Une Logique Temporelle pour les Objets Concurrents, et sa Procédure de Décision

Jean Paul Bahsoun, Stephan Merz, Corinne Servières

bahsoun,serviere@irit.fr, merz@informatik.tu-muenchen.de

IRIT, 118 route Narbonne 31062 Toulouse cedex France

phone: (33) 61 558211

Abstract

Dans cet article, nous commençons par décrire un concept d'objet concurrent ou agent. Puis nous proposons un cadre logique permettant de décrire formellement un agent et de montrer ses propriétés. Enfin, nous explicitons une procédure de décision dans ce cadre logique. Ces différents travaux sur la notion d'agent s'inscrivent dans un domaine d'intérêt plus large, celui de la réutilisabilité à différents niveaux qu'apporte une approche orientée objet.

1 Introduction

L'approche orientée objet est de plus en plus utilisée dans le domaine du développement des systèmes complexes, le génie logiciel, les langages de programmation, les bases de données et l'intelligence artificielle. Cette tendance s'explique par la facilité avec laquelle un objet se compose et se réutilise (voir par exemple [Mey88]). Depuis quelques années déjà, les notions de parallélisme et de concurrence ont été introduites dans l'approche orientée objet, pour aboutir au concept d'objet concurrent ou *agent* (par exemple [AH87, MW92]).

Bien que de nombreux programmes de recherches se soient récemment intéressés à la formalisation de différents concepts orientés objet ([AH87, Bak91, MW92]), peu ont étudié le problème de la vérification des propriétés d'un agent décrit formellement. Dans ce domaine, il faut mentionner cependant [FM92].

Notre travail actuel consiste à élaborer un concept d'objet concurrent ainsi qu'une formalisation de celui-ci ([BMS93, BMS94]). Le concept que nous proposons se distingue des propositions citées plus haut, notamment parce qu'il inclut plusieurs modes de communication. De plus, il permet la réutilisabilité à deux niveaux: par héritage de classes fonctionnelles ou de synchronisation, d'une part; et, par composition parallèle d'objets existants, d'autre part. La formalisation sur laquelle nous travaillons devra permettre non seulement de décrire formellement les agents, et de vérifier leurs propriétés, mais aussi de réutiliser au maximum les descriptions formelles et les propriétés montrées sur un agent constituant d'un agent plus complexe, pour spécifier ou montrer des propriétés de l'agent complexe.

Dans cet article, nous ne présenterons que la partie de ce travail qui concerne la formalisation d'un agent simple, construit sans réutilisation. Nous commencerons par décrire brièvement notre concept d'objet concurrent. Puis nous définirons une logique d'agent permettant de formaliser et de montrer les propriétés d'un agent. Nous exposerons ensuite une procédure de décision pour cette logique, sous la forme d'une méthode des tableaux. Enfin, nous conclurons en situant le contenu de cet article par rapport à nos travaux actuels.

2 Le Concept d'Agent

Dans cette section, nous explicitons ce que nous entendons par *agent* ou *objet concurrent*. Nous détaillerons plus particulièrement les aspects de ce concept formalisés à la section 3. Pour une description précise des autres aspects de notre concept, nous renvoyons le lecteur à [BMS93] ou [BMS94].

2.1 Un agent

Un agent possède une identité propre. Son *état* est décrit par un ensemble d'objets ou de variables encapsulés, donc invisibles à l'extérieur; son comportement est décrit par un ensemble d'*actions*.

A chaque action est associée une condition - que nous appelons garde ou *précondition* - portant sur l'état de l'agent. Pour qu'une action soit activable, c'est-à-dire pour que l'agent ait le droit de l'exécuter, il est nécessaire que sa précondition soit vraie.

Parmi les actions, nous distinguons les *méthodes* des *réflexes* qui sont des actions spontanées. Les méthodes quant à elles, constituent l'interface entre l'agent et son environnement. Donc, pour qu'une méthode soit activable, il est non seulement nécessaire que sa précondition soit vraie, mais il faut aussi qu'une requête demandant son exécution ait été reçue par l'agent.

Un agent choisit arbitrairement d'exécuter l'une de ses actions activables. Cependant, il doit respecter certains critères d'équité. Le choix d'une action effectué, l'agent exécute sans interruption les instructions dont est constituée l'action.

2.2 Un système d'agents

Un système réactif ou concurrent est constitué de plusieurs agents. Chacun d'eux possède son propre comportement, mais peut interagir avec les autres par l'envoi de messages demandant l'exécution d'une méthode.

Notre concept propose trois modes de communication distincts: asynchrone, synchrone et semi-synchrone. Après l'émission d'un message selon le mode asynchrone, l'agent émetteur poursuit son travail, sans attendre, par exemple, une éventuelle réponse de l'agent destinataire. Seul le mode asynchrone sera utilisé dans les exemples que nous exposerons.

2.3 Une classe

Comme dans l'approche orientée objet classique, une *classe* décrit une collection d'agents ayant les mêmes caractéristiques; un agent est donc obtenu par instanciation d'une classe d'agents.

La figure 1 donne la définition simplifiée de deux classes d'agents: la classe *serveur* et la classe *client*. Un agent serveur est censé gérer l'accès à une ressource imaginaire, tandis qu'un agent client tente d'accéder à cette ressource, par l'intermédiaire d'un serveur.

```
class serveur
  state
    actif: boolean := false
  method
    Accorder (c: agent) , Libérer
  behaviour
    for Accorder:
      when not actif
      do  actif := true
          send_asy(c,"Recevoir")
    for Libérer:
      do actif := false
end serveur
```

```
class client (s:serveur)
  state
    ressource, attente : boolean := false
  method
    Recevoir
  reflex
    Demander , Rendre
  behaviour
    for Demander:
      when (not ressource) and (not attente)
      do  attente := true
          send_asy(s,"Accorder",my_self)
    for Recevoir:
      do  (ressource,attente) := (true,false)
    for Rendre:
      when ressource
      do  ressource := false
          send_asy(s,"Libérer")
end client
```

Figure 1: Présentation de deux exemples de classes

Commentons brièvement ces deux classes. La liste des objets décrivant l'état d'un agent est donnée après le mot clé *state*. Nous remarquons que ces objets sont tous du type prédéfini booléen, et initialisés à faux. Les noms des méthodes, respectivement des réflexes, accompagnés des paramètres formels, sont donnés après le mot clé *method*, respectivement *reflex*. Le type *agent* utilisé dans la méthode *Accorder* regroupe l'ensemble de tous les agents, ce qui signifie que le paramètre effectif associé à c devra être l'identité d'un agent.

Le comportement de l'agent lorsqu'il exécute une action est décrit dans la rubrique *behaviour*. La précondition d'une action est une expression logique qui figure après le mot *when*. Lorsque l'action est activée, l'agent exécute la liste d'instructions située après le mot *do*. Dans l'exemple, nous avons utilisé seulement deux instructions: l'affectation multiple qui permet de modifier l'état d'un agent, et l'émission d'un message selon le mode asynchrone, grâce à la primitive *send_asy*. Les paramètres de cette primitive sont, respectivement, l'identité de l'agent destinataire, le nom de la méthode invoquée et la liste des paramètres effectifs de cette méthode.

