Rapport de projet annuel de 1ère année
de Master Informatique
Année scolaire 2021-2022

Développement d'un interpréteur de programmes à base de connaissances multi-agents

Université de Caen, Normandie
UFR des Sciences

Projet réalisé par
Guillaume LETELLIER (21804030)
Corentin PIERRE (21803752)

Projet encadré par
Bruno ZANUTTINI

Table des matières

Table des figures

Liste des algorithmes

1 Introduction

1.1 Contexte

Dans le cadre de notre première année en Master Informatique à l'Université de Caen Normandie, il nous a été proposé de réaliser un projet annuel en binôme afin de nous permettre de mettre en pratique nos capacités de développement et de raisonnement sur un projet.

Différents projets ont été soumis en septembre par différents professeurs pour que nous, étudiants, puissions les classer afin qu'un algorithme choisisse au mieux les sujets en fonction des demandes des groupes. À la suite de cela, nous avons eu le projet de M. ZANUTTINI intitulé "Interpréteur de programmes à base de connaissances multi-agents" comme projet annuel pour cette année 2021-2022.

1.2 But du projet

Il nous a été demandé d'implémenter un interpréteur, un simulateur de programmes multi-agents, et cela dans la langage orienté objet *Java*. Ces programmes sont des programmes à base de connaissances (KBP pour Knowledge-Based Programs), c'est-à-dire qu'en fonction des branchements dans le programme d'un agent, il effectuera une action plutôt qu'une autre grâce aux connaissances qu'il possède. Le simulateur doit pouvoir prendre en entrée un environnement, un ensemble d'agents qui possèdent chacun un programme à exécuter, et ainsi pouvoir lancer l'exécution globale de l'environnement dans laquelle chaque agent agit selon son programme.

Le développement d'un mécanisme de raisonnement sur des connaissances a donc été la toute première chose réalisée. Un tel système doit être capable de maintenir les connaissances d'un agent sur son environnement à un instant t en fonction des actions des autres agents. L'interpréteur a été la seconde partie réalisée venant ainsi se greffer à la première. Elle permet d'encapsuler les structures de raisonnement pour chaque agent, faciliter les appels pour une collection d'agents, etc.

Ce projet a pour vocation à compléter une bibliothèque de code créée par l'équipe MAD (Modèles, Agents, Décision) du laboratoire du GREYC à l'Université de Caen Normandie.

2 Problème principal à résoudre

Le projet étant complexe à première vue, il nous a été présenté sous la forme d'un problème, qui nous a suivi tout du long du développement afin de le résoudre avec notre interpréteur à la fin du projet. Nous avons donc décidé d'expliquer le projet sur ce même problème en tant que première approche. Les notions logiques et algorithmiques seront présentées plus en détail dans la section 4.

2.1 Présentation du problème : le problème des enfants sales

L'énoncé est le suivant : des enfants partent jouer en extérieur et certains d'entre eux reviennent avec de la saleté sur leur front. La mère (ou le père, dépendant des versions) répète "au moins un de vous est sale" jusqu'à ce que tous les enfants qui sont sales se dénoncent. Cependant, les enfants propres ne doivent pas se dénoncer. Il est intéressant de noter que chaque enfant peut voir les fronts des autres enfants mais ne peut pas voir son propre front et donc, cela implique que chacun des enfants connaît les états des autres enfants mais aucunement son propre état.

Il faut donc réaliser un mécanisme de raisonnement dans lequel un agent peut analyser les actions des autres agents afin de connaître, en partie, leurs connaissances pour se rapprocher du monde réel afin d'exécuter une certaine action en fonction de son programme. Dans cet exemple, un enfant voyant que d'autres ne se dénoncent pas, peut en déduire qu'il y a un autre enfant qui est sale (que ce soit lui-même, ou un autre dont il connaît déjà son état).

2.2 Exemple sur le problème principal

Généralités On note la proposition a "l'agent A est sale". Chaque agent possède comme programme l'algorithme 1 où il doit être adapté pour chaque agent. $K_A\, a$ représente le fait que l'agent A sait qu'il est sale.

Algorithme 1 Programme de l'agent A

Entrée: un modèle de Kripke $\mathcal{M}$ et un monde pointé w
  si $K_A\, a$ **alors**
    Se dénoncer
  sinon
    Se taire
  fin si

Initialisation Pour n'importe quel problème de raisonnement, nous pouvons créer une structure $\mathcal{M}$, dite de Kripke, permettant de représenter l'ensemble des mondes possibles à un instant t. Elle permet de garder à un seul endroit les connaissances d'un agent sur son environnement, que ce soit ses observations ou les déductions des connaissances des autres

agents en fonction des actions qu'ils ont exécuté. Cette structure peut être mise à jour en formulant une annonce publique φ restreignant ainsi cette structure. Informatiquement, elle est représentée par un graphe orienté où chaque sommet représente un monde possible et les arcs représentent l'hésitation d'un ensemble d'agents entre les deux mondes reliés par l'arc considéré.

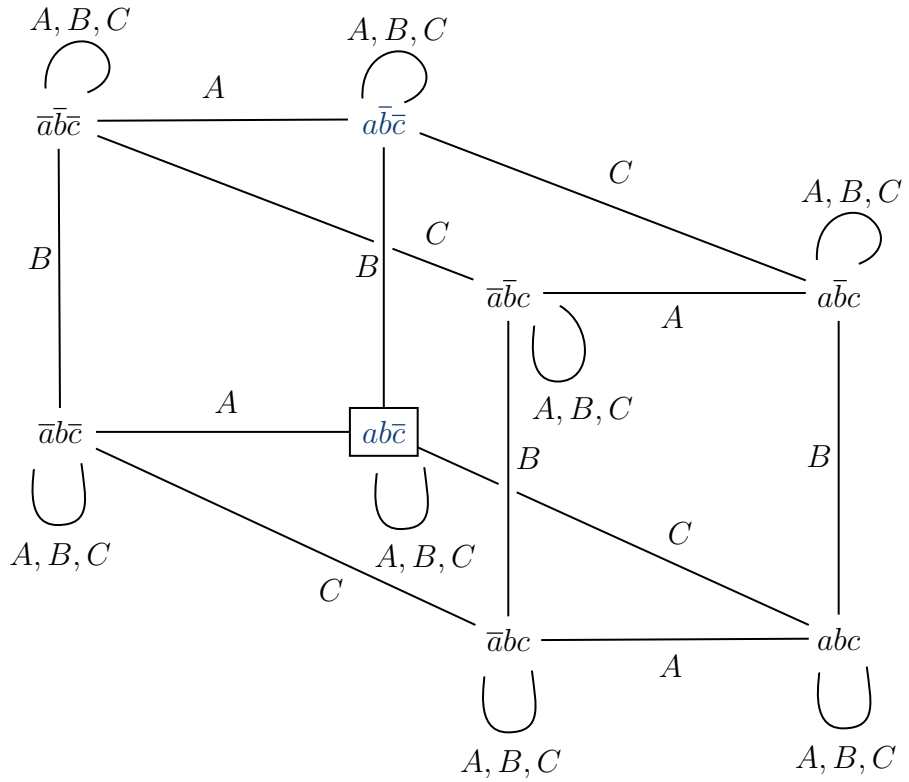

FIGURE 1 – Structure initiale avec arcs réflexifs

La figure 1 montre la structure initiale pour le problème des enfants sales avec trois enfants. Ici, pour illustrer l'exemple, nous avons choisi de considérer les agents A et B sales et C propre (le monde réel a été encadré afin de mieux le distinguer). On remarque que chaque monde possède une arc réflexif permettant d'utiliser la logique des connaissances $\mathbb{S}_5$ (voir section 4.2.2 sur la logique modale et épistémique). Ce $\mathbb{S}_5$ signifie que dans le modèle de Kripke, les relations d'accessibilité sont des relations d'équivalences (réflexives, transitives et symétriques). Les relations étant symétriques, nous les avons représentés par des arêtes et non des arcs. Aussi, étant donné que les arcs réflexifs existent pour n'importe quel problème de logique des connaissances $\mathbb{S}_5$ pouvant être modélisé, nous remplacerons par la suite cette figure par la figure 2 afin de simplifier leur compréhension.

