très simplifiée.

5. Conclusion et travaux futurs

Les artifacts apparaissent comme une bonne solution lorsque l'on veut introduire dans les SMA du code hérité dont le temps d'exécution est imprévisible. Les travaux de l'équipe de Bologne offrent une bonne base formelle, initialement utilisée pour décrire des artifacts de coordination, qui peut être adaptée aux artifacts de calcul.

Le langage de description de mode d'emploi d'artifacts à base d'algèbre de processus proposé par l'équipe de Bologne a été suffisant pour décrire le fonctionnement de nos artifacts de calcul dans notre application de localisation passive de bateaux. Il faudrait cependant l'étendre pour mieux prendre en compte les caractéristiques des artifacts de calcul. Par exemple, il serait intéressant que les perceptions ne marquent pas forcément la fin d'une action. Dans l'application présentée, cela ne posait pas de problème car l'algorithme utilisé était à contrat : la perception `result(E, I)` marque la fin du calcul ou du contrat. Par contre, dans le cas d'un algorithme qui donnerait la solution actuelle à intervalle régulier sans stopper ses calculs, il faudrait pouvoir utiliser des perceptions intermédiaires qui ne marquent pas la fin d'une action. Notons tout de même qu'un algorithme de ce type peut être décrit avec le langage actuel, mais de manière peu élégante, avec une action factice que l'agent exécute immédiatement après avoir perçu un état d'avancement de l'algorithme pour relancer le calcul. Egalement, pour introduire plus de flexibilité et pour permettre aux agents des raisonnements sur les modes d'emploi des artifacts plus étendus, il paraît intéressant que les paramètres des actions de délai et d'attente puissent être exprimées sous formes de contraintes.

Plus généralement, il serait intéressant d'expérimenter les idées développées dans cet article pour l'utilisation d'autres types d'outils que les artifacts de calcul mais qui ont les mêmes genres de caractéristiques (durées longues et parfois indéterminées) :

utilisation d'un véhicule pour se rendre d'un lieu à un autre, utilisation d'un automate, d'une machine-outil ou encore d'un appareil électro-ménager.

6. Bibliographie

[BOT 93] BOTELLA B., TAILLIBERT P., « Interlog : Constraint Logic Programming on Numeric Intervals », *3rd Int. Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics*, 1993.

[BUS 94] BUSSMAN S., DEMAZEAU Y., « An Agent Model Combining Reactive and Cognitive Capabilities », *Intelligent Robots and Systems '94. Advanced Robotic Systems and the Real World*, 1994.

[DIN 06a] DINONT C., DRUON E., MATHIEU P., TAILLIBERT P., « Artifacts for Time-Aware Agents », *AAMAS'06*, 2006.

[DIN 06b] DINONT C., DRUON E., MATHIEU P., TAILLIBERT P., « CPU Sharing for Autonomous Time-Aware Agents », *COGIS'06*, 2006.

[FER 92] FERGUSON I. A., « Touring Machines : Autonomous Agents with Attitudes », *Computer*, vol. 25, n° 5, 1992, p. 51–55, IEEE Computer Society Press.

[FER 95] FERBER J., *Les Systèmes Multi-Agents, Vers une intelligence collective*, InterEditions, 1995.

[FOK 00] FOKKINK W., *Introduction to Process Algebra*, Springer-Verlag, 2000.

[GAR 96] GARVEY A., LESSER V., « Design-to-time scheduling and anytime algorithms », *SIGART Bull.*, vol. 7, n° 2, 1996, p. 16–19, ACM Press.

[GUE 96] GUESSOUM Z., DOJAT M., « A Real-Time Agent Model in an Asynchronous-Object Environment », VAN HOE R., Ed., *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, 1996.

[LAC 06] LACOUTURE J., ANIORTÉ P., « Vers l'adaptation dynamique de services : des composants monitorés par des agents », *2eme journée multi-agent et composant*, 2006.

[LHO 93] LHOMME O., « Consistency Techniques for Numeric CSPs », *IJCAI'93*, 1993.

[LHO 94] LHOMME O., GOTLIEB A., RUEHER M., « Dynamic Optimization of Interval Narrowing Algorithms », *Journal of Logic Programming*, vol. 19-20, 1994.

[MUL 93] MULLER J., PISCHEL M., « The Agent Architecture InteRRaP : Concept and Application », 1993.

[OMI 04] OMICINI A., RICCI A., VIROLI M., CASTELFRANCHI C., TUMMOLINI L., « Coordination Artifacts : Environment-based Coordination for Intelligent Agents », *AAMAS'04*, 2004.

[VIN 01] VINCENT R., HORLING B., LESSER V., WAGNER T., « Implementing Soft Real-Time Agent Control », *Proceedings of the 5th ICAA*, , 2001, p. 355-362, ACM Press.

[VIR 04] VIROLI M., RICCI A., « Instruction-Based Semantics of Agent Mediated Interaction », *AAMAS'04*, 2004.

[ZIL 99] ZILBERSTEIN S., MOUADDIB A.-I., « Reactive Control of Dynamic Progressive Processing », *IJCAI '99 : Proceedings of the Sixteenth IJCAI*, Morgan Kaufmann Publishers Inc., 1999, p. 1268–1273.