L'objet *my_self*, paramètre de la méthode Accorder, dans le message émis par l'action Demander, contient l'identité de l'agent client émetteur.

Nous remarquons que la classe client est paramétrée par un agent de la classe serveur. Ceci implique qu'au moment d'instancier la classe client, nous devrons fournir l'identité d'un agent de la classe serveur, ce dernier jouera alors le rôle du serveur de l'agent client obtenu par instanciation.

Dans cet article, nous nous limiterons aux aspects de notre concept évoqués ci-dessus, notre but étant uniquement de présenter les bases de la formalisation du concept d'agent. Cependant, d'autres aspects tels que l'héritage, la réutilisation des contraintes de synchronisation, l'encapsulation d'agents ou la délégation sont présentés dans [BMS93] et [BMS94].

3 Formalisation du concept d'agent

Nous introduisons maintenant une logique permettant de formaliser les agents et les systèmes d'agents, et de démontrer leurs propriétés. Dans cette logique, nous distinguons trois niveaux de raisonnements.

Le niveau *action* traite les effets locaux des actions d'un agent. En effet, l'exécution d'une action a pour conséquence de modifier l'état de l'agent. Pour décrire ces changements d'états, un formalisme simple, basé sur les pré et postassertions des actions, est suffisant.

Au niveau *agent*, nous conduisons des raisonnements sur des agents individuels pour obtenir des propriétés de sureté ou de vivacité. Les règles de preuves dont on dispose à ce niveau s'appuient sur l'encapsulation de l'état de l'agent. Les aspects de communication et d'interaction avec l'environnement n'apparaissent à ce niveau que sous la forme d'hypothèses sur l'environnement de l'agent.

Enfin, le niveau *système* modélise le système global tel qu'il est vu par un observateur externe. A ce niveau, nous avons choisi de ne pas faire référence à l'état interne des agents qui constituent le système, seules l'exécution des actions et la transmission des messages sont visibles. Les propriétés auxquelles nous nous intéressons à ce niveau sont des propriétés de synchronisation ou de vivacité faisant intervenir plusieurs agents.

Pour obtenir ces trois niveaux de raisonnement, nous avons défini trois logiques différentes, reliées par des règles d'interface. Celles-ci permettent de dériver des propriétés d'un niveau supérieur, en utilisant des propriétés montrées au niveau immédiatement inférieur.

Dans la suite de cette section, nous présentons brièvement les niveaux action et agent qui sont les seuls nécessaires pour décrire un agent simple. De plus, nous nous restreindrons au fragment propositionnel de ces deux logiques. Ce fragment est suffisant pour traiter des agents dont les variables d'état sont de type booléen.

3.1 Le niveau action

Au niveau action, nous nous intéressons aux effets d'une action sur l'état local d'un agent, en ignorant tous les aspects liés à la communication. L'exécution d'une action est modélisée par une transition entre deux états. Formellement, une action α est une paire $\alpha = \textsf{pre}\ P_\alpha\ \textsf{act}\ \bar{\alpha}$, constituée:

- d'une précondition P_α, qui n'est autre qu'une formule de la logique propositionnelle classique dont les propositions sont des variables d'état de l'agent propriétaire de l'action α,
- d'une opération $\bar{\alpha}$. Dans nos exemples, la seule opération que nous utilisons est l'affectation multiple[1] de la forme $\vec{v_\alpha} := \vec{e_\alpha}$, où $\vec{v_\alpha}$ et $\vec{e_\alpha}$ sont deux listes finies de même longueur contenant respectivement, des variables d'état toutes distinctes et des expressions logiques formées à partir des variables d'états.

[1] La primitive send_asy également utilisée dans nos exemples concerne le niveau système.

La sémantique des opérations est donnée par des relations entre deux états, où $(s,t) \in [\![\gamma]\!]$ signifie que l'opération γ, exécutée dans l'état s, conduit à l'état t. Pour une action $\alpha = \mathsf{pre}\ P\ \mathsf{act}\ \bar{\alpha}$, nous définissons $[\![\alpha]\!] = \{(s,t) \in [\![\bar{\alpha}]\!] : s \models P\}$.

Le raisonnement concernant l'exécution des actions est basé sur les pré et postconditions. Formellement, nous utilisons la notation de Hoare $F\ \{\alpha\}\ G$ où F et G sont des formules de la logique propositionnelle classique, et α est une action. $F\ \{\alpha\}\ G$ est vraie si toute exécution de α, dans un état s qui satisfait F, conduit à un état t qui satisfait G.

Pour raisonner avec les triplets de Hoare, nous avons besoin de deux types de règles (ou axiomes): les règles dépendantes du langage, et les règles logiques qui permettent de déduire des assertions, à partir d'autres assertions.

Parmi ces dernières règles, nous trouvons les règles classiques de Hoare (comme la règle d'affaiblissement par exemple), ainsi que deux règles propres à notre définition d'une action gardée:

$$\neg P\ \{\mathsf{pre}\ P\ \mathsf{act}\ \bar{\alpha}\}\ \mathsf{false} \quad (\mathsf{act\text{-}pre}), \qquad \frac{F\ \{\bar{\alpha}\}\ G}{F\ \{\mathsf{pre}\ P\ \mathsf{act}\ \bar{\alpha}\}\ G} \quad (\mathsf{act\text{-}ope}).$$

Le seul axiome dépendant du langage, qui est nécessaire pour nos exemples, est l'axiome de l'affectation:

$$F[\vec{e}/\vec{v}]\ \{\vec{v} := \vec{e}\}\ F \quad (\mathsf{assign}).$$

3.2 Le niveau agent

La logique définie dans le paragraphe précédent n'est pas suffisante pour décrire les propriétés relatives au comportement global d'un agent. Pour cette raison, nous définissons une logique temporelle locale, sa structure correspond conceptuellement à l'exécution infinie d'un agent. Nous noterons la logique locale associée à l'agent a par $\mathcal{L}_a$. Une logique locale par agent rend notre approche formelle compositionnelle et facilite le raisonnement au niveau global d'un système. De plus, la propriété d'encapsulation est facilement prise en compte au niveau d'une logique locale.

Une telle logique doit nous permettre d'exprimer des propriétés concernant l'état de l'agent défini par ses variables et ses messages en attente. Chaque agent est considéré comme un système ouvert, il réagit aux messages en provenance de son environnement.

Dans la suite, nous décrivons d'abord la logique $\mathcal{L}_a$ de base qui ne prend pas en compte toutes les contraintes sémantiques d'un agent. Ce n'est que dans une deuxième étape que toutes ces contraintes seront prises en compte, par des restrictions sur la sémantique de la logique précédemment définie. Cette démarche nous permet d'obtenir une logique intermédiaire utilisable dans d'autres contextes, puisque indépendante de son utilisation finale.