On peut remarquer que les mondes $a\bar{b}\bar{c}$ et $ab\bar{c}$ sont colorés en bleu. Ce sont les deux seuls mondes possibles pour l'agent B avec les observations initiales $(a \wedge \bar{c})$ qu'il obtient en regardant les autres agents. Nous pouvons choisir l'un des deux mondes en tant que monde de référence pour l'évaluation de formules logiques car ils sont tous deux équivalents en fonction des connaissances relatives à cet agent. Si nous étions du point de vue de l'agent

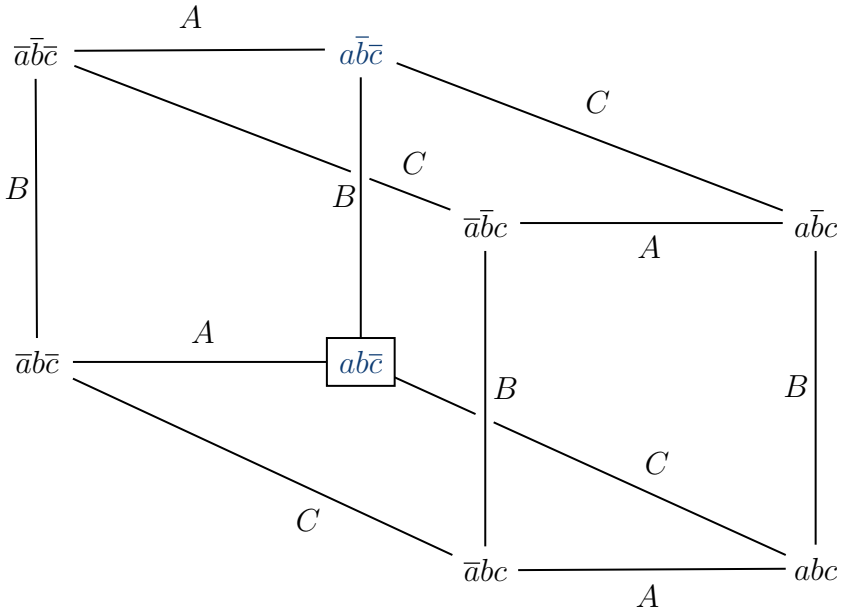

FIGURE 2 – Structure de Kripke initiale

C, il aurait fallu considérer le monde $ab\overline{c}$ ou abc car il voit A et B sales et hésite donc uniquement sur son propre état. On représentera le monde de référence avec une flèche arrivant sur celui-ci pour le distinguer des autres mondes. Il sera appelé par la suite le monde pointé.

Tour 1 On commence le premier tour. La mère rencontre ses enfants et dit "au moins un de vous est sale". Cette affirmation peut être modélisée par la formule suivante $\varphi = a \vee b \vee c$. Ceci étant une annonce publique, cela implique une mise à jour de la structure $\mathcal{M}$ par l'annonce φ.

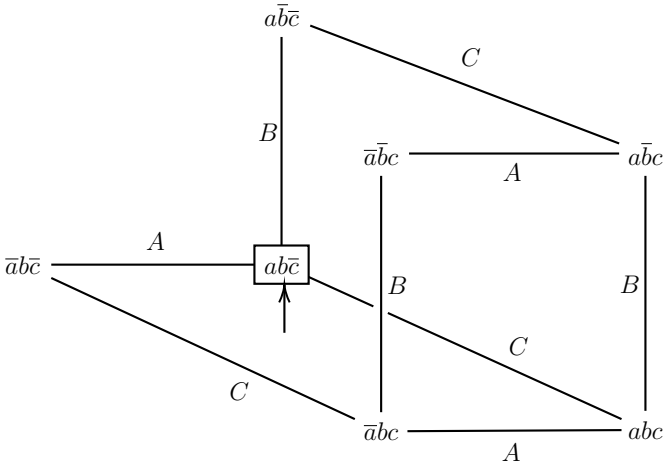

FIGURE 3 – Structure au tour 1 après l'annonce $a \vee b \vee c$

On obtient la figure 3 après cette annonce. L'annonce publique a uniquement supprimé le monde où tous les agents sont propres car il y en a au moins un d'entre eux qui est sale. Ce monde est donc impossible. À noter que les arêtes reliant ce monde ont aussi été supprimées car si un monde est jugé impossible, il ne doit plus être relié à d'autres qui eux le sont encore au sein du modèle. Actuellement, cette structure est partagée pour tous les agents car ils ont tous reçu la même annonce.

On peut maintenant exécuter le programme de l'agent B (et des autres parallèlement) sur le modèle $\mathcal{M}$ et le monde pointé w (ici $w = ab\bar{c}$). À partir de w, les seuls mondes accessibles sont w, du à l'arc réflexif, et le monde $a\bar{b}\bar{c}$. On peut en déduire que B ne sait pas qu'il est sale et d'après son programme (algorithme 1), il entre dans le sinon et donc se tait. Les autres agents font de même. Cela est logique car l'agent B voit A sale donc ne se dénonce pas en pensant que A se dénoncerait.

Désormais, il faut évaluer les connaissances de l'agent B à partir du modèle $\mathcal{M}$, du monde pointé w et des actions que les autres agents ont effectué. Nous avons donc aucun agent qui s'est dénoncé impliquant qu'aucun ne sait qu'il est sale (mais ils pourraient très bien savoir qu'ils sont propres néanmoins ici, ce n'est pas le cas). Nous passons les formules contenues dans les branchements à partir des actions que les autres agents ont exécuté. Cependant, il faut vérifier que l'agent considéré connaisse le programme des autres agents. Dans ce problème, tous les agents connaissent le programme des autres agents impliquant donc que $K_B\,(\neg K_A\,a \wedge \neg K_C\,c)$ (l'agent B sait que l'agent A ne sait pas qu'il est sale et que l'agent C ne sait pas qu'il est sale). On peut réaliser un raisonnement analogue pour les deux autres agents et nous obtenons pour l'agent A $K_A\,(\neg K_B\,b \wedge \neg K_C\,c)$ et pour l'agent C $K_C\,(\neg K_A\,a \wedge \neg K_B\,b)$.

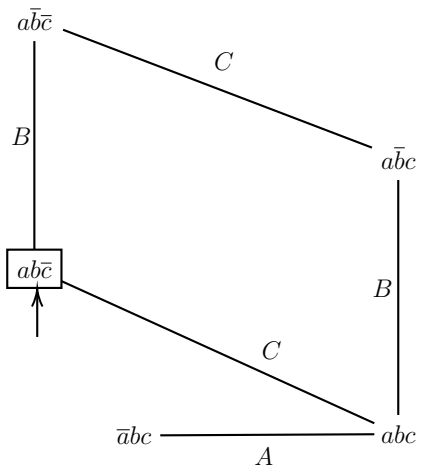

FIGURE 4 – Structure de l'agent A après l'annonce $K_A\,(\neg K_B\,b \wedge \neg K_C\,c)$

On réalise des annonces publiques séparées pour chaque agent car, pour rappeler, chaque agent à sa vision de la réalité et donc sa propre structure. Ainsi, cela semble logique de différencier les structures pour formuler des annonces différentes pour chaque agent. On obtient à la fin de ce tour les figures 4, 5 et 6.

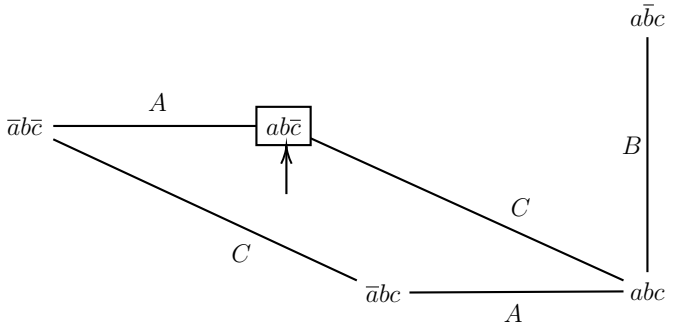

FIGURE 5 – Structure de l'agent B après l'annonce $K_B\,(\neg K_A\, a \wedge \neg K_C\, c)$

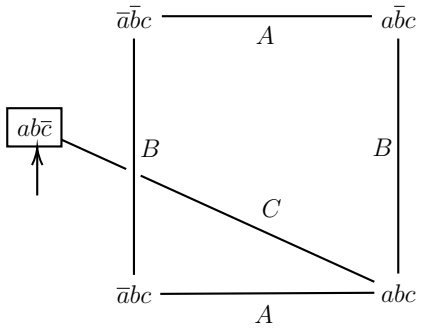

FIGURE 6 – Structure de l'agent C après l'annonce $K_C\,(\neg K_A\, a \wedge \neg K_B\, b)$

Tour 2 On continue avec un deuxième tour. La mère, voyant qu'aucun des enfants se sont dénoncés, répète "au moins un de vous est sale" ($\varphi = a \vee b \vee c$). On annonce cette formule à chaque agent et chacun d'eux modifie leur modèle. Aucun autre monde ne peut être enlevé pour chaque agent car tous les mondes restants respectent cette formule. On peut donc passer aux actions que les agents vont choisir.

Pour l'agent B, on observe sur la figure 5 qu'à partir du monde pointé, aucun autre monde n'est accessible pour l'agent B dans le modèle à part le même monde grâce à l'arc réflexif. Ici, la formule $K_B\, b$ est vraie et donc entre dans la première condition de son programme. Il exécute l'action associée qui est de "se dénoncer". Nous pouvons faire un raisonnement analogue pour l'agent A.

Cependant pour l'agent C, savoir que les agents A et B savent qu'ils sont sales revient exactement à connaître leurs états. Or, il les connaît déjà grâce au monde pointé donc on ne peut plus supprimer de mondes supplémentaires dans la structure associée à cet agent. Il est utile de spécifier ici que l'agent C peut savoir qu'il est propre soit en donnant le monde réel à C quand tous les objectifs de l'exécution de l'environnement sont validés, ou en passant à une logique d'ordre supérieur (voir 5.3 sur les axes d'amélioration).

Conclusion Les agents sales se sont bien dénoncés et les agents propres se sont bien tus. D'après le théorème énoncé sur cette page [1], les enfants sales sont censés se dénoncer au k-ième tour où k est le nombre d'enfants sales. Dans notre cas, nous avions $k = 2$ et les agents sales se sont dénoncés au tour 2, donc bien au k-ième tour.

3 Gestion du projet

3.1 Outils

Comme pour tout projet, de nombreux outils nous ont permis de faciliter le développement de celui-ci.

Tout d'abord, nous avons utilisé le gestionnaire de versions de code source Git, hébergé sur la forge [1]. Il nous permet de traquer les différentes versions de notre projet, revenir à une version ultérieure, ou encore travailler en coopération entre les membres du groupe. Notre dépôt est disponible en cliquant sur ce lien.

Ensuite, l'utilitaire Ant de Apache, nous a été très utile pour automatiser les lignes de commandes dans le but de compiler et distribuer rapidement notre projet, exécuter notre démonstration, générer la documentation ou encore lancer les tests unitaires. Les quelques commandes importantes sont décrites dans le `README.md` à la racine du projet.

3.2 Documentation et tests unitaires

Nous avons réalisé une documentation assez complète afin que les développeurs qui utiliseront notre projet puissent rapidement comprendre l'utilité de chaque méthode dans chacune des classes créées.

Dans chaque paquetage du projet, il s'y trouve un sous-paquetage dédié aux tests unitaires du paquetage considéré. En effet, les tests unitaires tout comme la documentation, font partie intégrante du développement d'un projet car ils permettent de vérifier que les algorithmes implémentés réalisent bien ce qu'ils sont censés faire à l'aide de la documentation.

3.3 Utilisation du temps

La figure 7 présente les période de temps de développement de chacune des grandes parties de notre projet à travers les six derniers mois.

1. https://forge.info.unicaen.fr/

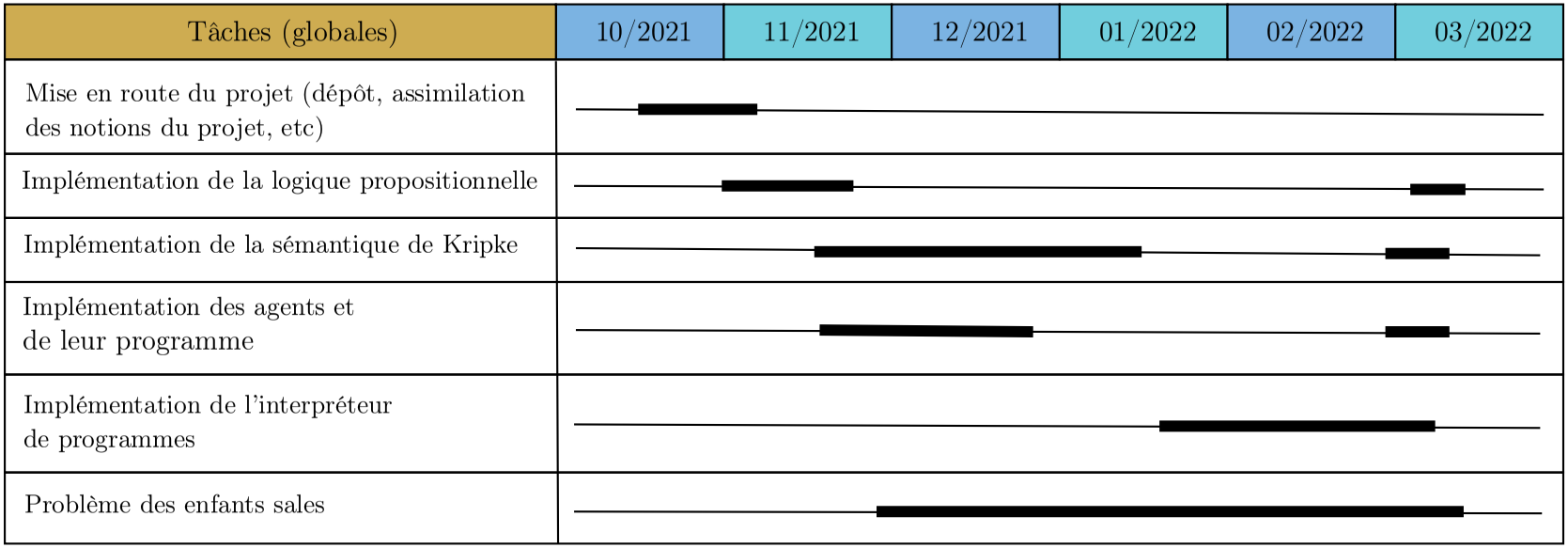

FIGURE 7 – Diagramme de Gantt

4 Développement du projet

Le projet étant conséquent (entre 6 000 et 7 000 lignes de codes incluant les tests unitaires et la documentation), nous avons choisi de ne représenter que les classes et interfaces importantes de notre projet. Les algorithmes importants seront expliqués dans cette section à partir de propositions et théorèmes mathématiques de la logique propositionnelle et modale, et des besoins pour la réalisation de l'interpréteur.

Structuration Nous avons décidé de découper notre projet en plusieurs sous-paquetages afin de mieux généraliser et encadrer les différentes classes dans des paquetages qui ne sont utiles que dans certaines parties du projet. La figure 8 présente le diagramme de classes mis en oeuvre.