Syntaxe de la logique $\mathcal{L}_a$

Soit $\mathsf{Act}(a) = \{\alpha_1, \ldots, \alpha_n\}$ l'ensemble de toutes les actions définies dans la classe de l'agent a, et $\mathsf{Meth}(a) \subseteq \mathsf{Act}(a)$ l'ensemble de tous les symboles de méthodes dans $\mathsf{Act}(a)$, et $\mathsf{Refl}(a) \subseteq \mathsf{Act}(a)$ l'ensemble de tous les symboles de réflexes. Soit StVar(a) l'ensemble des variables d'état de l'agent a. Soit $\mathcal{L}_P$ un langage propositionnel classique, construit à partir des propositions de l'ensemble StVar(a). Les formules de $\mathcal{L}_a$ sont définies inductivement de la manière suivante:

- Toute formule P de $\mathcal{L}_P$ est une formule de $\mathcal{L}_a$.
- Si $\alpha \in \mathsf{Act}(a)$ est un symbole d'action, alors $a.\alpha$ est une formule atomique de $\mathcal{L}_a$, qui intuitivement correspond à l'exécution de l'action α par l'agent a. La formule $a.\alpha$ peut être abrégée par α.
- Si $\alpha \in \mathsf{Meth}(a)$ est un symbole d'action dénotant une méthode, alors $a.\upharpoonleft\alpha$ est une formule atomique de $\mathcal{L}_a$ representant une requête en attente pour l'agent a, demandant l'exécution de la méthode α. La formule $a.\upharpoonleft\alpha$ peut être abrégée par $\upharpoonleft\alpha$.
- first est une formule atomique de $\mathcal{L}_a$.
- Si A et B sont des formules de $\mathcal{L}_a$, alors $\neg A$, $A \rightarrow B$, $\bigcirc_a A$, $\Box A$, A unless B sont des formules de $\mathcal{L}_a$.

D'autres opérateurs temporels et logiques sont définis comme abréviations des opérateurs de base. Ainsi, par exemple, $\Diamond$ abrège $\neg\Box\neg$. Nous définissons la formule idle par $\mathsf{idle} \equiv \bigwedge_{\alpha \in \mathsf{Act}(a)} \neg a.\alpha$.

Sémantique de la logique $\mathcal{L}_a$

Soit $\tau \notin \mathsf{Act}(a)$, représentant l'action vide de l'agent a. Une structure temporelle $\mathcal{K}_a$ pour $\mathcal{L}_a$ se compose d'une séquence infinie d'états $\sigma = (s_0, s_1, \ldots)$, où chaque état s_i affecte une valeur de vérité $s_i(v)$ à chaque variable d'état $v \in \mathsf{StVar}(a)$, et à chaque variable de la forme $a.\upharpoonleft\alpha$ où $\alpha \in \mathsf{Meth}(a)$, et d'une fonction de label partielle $\lambda : \mathbb{N} \rightarrow \mathsf{Act}(a) \cup \{\tau\}$, satisfaisant les conditions suivantes:

- $\lambda(0) = \tau$
- $\lambda(i)$ est définie pour une infinité de $i \geq 0$. Dans la suite, $\lambda(i)\downarrow$ et $\lambda(i)\uparrow$ dénoteront, respectivement, que $\lambda(i)$ est défini ou indéfini.
- Pour tout $i \geq 0$ tel que $\lambda(i) = \tau$, $s_i|_{\mathsf{StVar}(a)} = s_{i_{lab+}}|_{\mathsf{StVar}(a)}$, sachant que $i_{lab+} = \min\{l > i \mid \lambda(l)\downarrow\}$.
- Pour tout $i \geq 0$ tel que $\lambda(i+1)\uparrow$, $s_i = s_{i+1}$.

La relation de satisfaction pour les formules de $\mathcal{L}_a$ par rapport à une structure temporelle $\mathcal{K}_a$ est définie inductivement, de la manière suivante:

$\mathcal{K}_i \models p$	ssi	$s_i(p) = True$, où $p \in \mathsf{StVar}(a)$
$\mathcal{K}_i \models a.\alpha$	ssi	$\lambda(i)\downarrow$ et $\lambda(i) = \alpha$
$\mathcal{K}_i \models a.\upharpoonleft\alpha$	ssi	$s_i(\upharpoonleft\alpha) = True$
$\mathcal{K}_i \models \mathsf{first}$	ssi	$\lambda(j)\uparrow$ pour tout $0 < j \leq i$
$\mathcal{K}_i \models \neg A$	ssi	$\mathcal{K}_i \not\models A$
$\mathcal{K}_i \models A \rightarrow B$	ssi	$\mathcal{K}_i \not\models A$ ou $\mathcal{K}_i \models B$
$\mathcal{K}_i \models \bigcirc_a A$	ssi	$\mathcal{K}_j \models A$ où $j = \min\{l > i \mid \lambda(l)\downarrow\}$
$\mathcal{K}_i \models \Box A$	ssi	$\mathcal{K}_j \models A$ pour tout $j \geq i$
$\mathcal{K}_i \models A$ unless B	ssi	$\mathcal{K}_j \models A$ pour tout $j \geq i$ ou
		pour un $k \geq i : \mathcal{K}_k \models B$ et $\mathcal{K}_j \models A$ pour tout $i \leq j < k$

La sémantique de $\mathcal{L}_a$ diffère de celle des logiques temporelles linéaires classiques (par exemple [Krö87, MP92]), car elle prend en compte des états de bégaiement (voir [Lam91]), modélisés par $\lambda(i)\uparrow$. Intuitivement $\lambda(i)\uparrow$ correspond à une transition effectuée par l'environnement de l'agent, et qui passe complètement inaperçue pour l'agent a. Par contre, $\lambda(i)\downarrow$ correspond à une transition effectuée par l'agent a, dans l'état s_i. Autoriser le bégaiement fini permet de réaliser des preuves compositionnelles, modulaires pour les propriétés d'un système d'agents, à partir de propriétés obtenues pour des agents individuels. Le bégaiement infini est évité en exigeant que $\lambda(i)$ soit définie pour un nombre infini de $i \geq 0$. L'action τ permet éventuellement à l'agent de modifier l'ensemble des messages en attente, ce qui évite que l'agen! t ne soit définitivement bloqu

Une formule A est dite *valide dans* K, ce qui est noté par $\mathcal{K} \models A$, si $\mathcal{K}_i \models A$ est vraie pour tout $i \geq 0$. Elle est dite *valide* ($\models A$), si $\mathcal{K} \models A$ est vraie pour toute structure temporelle $\mathcal{K}$. Elle est dite *satisfiable*, si il existe une structure temporelle $\mathcal{K}$, un entier $i \geq 0$ tels que $\mathcal{K}_i \models A$.

Axiomatique de la logique $\mathcal{L}_a$

Pour donner une axiomatisation adéquate et complète de $\mathcal{L}_a$ (voir les démonstrations dans [Mer93]), nous avons besoin d'introduire trois classes de formules distinctes.

Les *formules restreintes pour les actions* ont la particularité de ne pas avoir de sous-formules de la forme $a.\alpha$, où $\alpha \in \mathsf{Act}(a)$, dans le champ d'un opérateur $\circ_a$.