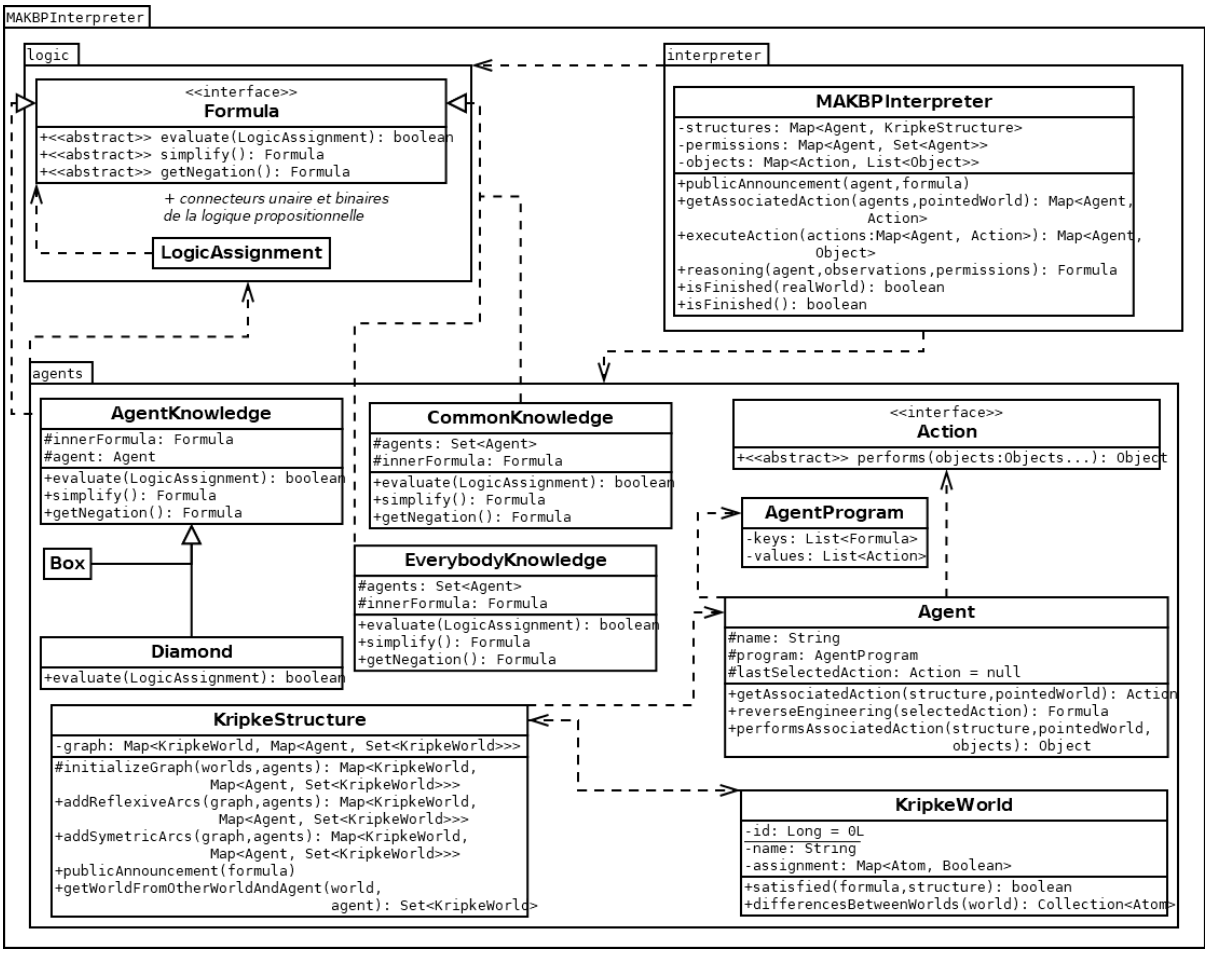

FIGURE 8 – Diagramme de classes

Il est utile de noter que nous n'avons présenté que les méthodes demandant une explicitation car certaines ont été créées dans le but de faciliter l'implémentation et l'utilisation des objets dans Java comme `equals` ou encore des surcharges de méthodes

pour différentes structures de données.

Pour rappel, sur la figure 8, les flèches en pointillé expriment une relation de dépendance (dans le sens de flèche, on a par exemple le paquetage `interpreter` qui dépend des paquetages `logic` et `agents`). Les fléches pleines expriment un héritage d'une autre classe et les flèches en pointillé avec un triangle plein expriment un héritage d'une interface.

4.1 Logique propositionnelle

Quelques rappels simples de logique propositionnelle [2] seront fait afin de faciliter la compréhension des autres parties et de l'implémentation de celle-ci.

4.1.1 Rappels de logique

Pour commencer, nous allons rapidement introduire la logique propositionnelle car elle est la base de tout raisonnement logique.

Dans cette logique, il existe des propositions dites atomiques (que l'on ne peut plus scinder en plus petites parties). On peut créer des calculs propositionnels en les liant avec différents connecteurs logiques (unaires, binaires, ternaires et nullaires). Nous n'avons choisi de ne représenter que les opérateurs de négation ($\neg$), de conjonction ($\wedge$), de disjonction ($\vee$) d'implication ($\rightarrow$) et d'équivalence ($\leftrightarrow$) car les autres opérateurs peuvent être créés à partir de ceux-là, ou sont facilement gérés dans l'implémentation réalisée. Par exemple, nous pouvons construire une formule $(p \wedge q) \vee r$ qui est propositionnelle.

Chaque formule propositionnelle peut être évaluée en passant $VRAI$ ou $FAUX$ pour chaque atomes constituant la formule. Certaines formules peuvent être simplifiée, développée ou mis en négation. Voici quelques règles de base dans cette logique :

- $VRAI \wedge a \equiv a$;
- $FAUX \vee a \equiv a$;
- $\neg(\neg a) \equiv a$ (involution) ;
- $\neg(a \wedge b) \equiv (\neg a \vee \neg b)$;
- $\neg(a \vee b) \equiv (\neg a \wedge \neg b)$;
- $(a \rightarrow b) \equiv (\neg a \vee b)$;
- $(a \leftrightarrow b) \equiv ((a \rightarrow b) \wedge (b \rightarrow a))$;
- lois distributives pour $\wedge$ et $\vee$;
- ...

4.1.2 Implémentation

Concernant l'implémentation de la logique propositionnelle, nous avons choisi de représenter les atomes et chacun des connecteurs logiques par des classes dérivants de l'interface `Formula`. Elle permet de donner une base commune concernant les méthodes à implémenter et cela pour tous les éléments essentiels dans une formule propositionnelle. Les quelques objets présentés par la suite nécessitent une explication supplémentaire quant à leur implémentation.

Interface Formula Cette interface définit une liste de méthodes qui doivent être implémentées par toutes classes dérivant de cette interface. Comme à la main, on peut choisir d'évaluer une formule à partir d'un ensemble d'atomes qui sont valués à $VRAI$ ou $FAUX$, mais aussi simplifier une formule ou encore créer sa négation.

Deux autres méthodes peuvent être mentionnées mais ne seront pas développées, ce sont celles qui aideront pour l'implémentation comme la méthode d'égalité ou de contenance d'un atome dans une formule.

Ces trois méthodes principales permettent de réaliser l'ensemble des opérations logiques. Il est intéressant de noter ici que nous n'avons pas créer de méthode pour développer des formules car développer une formule à la main peut être utile notamment pour faire des simplifications, mais pour une machine, cela impliquerait d'utiliser plus de mémoire et réaliser des calculs supplémentaires lors des évaluations de formules logiques.

Il est utile de notifier que la méthode d'évaluation prend en argument une instance de l'interface *LogicAssignment*. On peut passer différentes classes en fonction du type de formule évaluée comme $PropositionalLogicFormula$ ou bien $ModalLogicFormula$.

Atomes Pour les représenter, nous avons donc créer une classe très simple nommée `Atom`. Elle permet de stocker une proposition au sein d'une formule logique.

Les opérateurs de conjonction et disjonction Dans les définitions de ces opérateurs, ils sont présentés comme étant binaires et donc ne pouvant prendre en entrée que deux formules. Nous avons utiliser deux propriétés communes à ces opérateurs qui sont l'associativité et la commutativité. En effet, sachant cela, on peut simplifier leur implémentation en un simple ensemble de formules. Ceci évite ainsi d'avoir des formules inutilement dupliquées (idempotence) et ainsi réduire l'espace mémoire et le temps de calcul (même si la différence est à peine perceptible).

4.2 Logique épistémique et mécanisme de raisonnement

La logique modale est un type de logique mathématiques qui étend la logique propositionnelle avec des opérateurs modaux. Ces opérateurs modaux peuvent être "*je sais que*", "*je crois que*", "*j'envisage que*", etc. Dans ce type de logique, on peut créer deux opérateurs qui sont $\Box$ ("box" représentant la nécessité) et $\Diamond$ ("diamond" représentant la possibilité). Dans notre cas, nous raisonnons uniquement sur des connaissances et donc le savoir. On nomme cette logique la logique épistémique [3].

4.2.1 Syntaxe

Afin de poser les bases pour la suite, il nous faut établir une syntaxe pour notre langage permettant de réaliser des formules de la logique épistémique [4]. Cette logique est engendrée par la grammaire suivante :

$$\varphi, \theta, ... := p \mid \neg\varphi \mid (\varphi \wedge \theta) \mid (\varphi \vee \theta) \mid (\varphi \rightarrow \theta) \mid (\varphi \leftrightarrow \theta) \mid K_a\,\varphi \mid EK_J\,\varphi \mid CK_J\,\varphi \quad (1)$$

où $p \in P = \{p_i \mid i \in \mathbb{N}\}$ un ensemble de propositions, $a \in A = \{a_i \mid i \in \mathbb{N}\}$ un ensemble d'agents et $J \in \bigcup_{i \in |A|} i^A$ l'ensemble de tous les groupes d'agents pouvant être créés.

$K_a\,\varphi$ est une modalité signifiant que l'agent a sait que la formule φ est vraie. Quelques règles de simplification sont utiles d'être spécifiées :

— $K_a\,\varphi \implies \varphi$;
— $K_a\,\varphi \implies K_a\,K_a\,\varphi$;
— $\neg\,K_a\,\varphi \implies K_a \neg\,K_a\,\varphi$;

$EK_J\,\varphi = \bigwedge_{a \in J} K_a\,\varphi$ est une modalité signifiant qu'un ensemble d'agents nommé J savent que la formule φ est vraie pour chacun d'entre eux (connaissance partagée).

$CK_J\,\varphi = \bigwedge_{i=1}^{\infty} EK_J^i(EK_J^{i-1}(\varphi))$ avec $EK_j^0\,\varphi = \varphi$ est une modalité signifiant qu'un ensemble d'agents nommé J savent que la formule φ est vraie, pour toute profondeur finie (connaissance commune).

4.2.2 Modèle de Kripke : sémantique de notre langage

Comme pour la logique propositionnelle, on peut définir une sémantique, une façon d'évaluer les formules de la logique épistémique (et plus globalement modale).