Toute formule de la forme $\Box A$, $\neg\Box A$, A unless B or $\neg(A \mathbin{\mathsf{unless}} B)$ est une *formule globalement gardée*; de plus, si A et B sont deux formules globalement gardées, alors $A \wedge B$ est une formule globalement gardée. Pour toute formule globalement gardée F, nous définissons une formule $\iota(F)$ de la manière suivante: $\iota(\Box A) \equiv \mathsf{true}$, $\iota(\neg\Box A) \equiv A$, $\iota(A \mathbin{\mathsf{unless}} B) \equiv \neg B$, $\iota(\neg(A \mathbin{\mathsf{unless}} B)) \equiv A$, et $\iota(A \wedge B) \equiv \iota(A) \wedge \iota(B)$.

La classe des *formules preservées par le bégaiement* est définie inductivement de la manière suivante, en même temps qu'une formule $\kappa(F)$ pour toute formule F préservée par le bégaiement:

- Toute formule A de $\mathcal{L}_P$ est préservée par le bégaiement, et $\kappa(A) \equiv A$.
- Toute formule $\circ_a A$ est préservée par le bégaiement, et $\kappa(\circ_a A) \equiv \circ_a A$.
- Si A est une formule restreinte pour les actions, alors A est préservée par le bégaiement, et $\kappa(A) \equiv A$.
- Si A est globalement gardée, alors $A \wedge \iota(A)$ est préservée par le bégaiement, et $\kappa(A \wedge \iota(A)) \equiv A$.
- Si A et B sont préservée par le bégaiement, alors $A \wedge B$ et $A \vee B$ sont préservées par le bégaiement, et $\kappa(A \wedge B) \equiv \kappa(A) \wedge \kappa(B)$ et $\kappa(A \vee B) \equiv \kappa(A) \vee \kappa(B)$.

Dans [Mer93], nous avons montré le résultat suivant à propos des formules préservées par le bégaiement: si $\mathcal{K}_i \models F$ et F est une formule préservée par le bégaiement, si de plus $\lambda(i+1)\uparrow$, alors $\mathcal{K}_{i+1} \models \kappa(F)$.

Donnons maintenant une axiomatisation adéquate et complète de $\mathcal{L}_a$.

Axiomes:

(taut)	toute instance de tautologie propositionnelle
(ax1)	$\neg \bigcirc_a A \leftrightarrow \bigcirc_a \neg A$
(ax2)	$\bigcirc_a (A \rightarrow B) \rightarrow (\bigcirc_a A \rightarrow \bigcirc_a B)$
(ax3)	$\Box A \rightarrow A \wedge \bigcirc_a \Box A$
(ax4)	$B \rightarrow A \mathsf{\ unless\ } B$
(ax5)	$\Box A \rightarrow A \mathsf{\ unless\ } B$
(ax6)	$A \mathsf{\ unless\ } B \rightarrow A \vee B$
(ax7)	$\mathsf{first} \rightarrow (P \rightarrow \bigcirc_a P)$ si P est une formule de $\mathcal{L}_P$
(ax8)	$\bigcirc_a \mathsf{idle} \rightarrow (\bigcirc_a P \rightarrow \bigcirc_a \bigcirc_a P)$ si P est une formule de $\mathcal{L}_P$
(ax9)	$\mathsf{idle} \rightarrow (A \wedge \neg \Box A \rightarrow \bigcirc_a \neg \Box A)$
(ax10)	$\mathsf{idle} \rightarrow (\neg B \wedge A \mathsf{\ unless\ } B \rightarrow \bigcirc_a (A \mathsf{\ unless\ } B))$
(ax11)	$\mathsf{idle} \rightarrow (A \wedge \neg (A \mathsf{\ unless\ } B) \rightarrow \bigcirc_a \neg (A \mathsf{\ unless\ } B))$
(ax12)	$\mathsf{first} \rightarrow \mathsf{idle}$
(ax13)	$\bigcirc_a \neg \mathsf{first}$
(ax14)	$\alpha \rightarrow \neg \beta$ si $\alpha \not\equiv \beta$ pour $\alpha, \beta \in \mathsf{Act}(a)$

Règles d'inférence:

(mp)	$A, A \rightarrow B \vdash B$
(nex)	$A \vdash \bigcirc_a A$
(first)	$\mathsf{first} \rightarrow \Box A \vdash A$
(idle)	$A \rightarrow B \wedge \iota(B), A \rightarrow C, \mathsf{idle} \wedge B \wedge C \rightarrow \bigcirc_a D \vdash A \rightarrow \bigcirc_a B \vee \bigcirc_a D$ si B est globalement gardée et C est restreinte pour les actions
(ind)	$A \rightarrow B, A \rightarrow C, A \rightarrow \bigcirc_a A, \mathsf{idle} \wedge \kappa(B) \rightarrow C \vdash A \rightarrow \Box C$ si B est préservée par le bégaiement

Sémantique et axiomatique de programme

Dans cette sous-section, nous décrivons les contraintes sémantiques supplémentaires qui caractérisent une exécution d'un agent. Une struture temporelle $\mathcal{K}_a$ est une *exécution* d'un agent a si elle vérifie les contraintes suivantes:

1. Pour tout $i \geq 0$ tel que $\lambda(i) = \alpha \in \mathsf{Act}(a)$, on a $(s_i|_{\mathsf{StVar}(a)}, s_j|_{\mathsf{StVar}(a)}) \in [\![\bar{\alpha}]\!]$ où $j = i_{lab+}$. Ce qui signifie que si $\lambda(i) = \alpha$, la transition décrite par $(s_i|_{\mathsf{StVar}(a)}, s_j|_{\mathsf{StVar}(a)})$ correspond à une exécution de l'action α.

2. Pour tout symbole d'action $\alpha \in \mathsf{Meth}(a)$ dénotant une méthode de a et pour tout $i > 0$, si $s_i(\uparrow\alpha) = \mathit{true}$ et $\lambda(i)\downarrow$ et $\lambda(i) \neq \alpha$, alors $s_{i_{lab+}}(\uparrow\alpha) = \mathit{true}$. C'est-à-dire toute requête en attente demeure en attente jusqu'à ce que la méthode correspondante soit exécutée.

3. Pour tout symbole d'action $\alpha \in \mathsf{Meth}(a)$ et tout $i \geq 0$, si $\lambda(i) = \alpha$ alors $s_i(\uparrow\alpha) = \mathit{true}$. Ce qui signifie qu'une méthode ne peut être exécutée que si une requête est en attente pour cette méthode.