Modèle de Kripke On peut représenter notre connaissance d'un environnement par une structure de données nommée **modèle de Kripke**. Concrètement, c'est un graphe orienté où les sommets sont des mondes possibles (qui peuvent être réels ou non) et les arcs sont des hésitations d'un ou plusieurs agents à propos des deux mondes liés par l'arc considéré. Ici, les arcs sont remplacés par des arêtes car dans ce projet, nous avons principalement raisonné sur la logique des connaissance $\mathbb{S}_5$ qui modifie les relations d'accessibilité par des relations d'équivalence qui doivent être par définition réflexives (d'où les arcs réflexifs sur les états), symétriques (d'où les arêtes au lieu des arcs) et transitives. Il existe d'autres logiques des connaissances comme la complétude de $\mathbb{S}_4$ où les relations sont des relations de préordre (réflexives et transitives) ou bien encore la complétude de $\mathbb{T}$ où les relations d'accessibilité sont uniquement réflexives (si une proposition est valide, alors elle est prouvable dans $\mathbb{T}$).

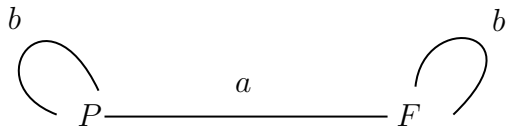

FIGURE 9 – Modèle de Kripke simple

La figure 9 permet de visualiser très facilement ce genre de structure avec un exemple de lancement d'une pièce de monnaie. En effet, sur cette figure on voit que l'agent a hésite entre les deux faces de la pièce. Quant à l'agent b, que le monde P ou F soit réel, il connaît le résultat du lancement et n'hésite aucunement entre deux mondes différents représentés au sein de cette structure. Ces arêtes sont dites réflexives. Elles permettent de continuer à évaluer une formule de la logique modale même s'il n'y a plus d'hésitations

avec un monde autre que celui en cours d'évaluation. Cela implique donc que l'agent hésitant uniquement avec le même monde est un agent qui sait.

On peut donc formellement modéliser cette structure par un aspect mathématique. Appelons un modèle de Kripke $\mathcal{M} = (W, R, V)$ où W est un ensemble non vide de mondes, $R : A \to 2^{W \times W}$ est un ensemble d'arêtes reliant deux mondes et la fonction de valuation $V : W \to 2^P$ qui pour chaque monde, associe un ensemble de propositions vraies (il est néanmoins conseiller de mettre les fausses afin d'avoir toutes les propositions sur le monde considéré afin de mieux le comprendre et interpréter les résultats des évaluations).

Évaluation Nous avons donc les formules de la logique épistémique et la structure sur laquelle les évaluer, mais voyons désormais quelques règles très simples qui ont conduit la création des algorithmes d'évaluation de ces mêmes formules.

Soit un monde $w \in W$. On appelle $\mathcal{M}, w$ un modèle pointé où une structure $\mathcal{M}$ possède un monde pointé w. On peut définir plusieurs règles pour le langage détaillé dans l'équation 1 :

- $\mathcal{M}, w \models p$ ssi $p \in V(w)$ (la proposition p est vraie dans le monde w) ;
- $\mathcal{M}, w \models \neg\, \varphi$ ssi $\mathcal{M}, w \not\models \varphi$ (la formule φ est fausse dans le monde w) ;
- $\mathcal{M}, w \models (\varphi \wedge \theta)$ ssi $\mathcal{M}, w \models \varphi$ et $\mathcal{M}, w \models \theta$ (les formules φ et θ sont vraies dans le monde w) ;
- $\mathcal{M}, w \models (\varphi \vee \theta)$ ssi $\mathcal{M}, w \models \varphi$ ou $\mathcal{M}, w \models \theta$ (si au moins une des formules entre φ et θ est vraie dans le monde w) ;
- $\mathcal{M}, w \models (\varphi \to \theta)$ ssi $\mathcal{M}, w \models \neg\, \varphi$ ou $\mathcal{M}, w \models \theta$ (l'implication est vraie dans le monde w) ;
- $\mathcal{M}, w \models (\varphi \leftrightarrow \theta)$ ssi $\mathcal{M}, w \models (\varphi \to \theta)$ et $\mathcal{M}, w \models (\theta \to \varphi)$ (l'équivalence est vraie dans le monde w) ;
- $\mathcal{M}, w \models K_a\, \varphi$ ssi $\forall u \in R_a(w),\ \mathcal{M}, u \models \varphi$ (un agent a sait que φ est vraie dans le monde w si dans tous les mondes u accessibles par l'agent a depuis w, la formule φ est vraie) ;
- $\mathcal{M}, w \models EK_J\, \varphi$ ssi $\forall u \in \left(\bigcup_{a \in J} R_a(w)\right),\ \mathcal{M}, u \models \varphi$ (tous les agents du groupe J savent que φ est vraie dans le monde w si dans tous les mondes u accessibles par les différents agents depuis w, la formule φ est vraie).
- $\mathcal{M}, w \models CK_J\, \varphi$ ssi $\forall u \in \left(\bigcup_{a \in J} R_a(w)\right)^*,\ \mathcal{M}, u \models \varphi$ (tous les agents du groupe J savent que φ est vraie dans le monde w si dans tous les mondes u accessibles par les différents agents depuis w, la formule φ est vraie, et cela pour toute profondeur)[5].

Les premières règles correspondant à la logique propositionnelle, n'ont donc rien de changer mais permettent de comprendre le système d'évaluation d'un modèle pointé. Les évaluations importantes ici sont celles correspondant à la logique épistémique. En effet, on voit qu'on utilise vraiment la structure car on évalue une même formule sur différents mondes u reliés à celui pointé (w) afin de vérifier la connaissance d'un seul ou d'un ensemble d'agents.

Si l'on reprend la formule de connaissance d'un agent a sur le modèle de la figure 9. On voit que a hésite entre les deux mondes donc lors de l'évaluation d'une formule $\varphi = P$, le résultat de l'évaluation pour le monde P sera vrai mais pour F sera faux,

impliquant donc d'après la règle exprimée ci-dessus, que le résultat est faux et donc que l'agent a ne sait pas que P est vrai. Si nous n'avions que le monde P, a aurait su que P est vrai car le seul monde avec lequel il hésite (qui est le même) renvoie vrai.

Pour la formule de connaissance commune, $(EK)^*$ signifie que pour toutes les profondeurs (limité par la taille du modèle de Kripke considéré), les connaissances partagées par le groupe J sont vraies (avec φ comme formule quand la profondeur est à 0). Par exemple, une connaissance partagée de ε sur une profondeur de 2 donnera la formule suivante à évaluer :

$$CK_J^2\,\varepsilon = \varepsilon \wedge EK_J\,\varepsilon \wedge EK_J\,EK_J\,\varepsilon$$

Annonces publiques Une annonce publique sur un modèle de Kripke est une formule φ que l'on donne au modèle. Ce type de formule appartient certes à la logique épistémique, mais à sa sous-catégorie dite "logique épistémique dynamique". On l'appelle "dynamique" car le résultat de l'application de cette formule est un modèle de Kripke dit *restreint par* φ qui ne contient que les mondes u tels que $\mathcal{M}, u \models \varphi$. L'annonce modifie la structure et la rend donc dynamique. À partir de là, nous pouvons restreindre les différents ensembles composants un modèle de Kripke afin de ne garder que les relations et les propositions pour la formule annoncée.

Pour aller plus loin, on peut réaliser du *model checking*. Cette méthode permet de vérifier la validité d'une annonce dans un modèle pointé. Ainsi, on peut réduire la taille du modèle. On peut désormais ajouter les annonces publiques à la grammaire de notre langage. On obtient donc :

$$\begin{aligned}\varphi, \theta, ... :=& p \mid \neg\varphi \mid (\varphi \wedge \theta) \mid (\varphi \vee \theta) \mid (\varphi \rightarrow \theta) \mid (\varphi \leftrightarrow \theta) \mid \\ & K_a\,\varphi \mid EK_J\,\varphi \mid CK_J\,\varphi \mid [\varphi!]\theta\end{aligned} \tag{2}$$

$[\varphi!]\theta$ signifie que si la formule φ est vraie, alors après avoir annoncé φ publiquement, θ est vraie aussi. Ce type de formules peut permettre de modéliser des annonces certes publiques (tous les agents la reçoivent), mais aussi des annonces privées vers un ou plusieurs agents spécifiques, simuler des communications entre agents souhaitant partager une connaissance, etc.

4.2.3 Implémentation

Nous allons développer et expliquer les algorithmes principaux permettant le bon fonctionnement du mécanisme de raisonnement.