Aux contraintes 2 et 3 correspondent les deux axiomes suivants:

(request) $\bigcirc_a(a.\uparrow\alpha \wedge \neg a.\alpha) \rightarrow \bigcirc_a\bigcirc_a a.\uparrow\alpha$

(method) $a.\alpha \rightarrow a.\uparrow\alpha$

La règle (action) donnée ci-après prend en compte la contrainte 1. Quant à la règle (agent), elle capture l'hypothèse d'encapsulation d'un agent a. Ces deux règles servent aussi d'interface entre le niveau action et le niveau agent.

$$\frac{F\ \{\alpha_1,\ldots,\alpha_n\}\ G}{F \wedge (\alpha_1 \vee \ldots \vee \alpha_n) \rightarrow \bigcirc_a G} \quad \text{(action)}$$

$$\frac{F\ \{\alpha\}\ G \quad \text{pour toute action } \alpha \in \mathsf{Act}(a)}{F \rightarrow \bigcirc_a(F \vee G)} \quad \text{(agent)}$$

Etat initial et équité

La donnée des ensembles Act(a) et StVar(a) n'est pas suffisante pour décrire complètement un agent a. En effet, le comportement de celui-ci dépend aussi de la valeur de ses variables d'état, dans l'état initial. Cette valeur est précisée dans la description formelle d'un agent, sous la forme d'un axiome d'initialisation du style: $\mathsf{first} \rightarrow P$, où P est une formule de $\mathcal{L}_P$, conjonction de tous les littéraux issus de StVar(a), de façon à décrire complètement l'état initial.

Pour que notre description d'un agent soit complète, il manque encore des conditions d'équité sur les actions. Nous avons opté pour une approche flexible en exigeant des axiomes explicites, dans la description d'une classe, plutôt que de supposer, par défaut, un certain type d'équité pour toutes les actions d'une classe. Les conditions d'équité les plus courantes sont l'équité faible et l'équité forte. Dans notre langage où les actions sont en fait des affectations gardées de la forme $\alpha = \mathsf{pre}\ P_\alpha\ \mathsf{act}\ \bar{\alpha}$, les axiomes d'équité faible et forte s'exprimeront respectivement de la manière suivante:

$$\begin{aligned} \mathit{Efaible}(\alpha) &= \Diamond\Box P'_\alpha \rightarrow \Box\Diamond\alpha \quad \text{et} \\ \mathit{Eforte}(\alpha) &= \Box\Diamond P'_\alpha \rightarrow \Box\Diamond\alpha, \end{aligned}$$

où $P'_\alpha \equiv P_\alpha \wedge \uparrow\alpha$ si $\alpha \in \mathsf{Meth}(a)$, et $P'_\alpha \equiv P_\alpha$ sinon.

Exemple

Formalisons maintenant la classe *serveur* de l'exemple 1. Nous remarquons que l'action *Accorder* est paramétrée par l'identité de l'agent demandeur de la ressource. Or le fragment propositionnel de la logique $\mathcal{L}_a$ ne permet pas de considérer les actions paramétrées. Cependant, nous pouvons contourner cet obstacle, en faisant l'hypothèse réaliste que le nombre de clients d'un agent serveur est fini, et par conséquent en supposant l'existence d'une action *Accorder* pour chacun de ces clients.

Les actions *Accorder(c)* et *Libérer* sont décrites par:

$$\text{Accorder(c)} = \mathsf{pre}\ \neg actif\ \mathsf{act}\ actif := \mathsf{true}, \quad \text{Libérer} = \mathsf{pre\ true\ act}\ actif := \mathsf{false}.$$

De plus, nous avons $\mathsf{StVar}(s) = \{actif\}$, et Act(s) = {Accorder(c),Libérer}.

Pour que l'agent s appartienne à la classe serveur nous ajoutons l'axiome d'initialisation:

$$\mathsf{first} \rightarrow \neg\mathsf{actif}.$$

En ce qui concerne l'équité, nous supposerons une équité forte pour toutes les actions Accorder(c), et une équité faible pour l'action Libérer, ce que précisent les deux axiomes suivants:

$$\Box\Diamond(\neg\mathsf{actif} \wedge \uparrow\mathsf{Accorder}(c)) \rightarrow \Box\Diamond\mathsf{Accorder}(c),$$
$$\Diamond\Box\uparrow\mathsf{Libérer} \rightarrow \Box\Diamond\mathsf{Libérer}.$$

Dans [Mer93], nous avons montré notamment les propriétés de sureté et de vivacité suivantes, pour l'agent s:

$$\vdash \mathsf{Accorder}(c) \rightarrow \bigcirc_s(\neg\mathsf{Accorder}(c') \text{ } \mathsf{unless} \text{ } \mathsf{Libérer})$$
$$\mathsf{Accorder}(c') \rightarrow \Diamond\uparrow\mathsf{Libérer} \vdash \ \uparrow\mathsf{Accorder}(c) \rightarrow \Diamond\mathsf{Accorder}(c)$$

4 La méthode des tableaux pour la logique d'agent

La méthode des tableaux élaborée pour $\mathcal{L}_a$ est une extension de la méthode des tableaux pour la logique temporelle propositionnelle classique présentée dans [Wo85]. Les connecteurs logiques sont traités comme dans la logique propositionnelle classique (voir par exemple [Sm68]) et les connecteurs temporels sont décomposés en une exigence sur l'état courant et en une exigence sur le reste de la séquence. Grâce à cette décomposition, la construction du tableau, qui est en fait un graphe orienté, s'effectue en recherchant, état par état, un modèle pour la formule traitée.

Nous commençons par présenter une méthode pour la logique d'agent de base, c'est-à-dire sans les trois contraintes sémantiques d'une exécution. Nous verrons ensuite la méthode étendue pour la logique avec ces trois contraintes. Faute de place, nous décrirons seulement l'essentiel de la méthode des tableaux, en particulier nous ne nous donnerons pas les démonstrations. Cependant celles-ci figurent dans le rapport [Ser94].

4.1 La méthode des tableaux pour la logique d'agent de base

L'opérateur next

Pour les besoins de la méthode des tableaux, nous introduisons un nouvel opérateur temporel unaire que nous notons **next**. Si F est une formule de $\mathcal{L}_a$, alors $\mathsf{next}(F)$ sera une formule utilisable pour développer la méthode des tableaux, bien qu'elle ne fasse pas partie de la logique $\mathcal{L}_a$. Intuitivement, l'opérateur **next** nous servira à décomposer les opérateurs temporels $\Box$ et **unless** de la logique $\mathcal{L}_a$ en une exigence sur l'état courant et une exigence sur le reste de la séquence.

Si $\mathcal{K}_a$ est une structure temporelle de $\mathcal{L}_a$, et F est une formule de $\mathcal{L}_a$, la relation de satisfaction pour la formule $\mathsf{next}(F)$ est définie de la manière suivante:

$$K_i \models \mathsf{next}(F) \text{ ssi } \left\{ \begin{array}{lll} \mathcal{K}_{i_{lab+}} & \models & F \quad \text{si } \lambda(i) = \tau \text{ ou } \lambda(i)\uparrow \text{ ou },\\ & & \quad F \text{ est une formule restreinte pour les actions} \\ \mathcal{K}_{i+1} & \models & F \quad \text{sinon} \end{array} \right\}$$

Le lemme suivant établit de quelle manière sont décomposées les formules de la forme $\Box A$, $\neg\Box A$, $A \mathsf{unless} B$, et $\neg(A \mathsf{unless} B)$ grâce à l'opérateur **next**.