Les mondes de la structure de Kripke La méthode principale ici est celle d'évaluation d'une formule avec une structure de Kripke. Elle permet de savoir si le monde considéré dans la structure satisfait la formule donnée. Pour cela, on appelle la méthode d'évaluation de la formule donnée avec les paramètres. On obtient ainsi un booléen indiquant si le monde satisfait la formule dans la structure.

Le modèle ou structure de Kripke Tout d'abord, nous allons considérer l'implémentation d'une annonce publique d'une formule sur un modèle de Kripke avec l'algorithme 2.

Algorithme 2 Algorithme pour annoncer une formule publiquement (*publicAnnouncement*)

Entrée: un modèle de Kripke $\mathcal{M}$ et une formule φ
  $worlds\_to\_remove \leftarrow \{\}$
  pour $w \in getWorlds(\mathcal{M})$ **faire**
    si $\neg\, w.satisfied(\varphi, \mathcal{M})$ **alors**
      $worlds\_to\_remove \leftarrow worlds\_to\_remove \cup \{w\}$
    fin si
  fin pour
  pour $w \in worlds\_to\_remove$ **faire**
    Supprimer les liens partants de ou arrivants vers w
    Supprimer le monde w de $\mathcal{M}$
  fin pour

Cet algorithme permet à terme, de supprimer tous les mondes qui se satisfont pas la formule φ dans le modèle de Kripke $\mathcal{M}$. En effet, quand on réalise une annonce publique, on souhaite que la structure résultante soit restreinte en fonction de φ. Il est donc nécessaire de supprimer les mondes qui ne respectent pas l'annonce. À noter que la fonction *getWorlds* dépend des choix des structures de données utilisées (nous l'avons modéliser ici comme une fonction mais dans l'implémentation, c'est une méthode associée à la classe `KripkeStructure`).

Ensuite, une autre méthode très utile pour la suite, notamment lors de l'évaluation de formules dans la logique modale, est présentée avec l'algorithme 3.

Algorithme 3 Algorithme de la méthode *getWorldsFromOtherWorldAndAgent*

Entrée: un modèle de Kripke $\mathcal{M}$, un monde $w \in getWorlds(\mathcal{M})$ et un agent A
Sortie: un ensemble de mondes accessibles à partir d'un monde w pour l'agent A
  $map\_agent\_worlds \leftarrow \mathcal{M}.get(w)$
  $worlds \leftarrow map\_agent\_worlds.get(A)$
  retourne $worlds$

Cette méthode est très dépendante de la structure de données choisie. Nous avons décidé de la montrer comme nous avons choisi de l'implémenter, avec une liste de successeurs. Pour chaque monde, on a un dictionnaire où les clés sont des agents et les valeurs un ensemble de mondes. Cela décrit un ensemble de mondes accessibles à partir d'un monde pour chaque agent. Il est intéressant d'utiliser cette structure car elle est peu gourmande en mémoire (sauf si le nombre d'arêtes commence à devenir important par rapport au nombre de sommets) mais aussi et surtout car elle reprend le concept visuel

de la structure. Effectivement, si on observe la figure 1, on voit clairement qu'il y a des mondes qui sont reliés à un ensemble d'autres mondes par au moins un agent.

D'autres méthodes dans cette structure ont été implémentées comme l'initialisation d'un graphe vide afin de le remplir facilement par la suite, ou d'autres permettant à l'utilisateur de remplir partiellement un graphe et ajouter automatiquement les arêtes réflexives et symétriques.

Opérateurs modaux Nous n'allons pas détailler ici l'implémentation des opérateurs $\Box_A \varphi$ et $\Diamond_A \varphi$ car ils sont équivalents, à des connecteurs logiques près, à la modalité $K_A \varphi$. Pour $\Box \varphi$, il est totalement équivalent à la connaissance relative à un agent A et pour $\Diamond_A \varphi$, son équivalent logique est $\neg K \neg \varphi$ ($\equiv \hat{K}_a \varphi$) qui exprime l'hésitation sur la validité d'une formule φ ou sa "non impossibilité" et donc bien la possibilité comme énoncé au début de la section 4.2.

Algorithme 4 Algorithme d'évaluation sur une connaissance d'un agent (K_a)

Entrée: un modèle pointé $\mathcal{M}, w$ et une formule de type $K_A \varphi$
  $result \leftarrow VRAI$
  pour $u \in getWorldsFromOtherWorldAndAgent(\mathcal{M}, A)$ **faire**
    $result \leftarrow result \wedge u.satisfied(\varphi, \mathcal{M})$
  fin pour
  retourne $result$

L'algorithme 4 présente la méthode d'évaluation d'une formule de connaissances relatives à un agent A. Cet algorithme est une implémentation stricte de la règle $\mathcal{M}, w \models K_a \varphi$ ssi $\forall u \in R_a(w),\ \mathcal{M}, u \models \varphi$.

Analoguement, on peut réaliser l'algorithme d'évaluation de formules de connaissances partagées par un groupe d'agents J (montré à l'algorithme 5).

Algorithme 5 Algorithme d'évaluation sur une connaissance partagée (EK_J)

Entrée: un modèle pointé $\mathcal{M}, w$ et une formule de type $EK_J \varphi$
  $result \leftarrow VRAI$
  pour $A \in J$ **faire**
    pour $u \in getWorldsFromOtherWorldAndAgent(\mathcal{M}, A)$ **faire**
      $result \leftarrow result \wedge u.satisfied(\varphi, \mathcal{M})$
    fin pour
  fin pour
  retourne $result$

Cet algorithme est une implémentation stricte de la règle $\mathcal{M}, w \models EK_J \varphi$ ssi $\forall u \in \left(\bigcup_{a \in J} R_a(w)\right),\ \mathcal{M}, u \models \varphi$.

Enfin, nous pouvons réaliser l'algorithme d'évaluation de formules de connaissances communes d'un groupe d'agents J (proposé à l'algorithme 6).

Algorithme 6 Algorithme d'évaluation sur connaissance commune (CK_J)

Entrée: un modèle pointé $\mathcal{M}, w$, une formule de type $CK_J\,\varphi$

```
file ← createQueue()
markedWorlds ← {}
file.enqueue(w)
markedWorlds ← markedWorlds ∪ {w}
tant que file n'est pas vide faire
    world ← file.dequeue()
    si ¬ world.satisfied(EK_J φ, M) alors
        retourne FAUX
    fin si
    pour A ∈ J faire
        pour u ∈ getWorldsFromOtherWorldAndAgent(M, A) faire
            si u ∉ markedWorlds alors
                file.enqueue(w)
                markedWorlds ← markedWorlds ∪ {w}
            fin si
        fin pour
    fin pour
fin tant que
retourne VRAI
```

La règle $\mathcal{M}, w \models CK_J\,\varphi$ ssi $\forall u \in \left(\bigcup_{a\in J} R_a(w)\right)^*,\ \mathcal{M}, u \models \varphi$ est implémentée avec l'aide de l'algorithme BFS (Breadth-first search ou parcours en largeur en français). Réaliser un parcours en largeur permet d'accéder aux sommets d'un graphe pour chaque niveau de profondeur. En l'adaptant pour ne pas afficher les mondes mais évaluer une formule, nous obtenons l'algorithme 6. Nous avons utilisé cet algorithme car il correspond exactement aux besoins pour évaluer une formule de type $CK_J\,\varphi$. En effet, nous devons vérifier qu'à chaque monde accessible par un agent a dans un groupe J, la formule $EK_J\varphi$ est vraie. On peut simplifier cela en "tous les agents savent φ vraie dans tous les mondes accessibles à partir de w dans $\mathcal{M}$ et pour toute profondeur finie dans le modèle".

4.3 Interpréteur de programmes

L'interpréteur, le coeur de ce projet, doit permettre de prendre des programmes d'agents en entrée et par des annonces publiques, faire en sorte que les agents en déduisent des états en observant les actions des autres agents dont ils connaissent le programme.

4.3.1 Le programme d'un agent

Il nous faut expliciter ce que nous entendons par "programme d'un agent".

4.3.1.1 Modélisation

On peut modéliser cela par une suite de conditions, de branchements. En effet, un agent possédant un programme agit selon lui en fonction des conditions liées à l'environnement dans lequel il évolue. Par exemple, un programme pourrait être : si a = 1 alors retourner Vrai sinon Faux. Ces conditions doivent être ordonnées car on peut vouloir réaliser une action précise avant une autre si celle-ci n'est pas valide. Dans notre cas, un agent doit être capable de réfléchir en fonction de formules logiques. Ainsi, il est nécessaire que les conditions soient des formules qui devront être vraies afin d'exécuter les instructions situées dans le branchement pris. Pour les branchements, on peut considérer que si la formule du branchement considéré est valide, on effectue l'action associée sur l'environnement (une action qui peut avoir une répercussion sur l'environnement ou non).

4.3.1.2 Implémentation

Nous avons choisi d'implémenter un programme d'agent en dérivant l'interface `Map`. Comme dit précédemment, on associe une formule à une action en faisant attention à l'ordre des associations dans ce dictionnaire. Pour réaliser cela, on aurait pu faire hériter une `LinkedHashMap` ce qui résolvait le problème de l'ordre des conditions. Néanmoins, l'ordre est créé en fonction de l'ordre d'insertion, ce qui induit que si un programme choisi de modifier l'ordre des conditions ou en insérer de nouvelles, celles-ci seraient placées à la fin ce qui casse donc l'ordre d'exécution des évaluations des formules (et donc les conditions). Nous avons donc ré-implémenté une classe complète maintenant deux listes internes, une pour les formules à évaluer et l'autre les actions à exécuter si la formule associée est vraie.

Le point le plus important et le plus intéressant de notre implémentation est la possibilité de créer un `sinon` avec un `null` comme clé. Cela est possible dans les dictionnaires classiques mais dans notre implémentation, lors de l'ajout, l'insertion à un indice, la modification ou la suppression d'une association, le `sinon` sera toujours en dernière position (car il doit toujours être exécuté en dernier si toutes les évaluations ont échoué). De plus, nous réalisons aussi une vérification de la non nullité de l'action permettant ainsi de réduire le nombre de levées d'exceptions dans l'exécution de l'application. Pour représenter une action nulle, on peut tout simplement créé une fonction lambda qui ne prend aucun argument et ne renvoie aucun objet (n'observe rien et n'agit pas sur l'environnement).

4.3.2 Un agent

4.3.2.1 Modélisation

Un agent peut être représenté par un nom afin de le reconnaître plus facilement parmi plusieurs autres entités, mais aussi par le programme qu'il exécute (voir section 4.3.1). Un agent n'est globalement qu'une entité qui possède un programme à exécuter dans un environnement dans lequel il évolue. Il peut donc récupérer et exécuter une action, mais aussi être capable de récupérer la connaissance à partir de son programme qui a pu entraîner l'action qu'il a exécuté (algorithme d'ingénierie inversée).

4.3.2.2 Implémentation

Récupération d'une action à partir du programme d'un agent Tout d'abord, il est important de préciser que l'agent exécutant son programme le fait en fonction des connaissances qu'il possède à cet instant pour un modèle pointé donné. L'algorithme 7 présente la façon d'exécuter le programme d'un agent pour récupérer la bonne action en fonction de ses connaissances ($\emptyset$ représente un `null` en programmation).

Algorithme 7 Algorithme d'exécution du programme

Entrée: un modèle pointé $\mathcal{M}, w$ et le programme P de l'agent A
  $i \leftarrow -1$
  $condition\_validate \leftarrow FAUX$
  tant que $!condition\_validate$ **faire**
    $i \leftarrow i + 1$
    si $i >= size(P)$ **alors** ▷ si aucun sinon n'est présent dans P
      retourne $\emptyset$
    fin si
    $formula \leftarrow P.getKey(i)$
    si $formula \neq \emptyset$ **alors**
      $condition\_validate \leftarrow formula.evaluate(\mathcal{M}, w)$
    sinon ▷ gestion du sinon du programme
      $condition\_validate \leftarrow VRAI$
    fin si
  fin tant que
  retourne $P.getValue(i)$

Cet algorithme parcourt successivement toutes les clés du dictionnaire, qui sont des formules et les évaluent jusqu'à qu'il n'y est plus de conditions, qu'on soit arrivé au "sinon" ou si une évaluation renvoie `VRAI`. Nous avons donc plusieurs cas :

1. l'évaluation renvoie `VRAI` : l'exécution s'arrête et on renvoie l'action associée à la formule vraie ;
2. on arrive jusqu'au sinon : on arrête la boucle de parcourt de test des conditions et on renvoie l'action associée au sinon ;
3. si toutes les évaluations renvoient faux et qu'il n'y a pas de sinon, on retourne un `null`.

Ceci nous permet de gérer tous les cas possibles liés à l'exécution d'un programme et donc le développeur est libre de créer n'importe quel programme pour ses agents.

Récupération de connaissances d'un agent à partir de son programme Ceci implémente une partie du système de raisonnement. En effet, un agent ne peut raisonner en fonction des connaissances des autres agents uniquement s'il les possède. Mais le seul moyen de déduire ces connaissances est de les récupérer à partir des programmes des

autres agents (dont il connaît le programme) pour ainsi pouvoir continuer son mécanisme de raisonnement. L'algorithme 8 présente l'algorithme mis en oeuvre pour réaliser cette déduction ($\emptyset$ représente un `null` en programmation).

Algorithme 8 Algorithme de déduction des connaissances

Entrée: une action *action* et le programme P de l'agent A
  si $action = \emptyset$ **alors**
    retourne $\emptyset$
  fin si
  $formulas \leftarrow \{\}$
  $negativeConditions \leftarrow \{\}$
  pour i, $1 \leq i \leq size(P)$ **faire**
    $formula \leftarrow P.getKey(i)$
    $k \leftarrow P.getValue(i)$
    si $action = k$ **alors**
      $form \leftarrow \emptyset$
      si $size(negativeConditions) \neq 0$ **alors**
        $form = \bigwedge_{f \in negativeConditions} f$
      fin si
      $formulas \leftarrow formulas \cup \{formula \wedge form\}$
    fin si
    si $formula \neq \emptyset$ **alors**
      $negativeConditions \leftarrow negativeConditions \cup \{\neg formula\}$
    fin si
  fin pour
  retourne $\bigvee_{f \in formulas} f$

Pour mieux visualiser l'algorithme, nous allons expliciter les attendus sur un exemple de programme à base de connaissances (KBP) présenté à l'algorithme 9.

Imaginons que l'agent ait exécuté *action*1, il faut pouvoir récupérer les connaissances associées. Ici, on observe que *action*1 est dans deux branchements donc les deux connaissances associées sont possibles. Tout d'abord, nous avons φ_1 car c'est la première possibilité. Ensuite, nous avons $\varphi_3 \wedge \neg\varphi_1 \wedge \neg\varphi_2$ car s'il rentre dans le troisième branchement, cela signifie que φ_1 et φ_2 sont faux et que φ_3 est vrai. Ainsi, nous obtenons $\varphi_1 \vee (\varphi_3 \wedge \neg\varphi_1 \wedge \neg\varphi_2)$. On réalise une disjonction car on ne sait pas quel branchement a été emprunté. Cet exemple permet de bien voir que l'ordre de branchement a ici une importance dans l'exécution des actions et les déductions des connaissances à partir d'elles. Ici, le branchement sinon est considéré comme une condition normale. En effet, dans notre programme sinon est équivalent à un `null` mais dans les implémentations, nous avons géré le fait qu'une formule pouvait être à `null` et donc cela n'entraîne aucune erreur dans l'exécution de notre algorithme.

Algorithme 9 Exemple de KBP

si φ_1 **alors**
    Exécuter *action*1
sinon si φ_2 **alors**
    Exécuter *action*2
sinon si φ_3 **alors**
    Exécuter *action*1
sinon
    Exécuter *action*3
fin si

4.3.3 L'interpréteur

L'interpréteur a pour but de gérer les structures associées à chaque agent, d'exécuter des méthodes pour une collection d'agents pour ainsi simplifier l'utilisation.

4.3.3.1 Annonces

Plusieurs surcharges de méthodes sont disponibles afin que le développeur puisse passer différents types de structures de données. Deux surcharges associent un à un des agents et formules afin de donner des informations différentes (ou non) à chaque agents. Deux autres prennent en paramètres un ou plusieurs agents avec une formule à annoncer à chacun d'entre eux. Ces deux approches permettent ainsi de gérer différents types d'annonces :

— une affirmation que l'on annonce à tous les agents, les annonces publiques ;
— une affirmation, que l'on annonce à seulement une partie des agents : les annonces publiques partielles ;
— des affirmations différentes pour chaque agent : les annonces privées.

On peut donc, dans ce système, annoncer n'importe quelle formule à n'importe quel agent pour ainsi gérer toute situation.

4.3.3.2 Actions

Comme on a pu le voir à la section 4.3.2, les agents peuvent récupérer des actions à partir d'évaluation de connaissances et les exécuter. L'interpréteur, en associant une structure à chaque agent, permet de gérer efficacement la récupération et l'exécution des actions des agents.

Pour exécuter les actions, il est nécessaire d'expliciter l'algorithme réalisé (voir algorithme 10).