Lemme 1 *Soit $\mathcal{K}_a$ une structure temporelle de $\mathcal{L}_a$, et $i \geq 0$. Alors,*

(i) $\mathcal{K}_i \models \Box A$ *ssi* $\mathcal{K}_i \models A \wedge \mathsf{next}(\Box A)$.

(ii) $\mathcal{K}_i \models \neg\Box A$ *ssi* $\mathcal{K}_i \models \neg A \vee \mathsf{next}(\neg\Box A)$.

(iii) $\mathcal{K}_i \models A \mathsf{unless} B$ *ssi* $\mathcal{K}_i \models B \vee (A \wedge \mathsf{next}(A \mathsf{unless} B))$.

(iv) $\mathcal{K}_i \models \neg(A \mathsf{unless} B)$ *ssi* $\mathcal{K}_i \models \neg B \wedge (\neg A \vee \mathsf{next}(\neg(A \mathsf{unless} B)))$.

Formules élémentaires

Nous distinguons deux types de formules: élémentaires et non-élémentaires. Les formules élémentaires sont celles qui sont de la forme $\bigcirc_a F$, ou $\mathsf{next}(F)$ (où F est une formule de $\mathcal{L}_a$), ou qui sont soit une formule atomique soit la négation d'une formule atomique.

Noeuds du graphe

Pour tester la satisfiabilité d'une formule f, la méthode des tableaux construit un graphe orienté. Chaque noeud n du graphe est labellé par un ensemble de formules T_n, et est étiqueté par une étiquette E_n. Quatre étiquettes sont possibles: "action", "τ", "défini" ou "indéfini". Ces quatre étiquettes correspondent à quatre possibilités pour la fonction de label λ en un point i. Celle-ci peut être indéfinie ou définie, et dans ce cas elle peut associer à i, une action $\alpha \in \mathsf{Act}(a)$ ou l'action vide τ.

Chaque fois qu'une formule f du graphe sera traitée, nous marquerons cette formule avec le symbole $*$, pour obtenir f^*, ceci afin d'éviter de traiter deux fois la même formule.

Un noeud qui ne contient que des formules élémentaires ou marquées sera appelé:

- un *état*, s'il est étiqueté par "action", "τ", ou "indéfini",
- un *double-état*, s'il est étiqueté par "défini".

Le noeud initial et les fils immédiats d'un état sont des *pré-états*.

Un noeud qui n'est pas étiqueté avec "défini", et dont aucun ancêtre n'est étiqueté avec "défini" est appelé un *noeud inactif initial*. Par conséquent, pour qu'un noeud ne soit pas inactif initial, il suffit qu'il soit étiqueté par "défini" ou à défaut que l'un de ses ancêtres le soit.

Construction du graphe

Donnons tout d'abord, les règles de décomposition qui nous permettrons d'obtenir des formules élémentaires à partir d'une formule non-élémentaire f. Ces règles de décomposition associent, à chaque formule f non-élémentaire, un ensemble Σ d'ensembles S_i de formules f_j. Elles sont exprimées de la manière suivante.

$$
\begin{array}{llll}
\textbf{D1} & \neg\neg f & \rightarrow & \{\{f\}\} \\
\textbf{D2} & \neg\bigcirc_a f & \rightarrow & \{\{\bigcirc_a \neg f\}\} \\
\textbf{D3} & f_1 \rightarrow f_2 & \rightarrow & \{\{\neg f_1\}, \{f_2\}\} \\
\textbf{D4} & \neg(f_1 \rightarrow f_2) & \rightarrow & \{\{f_1, \neg f_2\}\} \\
\textbf{D5} & \Box f & \rightarrow & \{\{f, next(\Box f)\}\} \\
\textbf{D6} & \neg\Box f & \rightarrow & \{\{\neg f\}, \{next(\neg\Box f)\}\} \\
\textbf{D7} & f_1 \,\mathsf{unless}\, f_2 & \rightarrow & \{\{f_2\}, \{f_1, next(f_1 \,\mathsf{unless}\, f_2)\}\} \\
\textbf{D8} & \neg(f_1 \,\mathsf{unless}\, f_2) & \rightarrow & \{\{\neg f_1, \neg f_2\}, \{\neg f_2, next(\neg(f_1 \,\mathsf{unless}\, f_2))\}\}
\end{array}
$$

Nous pourrions établir d'autres règles de décomposition pour les opérateurs logiques ou temporels définis à partir des opérateurs de base, comme par exemple $\Diamond f \rightarrow \{\{f\}, \{next(\Diamond f)\}\}$.

Nous remarquons que pour chaque règle de la forme $f \rightarrow \{S_i\}$, f est satisfiable si et seulement si au moins l'un des ensembles de formules S_i est satisfiable.

Avant d'expliciter la procédure de construction du graphe de la méthode des tableaux, donnons quelques notations préliminaires.

Soit $\mathcal{F}$ un ensemble de formules. Nous définissons les sous-ensembles de formules suivants:

$$
\begin{array}{lcl}
\mathcal{F}^{next_a} & = & \{A \mid \bigcirc_a A \in \mathcal{F}\} \\
\mathcal{F}^{next} & = & \{A \mid \mathsf{next}(A) \in \mathcal{F}\} \\
\mathcal{F}^{prop} & = & \{p \mid p \in \mathcal{F}, \text{ et p est une proposition de } \mathcal{L}_p\} \quad \cup \\
 & & \{\neg p \mid \neg p \in \mathcal{F}, \text{ et p est une proposition de } \mathcal{L}_p\} \\
\mathcal{F}^{\uparrow act} & = & \{a.\!\uparrow\!\alpha \mid a.\!\uparrow\!\alpha \in \mathcal{F}, \text{ et } \alpha \in Meth(a)\} \quad \cup \\
 & & \{\neg a.\!\uparrow\!\alpha \mid \neg a.\!\uparrow\!\alpha \in \mathcal{F}, \text{ et } \alpha \in Meth(a)\}
\end{array}
$$

Si n est un état du graphe, donc s'il est labellé avec l'ensemble T_n et s'il est étiqueté avec E_n, nous posons,

$$
\begin{array}{lcl}
Def(n) & = & T_n^{next} \cup T_n^{next_a} \cup \gamma_{prop}(n) \\
Indef(n) & = & \bigcirc_a T_n^{next_a} \cup T_n^{prop} \cup T_n^{\uparrow act} \cup \gamma_{next}(n)
\end{array}
$$

avec

- $\gamma_{prop}(n)$

$$
\begin{array}{lll}
= & T_n^{prop} & \text{si } E_n = \text{“}\tau\text{''} \\
= & \emptyset & \text{sinon}
\end{array}
$$

- $\gamma_{next}(n)$

$$
\begin{array}{ll}
= & \bigcirc_a T_n^{prop} \cup \mathsf{next}(T_n^{next}) \quad \text{si } E_n = \text{“}\tau\text{''} \\
= & \{F \mid \mathsf{next}(F) \in T_n \text{ et } F \text{ n'est pas une formule restreinte pour les actions}\} \cup \\
 & \{\mathsf{next}(F) \mid \mathsf{next}(F) \in T_n \text{ et } F \text{ est une formule restreinte pour les actions}\} \text{ sinon}
\end{array}
$$

Expliquons intuitivement la signification de $Def(n)$ et $Indef(n)$.