Cet algorithme exécute les actions récupérées pour les agents en fonction des arguments de celles-ci stockées dans *objects*. Elle retourne les retours d'exécution des actions. En effet, dans certains programmes une action peut ne rien retourner mais dans la majorité des cas, elle retourne un objet qui doit pouvoir être récupéré afin de le traiter dans

Algorithme 10 Algorithme d'exécution des actions dans l'interpréteur

Entrée: une association agent-actions *actions* et une association agent-objets *objects*
  $returns \leftarrow \{\}$
  pour $(agent, action) \in actions$ **faire**
    si $action = \emptyset$ **alors**
      continue
    fin si
    $obj \leftarrow agent.performs(objects.get(action))$
    $returns \leftarrow returns \cup \{(agent, obj)\}$
  fin pour
  retourne *returns*

l'environnement. Cet objet pourra possiblement entraîner une modification de l'environnement d'exécution.

4.3.3.3 La rétro-ingénierie

Cette méthode dans l'interpréteur a le même but que celui explicité dans l'algorithme 8. La particularité ici est que l'on gère pour tous les agents et non plus uniquement pour un agent. On renvoie une association agent-formule qui sera utilisée par l'algorithme montré dans le paragraphe suivant.

4.3.3.4 Le raisonnement

Ceci est le point clé de l'interpréteur. En effet, le but du projet est que des agents puissent raisonner en fonction des actions des autres agents s'ils connaissent leur programme. Ces connaissances de programmes, on va les nommer des permissions. Un agent A pourra connaître la déduction à partir des actions d'un agent B uniquement s'il a eu connaissance du programme de l'agent B et donc eu la permission d'accéder à son programme. L'algorithme 11 montre l'implémentation d'un tel mécanisme.

Algorithme 11 Algorithme de raisonnement sur connaissances

Entrée: une collection d'agents *agents*, une association agent-formules *observations* et une association agent-(collection d'agents) *permissions*
  $formulas \leftarrow \{\}$
  pour $agentA \in agents$ **faire**
    $formula \leftarrow VRAI$
    pour $agentB \in permissions.get(agentA)$ **faire**
      $formula \leftarrow formula \wedge observations.get(agentB)$
    fin pour
    $formulas \leftarrow formulas \cup \{(agentA, formula)\}$
  fin pour
  retourne $formulas$

Cet algorithme crée une formule logique qui doit satisfaire chacune des connaissances des agents dont il a eu accès au programme. On retourne une association agent-formule qui peut ensuite être passée dans la méthode d'annonce publique et ici, chaque agent possède sa propre formule et donc les structures seront possiblement différentes.

4.3.3.5 Terminaison de l'interpréteur

Nous avons implémenté des méthodes très simples afin de vérifier si l'interpréteur peut encore continuer à interpréter les programmes des agents en fonction de l'état de leurs connaissances à l'instant t, notamment grâce aux mondes encore disponibles au sein des structures associées aux agents.

Deux implémentations ont été proposées :

1. prise en argument d'un monde pointé : on vérifie qu'il n'y a plus d'autres mondes accessibles à l'agent associé à la structure à partir du monde pointé et cela pour chaque agent ;
2. aucun argument : on vérifie que la structure associée à l'agent ne possède plus qu'un seul monde et cela pour chaque agent.

Mais cela introduit des limites dans les possibilités de l'interpréteur. En effet, dans certains cas, tous les agents ne connaissent pas exactement le monde réel dans leur structure et certains le savent (dépendant de l'environnement créé et des déductions réalisées) et cela implique que les formules suivantes déduites ne pourront plus modifier les structures faisant donc entrer le programme dans une boucle infinie. Pour gérer cela nous avons deux options :

1. ajouter une limite d'itérations (qu'il faut bien choisir) lors de l'utilisation de l'interpréteur, afin de réaliser un certain nombre de fois l'interprétation des programmes des agents. Ceci implique qu'il existe un risque que le programme s'arrête avant d'avoir trouvé la solution ;
2. définir une fonction lambda ou méthode pour gérer plus subtilement l'arrêt de l'interpréteur notamment avec l'aide des règles de l'environnement et les objets retournés par l'exécution des actions des agents.

En utilisant au moins l'une des deux, si elles sont correctement créées et utilisées, permettent d'arrêter l'exécution de l'interpréteur et ainsi d'obtenir les résultats probants.

5 Conclusion

5.1 Apports du projet

La réalisation de ce projet nous a permis de continuer à développer de multiples compétences, notamment dans la gestion de projets, le développement objet et notre capacité à comprendre un problème pouvant paraître simple au premier abord mais plus ou moins complexe à la compréhension et implémentation. Cependant, ce projet nous a apporté de nouvelles connaissances, que ce soit dans le type de systèmes multi-agents mais aussi dans cette forme de logique dite modale que nous ne connaissions pas du tout auparavant.

5.2 Conclusion générale

Ce projet annuel est pour nous une réussite car quand nous avons vu au début la difficulté du projet, nous nous sommes vite demandé si nous allions arriver à faire le projet dans les temps et avec toutes les fonctionnalités requises à son bon fonctionnement. Finalement, nous y sommes arrivés et cela a été un très bon projet car nous avons pris plaisir à comprendre le mécanisme de raisonnement logique dans un système multi-agents avec les modèles de Kripke, mais aussi à implémenter cela.

Venant de parcours différents, Corentin de DOP (Décision et Optimisation) et Guillaume d'IDM (Images & Données Multimédia), cela a été fort intéressant car le projet nous a permis d'aller plus loin dans les connaissances de l'un et de l'autre. Grâce à ces deux horizons, on apportait chacun une vision qui pouvait parfois diverger mais cela a été bénéfique pour la réalisation du projet.

De plus, les rendez-vous réguliers avec M. ZANUTTINI nous ont vraiment permis d'avancer à un bon rythme, d'avoir un très bon suivi et ainsi, ne pas procrastiner le développement du projet, notamment comme d'autres groupes qui n'ont pas eu de rendez-vous aussi souvent que nous et qui ont parfois repoussé le développement. Ce suivi régulier a vraiment été important car le projet étant assez complexe, nous avions besoin de quelqu'un pour nous aider sur les notions parce que le développement a parfois été ralenti car nous n'avions pas très bien compris certaines d'entre elles.

5.3 Ouverture et axes d'amélioration

Tout d'abord, lors de l'implémentation et des premiers tests de l'interpréteur, nous avions remarqué que dans le problème des enfants sales (section 2), les enfants qui étaient propres ne le savent pas mais ceux qui sont sales savent qu'ils sont sales. Nous pensions que c'était en problème mais ce n'en était pas un. En effet, de l'extérieur, nous pensons que l'enfant propre sait qu'il l'est mais à sa place, il ne peut pas savoir en observant uniquement les actions des autres agents, il doit être capable de raisonner sur son propre modèle de fonctionnement. Cet axe d'amélioration serait donc de mettre en place une logique plus complexe comme de la logique d'ordre supérieur afin que les derniers agents ne connaissant pas leur état puissent le connaître.

Ensuite, dans l'interpréteur encore une fois, on peut éviter d'avoir une structure pour chaque agent (donc maintenir autant de structures qu'il y a d'agents). En effet, cela entraîne une plus forte utilisation de la mémoire vive que simplement en stocker une seule représentant les connaissances de tous les agents. Pour cela, il est possible de faire une intersection des différentes structures de Kripke associées aux agents afin d'obtenir une seule structure, et avec un seul et même monde, le monde réel. Cela permet donc régler le problème présenté au paragraphe précédent et à la section 4.3.3.5. Une autre méthode permettant d'arriver au même point, ce serait de n'avoir qu'une seule structure mais annoncer publiquement les déductions de chaque enfant. Ces deux solutions reviennent à n'avoir au final qu'une seule structure en mémoire à la fin d'un tour d'exécution.

Enfin, on peut trouver des axes d'optimisation, notamment lors de la création du graphe pour la structure de Kripke. On pourrait aussi optimiser le système d'évaluation des formules avec de la programmation dynamique afin d'éviter de réévaluer des formules avec les mêmes états pour ainsi être plus rapide dans l'exécution de l'interpréteur. On pourrait aussi réaliser du multi-threading pour les méthodes n'ayant pas d'interactions entre les agents. Cela permettrait d'accélérer l'exécution de l'interprétation.

Références

[1] *Problème des enfants sales (anglais).* http://sofia.nmsu.edu/~pmorandi/math111f01/MuddyChildren.html.

[2] *Logique propositionnelle.* https://www.wikiwand.com/fr/Formule_propositionnelle.

[3] *Logique épistémique.* https://www.wikiwand.com/fr/Logique_%C3%A9pist%C3%A9mique.

[4] Francois SCHWARZENTRUBER. *Francois Schwarzentruber: Logique épistémique dynamique.* https://www.youtube.com/watch?v=VO_favwnQBA.

[5] *Connaissance commune.* https://www.wikiwand.com/en/Common_knowledge_(logic).