Si n est un état du graphe, $Def(n)$ est l'ensemble des formules qui doivent être satisfaites dans un état s_{i+1} d'une structure temporelle, et tel que $\lambda(i+1)\!\downarrow$, pour que l'ensemble des formules T_n soit satisfait dans l'état s_i.

De même, si T_n est un ensemble de formules à satisfaire dans un état s_i d'une structure temporelle, et tel que $\lambda(i)\downarrow$, et si s_{i+1} est tel que $\lambda(i+1)\uparrow$, alors $Indef(n)$ est l'ensemble des formules qui doivent être satisfaites dans l'état s_{i+1} pour que l'ensemble T_n soit satisfait en s_i.

Intéressons-nous maintenant à la construction des noeuds du graphe.

Si f est la formule dont on veut tester la satisfiabilité, le noeud initial du graphe est labellé avec la formule $\Diamond f$, et porte l'étiquette "τ". Dans la suite, f sera appelée *formule initiale*.

Dans l'algorithme de construction du graphe, quand nous dirons "créer un fils pour le noeud n", cela sous-entendra que ce fils n'est effectivement créé que lorsqu'un noeud identique au fils à créer n'existe pas déjà dans le graphe. Si un tel noeud existe, il suffira d'ajouter une flèche menant de n à ce noeud. Deux noeuds seront considérés comme identiques, s'ils ont le même label, la même étiquette et s'ils sont tous les deux inactifs initiaux ou si aucun des deux n'est inactif initial.

La construction du graphe, à partir du noeud initial, est effectuée en répétant les trois étapes suivantes:

`Etape 1` Si un noeud n, labellé par T_n et étiqueté par E_n, contient une formule f non-élémentaire et non-marquée, et si la règle de décomposition pour f est $f \rightarrow \{S_i\}$, alors pour chaque S_i, créer un fils de n étiqueté par E_n, et labellé par $(T_n - \{f\}) \cup S_i \cup \{f^*\}$, où f^* est f marquée.

`Etape 2` Si un noeud n, labellé par T_n et étiqueté par E_n, est un état, alors créer le ou les fils de n suivants:

- un fils labellé par $Def(n)$ et étiqueté par "défini".
- un fils labellé par $Indef(n)$ et étiqueté par "indéfini", à condition que l'étiquette E_n soit différente de "indéfini".

`Etape 3` Si un noeud n est un double-état, alors créer les deux fils de n suivants:

- un fils labellé par T_n et étiqueté par "τ".
- un fils labellé par T_n et étiqueté par "action".

Nous constatons que ces deux fils sont des états.

Après avoir construit le graphe, pour décider de la satisfiabilité de la formule initiale, nous devons éliminer les noeuds du graphe qui ne sont pas satisfiables. Pour ce faire, nous utilisons les règles d'élimination suivantes, qui indiquent à quelles conditions un noeud est éliminé.

`E1` Le noeud contient à la fois une formule atomique f et sa négation $\neg f$. Rappelons qu'une formule atomique est une proposition de la logique $\mathcal{L}_P$, une action $a.\alpha$ avec $\alpha \in \mathsf{Act}(a)$, une formule du style $a.\!\uparrow\!\alpha$ où $\alpha \in Meth(a)$, ou encore la formule **first**.

`E2` Le noeud contient $a.\alpha$ et $a.\beta$ où $\alpha, \beta \in \mathsf{Act}(a)$, et α et β sont distinctes.

`E3` Le noeud n est étiqueté avec "action" et, pour toute action $\alpha \in \mathsf{Act}(a)$, on a $\neg a.\alpha \in T_n$.

`E4` Le noeud n est étiqueté avec "τ" ou "indéfini" et, il existe une action $\alpha \in \mathsf{Act}(a)$, telle que $a.\alpha \in T_n$.

E5 Le noeud n est un noeud inactif initial et $\neg\mathsf{first} \in T_n$.

E6 Le noeud n n'est pas un noeud inactif initial et $\mathsf{first} \in T_n$.

E7 Tous les successeurs du noeuds ont été éliminés.

E8 Le noeud est un pré-état et contient une formule de la forme $\neg\Box f$ ou $\neg(f_1 \,\mathsf{unless}\, f_2)$ qui n'est pas satisfiable dans ce noeud. Nous dirons qu'une formule $\neg\Box f$ est satisfiable dans un pré-état n, si il existe, dans le tableau, un chemin originaire de ce pré-état menant à un noeud contenant la formule $\neg f$. De même, nous dirons qu'une formule $\neg(f_1 \,\mathsf{unless}\, f_2)$ est satisfiable dans un pré-état n, si il existe un chemin originaire de ce pré-état menant à un noeud contenant la formule $\neg f_1$.

La procédure de décision se termine lorsque tous les noeuds insatisfiables ont été éliminés. Si le noeud initial a été éliminé, alors la formule $\Diamond f$, présente dans le noeud initial, est insatisfiable dans l'état initial de toute structure temporelle. Nous pouvons donc conclure que la formule f initiale est insatisfiable.

Nous pourrions montrer le théorème suivant:

Théorème 2 *Une formule f de $\mathcal{L}_a$ est satisfiable si et seulement si le noeud initial du graphe généré par la méthode des tableaux appliquée à f n'est pas éliminé.*

4.2 Méthode des tableaux pour $\mathcal{L}_a$ avec la sémantique de programme

La méthode des tableaux développée dans la sous-section précédente correspond à la logique d'agent de base. Le but des paragraphes suivants est de décrire les modifications à apporter à la méthode décrite précédemment, pour qu'elle traite les formules de la logique prenant en compte la sémantique d'une exécution. Ces modifications sont les suivantes:

- Si f est la formule dont on veut tester la satisfiabilité et $\mathsf{first} \rightarrow init$ est l'axiome décrivant complètement l'état initial de l'agent a, alors nous labellons le noeud initial du graphe avec la formule $init \wedge \Diamond f$. Le principe est de connaître dans chaque noeud du graphe, la valeur de vérité de toutes les variables d'état de StVar(a). En d'autres termes, il faut garantir que tout noeud du graphe qui n'est pas éliminé contient soit p, soit $\neg p$, pour tout $p \in \mathsf{StVar}(a)$.
- A l'étape 3 de la construction du graphe, nous attribuons des fils à un double-état en appliquant l'algorithme ci-après dans lequel si F est un ensemble de formules, les ensembles $Req_\alpha(F)$ et $Req(F)$ sont définis par :

  $$Req_\alpha(F) = \{\bigcirc_a a.{\uparrow}\beta \mid \beta \in \mathsf{Meth}(a) \text{ et } a.{\uparrow}\beta \in F \text{ et } \beta \neq \alpha\}$$
  $$Req(F) = \{\bigcirc_a a.{\uparrow}\beta \mid \beta \in \mathsf{Meth}(a) \text{ et } a.{\uparrow}\beta \in F\}$$

  Ces ensembles permettent de propager les messages en attente.

  Algorithme:

  Soit n un double-état du graphe,

  - si il existe une action unique α telle que $a.\alpha \in T_n$ alors créer un fils unique s tel que:

* $E_s =$ "*action*"
* $T_s = \left\{ \begin{array}{ll} T_n \cup Req_\alpha(T_n) \cup \{a.\uparrow\alpha, P_\alpha, \bigcirc_a Post_\alpha\} & \text{si } \alpha \in \mathsf{Meth}(a) \\ T_n \cup Req_\alpha(T_n) \cup \{P_\alpha, \bigcirc_a Post_\alpha\} & \text{sinon} \end{array} \right\}$

– sinon si pour toute action $\alpha \in \mathsf{Act}(a)$, $\neg a.\alpha \in T_n$ alors créer un fils unique s tel que:
* $E_s =$ "τ"
* $T_s = T_n \cup Req(T_n)$

– sinon si $\forall \alpha \in \mathsf{Act}(a)$ $a.\alpha \notin T_n$, effectuer les étapes 1, 2 et 3 suivantes
1. créer un fils s tel que:
   * $E_s =$ "τ"
   * $T_s = T_n \cup Req(T_n)$
2. pour chaque action $\alpha \in \mathsf{Meth}(a)$ telle que $\neg a.\alpha \notin T_n$ créer un fils s tel que:
   * $E_s =$"*action*"
   * $T_s = T_n \cup Req_\alpha(T_n) \cup \{a.\alpha, a.\uparrow\alpha, P_\alpha, \bigcirc_a Post_\alpha\}$
3. pour chaque action $\alpha \in \mathsf{Refl}(a)$ telle que $\neg a.\alpha \notin T_n$ créer un fils s tel que:
   * $E_s =$ "*action*"
   * $T_s = T_n \cup Req(T_n) \cup \{P_\alpha, \bigcirc_a Post_\alpha\}$

Dans l'algorithme, si α est une action décrite par $\alpha =$ **pre** P_α **act** $\vec{v_\alpha} := \vec{e_\alpha}$, $Post_\alpha$ est la conjonction des littéraux de $\vec{v_\alpha}$ selon leur valeur après l'affectation.

- Nous modifions la définition de $Def(n)$ de la manière suivante:

$$Def(n) = T_n^{next} \cup T_n^{next_a} \cup \gamma_{prop}(n) \cup mes(n)$$

Si n est un noeud du graphe, nous noterons *état_père(n)* le noeud qui est l'état père de n dans le graphe, si ce noeud existe, et nous définissons l'ensemble *mes(n)* par:

$$mes(n) = \left\{ \begin{array}{l} \{a.\uparrow\alpha \mid a.\uparrow\alpha \in T_n \text{ et } a.\uparrow\alpha \notin T_p, \text{ p = état\_père(n) } \} \\ \qquad \text{si } E_n = \text{indéfini et } E_p = \text{action, où p=état\_père(n)} \\ \emptyset \qquad \text{sinon} \end{array} \right\}$$

5 Conclusion

Dans cet article, nous avons présenté une partie de notre concept d'objet concurrent; et nous l'avons formalisé dans un cadre logique possédant une procédure de décision.

Comme nous l'avons dit dans l'introduction, le travail présenté ici s'inscrit dans un cadre plus large, celui de la réutilisation, aussi bien au niveau du concept (réutilisation de classe, par héritage, ou d'agent, par composition parallèle) qu'au niveau de la formalisation (réutilisation des descriptions formelles et des propriétés).

Nous avons déjà obtenu des résultats intéressants dans ce cadre. Par exemple, dans notre concept, le problème de l'anomalie de l'héritage ne se pose pas, sur les exemples classiques de la littérature; les contraintes de synchronisation définies pour les actions d'une classe sont réutilisables dans d'autres classes, grâce aux *classes de synchronisation*. En ce qui concerne le formalisme, le découpage en trois niveaux de logique permet de réutiliser dans une sous-classe, certaines propriétés montrées dans une classe parente. De même des propriétés d'un système d'agents sont démontrables à partir de celles des agents constituants.

Cependant de nombreux points demeurent ouverts: dans quelle mesure peut-on considérer une logique du niveau système d'agent comme une logique d'agent? La réponse à cette question nous permettrait de construire une hierarchie d'agents qui pourrait elle-même être considérée comme un agent. Peut-on établir une classification des différentes formes d'héritage, et déduire dans chaque cas, quelles propriétés d'une classe parente sont conservées dans une classe fille?

References

[AH87] G. Agha and C. Hewitt *Concurrent Programming Using Actors*, In "Object-Oriented Concurrent Programming", (ed) A. Yonezawa and M. Tokoro, MIT Press, 1987, 37-53.

[Bak91] J. de Bakker, W. de Roever and G. Rozenberg (eds) *Foundations of Object-Oriented Languages*, LNCS 489, Springer-Verlag 1991.

[BMS93] J. P. Bahsoun, S. Merz, C. Servieres *A Framework for Programming and Formalizing Concurrent Objects* Proceedings of ACM Sigsoft 93

[BMS94] J. P. Bahsoun, S. Merz, C. Servieres *A Framework for formalizing and proving concurrent objects* WS 4 ECOOP'94, July 1994-Bologna Italy.

[FM92] J.Fiadeiro, T.Maibaum *Temporal Theories as Modularisation Units for Concurrent System Specification*, Formal aspects of computing 4(3), 1992, 239-272

[Krö87] F. Kröger: *Temporal Logic of Programs*. EATCS Monographs on Theoretical Computer Science **8**. Berlin: Springer-Verlag (1987)

[Lam91] L. Lamport: *The Temporal Logic of Actions*. Research Report 79, DEC Systems Research Center, December 1991

[MP92] Z. Manna, A. Pnueli: *The Temporal Logic of Reactive and Concurrent Systems — Specification*. New York etc.: Springer-Verlag (1992)

[Mer93] S. Merz: *Three aspects of objects systems: On the formal specification of concurrent objects*. Rapport interne IRIT(1993)

[MW92] J. Meseguer, T. Winkler *Parallel Programming in Maude*. In "Research Directions in High-Level Parallel Programming Languages" J.B. Banatre, D. LeMetayer (Eds) Springer Verlag 1992

[Mey88] B. Meyer *Object oriented software construction*, Prentice Hall 1988.

[Ser94] C. Servières *Une procédure de décision pour la logique locale $\mathcal{L}_a$*. Rapport interne IRIT (1994)

[Sm68] R.M. Smullyan: *First Order Logic*, Springer-Verlag Berlin 1968.

[Wo85] P. Wolper: *The tableau method for temporal logic: an overview.* Logique et Analyse 110-111 numéro spécial, Nauwelaerts Printing, 1985.