IPSEITY : une plateforme *open source* dédiée à l'étude de problèmes de décision séquentielle dans des systèmes multiagents

Fabrice Lauri et Abderrafiaa Koukam

IRTES-SeT
Rue Thiery-Mieg, 90010 Belfort
fabrice.lauri@utbm.fr

Résumé : Cet article présente les fonctionnalités, les concepts sous-jacents et l'architecture générale de la plateforme *open source* IPSEITY, développée en C++ avec le framework *Qt* [1]. IPSEITY a été conçue pour permettre de faciliter le développement et l'étude de techniques d'Intelligence Artificielle appliquées à la résolution de problèmes de décision séquentielle dans des environnements multiagents. La version actuelle (1.2.0) de la plateforme propose un ensemble de plugins implémentant, entre autres, des environnements monoagents et multiagents, des techniques classiques d'apprentissage par renforcement comme Q-Learning, Sarsa et Fitted Q-Iteration avec *extra-trees*, des algorithmes de sélection d'action comme Epsilon-Greedy et Softmax, ainsi que des approximateurs de fonctions linéaires. Actuellement, IPSEITY permet d'exécuter notamment des algorithmes *model-free* d'Apprentissage par Renforcement qui peuvent être *off-line* ou *on-line*. Pour les algorithmes *off-line*, les échantillons d'apprentissage sont collectés a priori, soit manuellement en interagissant directement avec l'environnement, soit à l'aide de contrôleurs prédéfinis pour certains environnements.
Mots-clés : IPSEITY, plateforme *open source*, décision séquentielle, apprentissage par renforcement, système multi-agent.

1 Introduction

Les systèmes multiagents (SMA) constituent un paradigme adapté pour représenter et pour simuler des phénomènes complexes émergeant de l'interaction de nombreuses entités autonomes. Dans un système multiagent, chaque agent perçoit des informations généralement partielles issues de son environnement et agit sur la base de ces perceptions. Une classe d'agents particulièrement étudiés sont les agents rationnels (Russell & Norvig, 2009). Un agent est rationnel s'il choisit, à chaque prise de décision, une action qui maximise une mesure de performance donnée, compte tenu de l'historique de ses perceptions et des connaissances qu'il a de son environnement. Cette mesure de performance définit un critère de succès sur la séquence des actions réalisées par l'agent. L'étude des agents rationnels pour la résolution de problèmes de décision séquentielle constitue une des branches de l'intelligence artificielle particulièrement féconde. Les travaux étudiant ces types d'agents peuvent avoir pour objectif d'apporter des réponses théoriques ou pratiques à des questions aussi diverses que :

- Quelle mesure de performance utiliser pour que tel objectif puisse être accompli par des agents rationnels ?
- Quelle quantité de données permet de garantir une performance satisfaisante de tel algorithme de prise de décision ?
- Quel(s) algorithme(s) parmi un ensemble prédéfini d'algorithmes permet(tent) d'obtenir la meilleure performance pour accomplir une tâche donnée ?

Comme souligné par Kovacs (Kovacs & Egginton, 2011), résoudre des problèmes de décision séquentielle nécessite d'écrire des algorithmes, de conduire des expériences et d'analyser des résultats provenant de techniques appartenant à des domaines éventuellement très différents. Les types d'environnements, les architectures d'agent, les algorithmes de prises de décisions et les mesures de performance sont nombreux.

1. http://www.qt-project.org

En effet, par exemple, définir un environnement multiagent particulier nécessite de spécifier son espace de temps (discret ou continu), l'espace d'états et l'espace d'actions des agents, de préciser si les transitions entre états de l'environnement sont déterministes ou stochastiques, si les prises de décision des agents sont statiques (la mise à jour de l'environnement intervient uniquement lorsque tous les agents ont transmis leurs influences) ou au contraire dynamique (l'environnement peut être mis à jour alors qu'un agent est encore en train de "réfléchir"), si les agents sont coopératifs ou compétitifs, etc.

A la diversité des environnements multiagents s'ajoute celle des algorithmes de résolution, car les comportements individuels des agents composant un SMA peuvent être définis selon plusieurs mécanismes de prise de décision. Par exemple, un algorithme de planification peut être utilisé lorsque l'agent connaît la représentation de l'ensemble des états buts à atteindre et doit décider de la séquence d'actions à réaliser pour atteindre un de ces buts. Les actions d'un agent peuvent aussi être déterminées par une machine d'états finis, un système de classeurs, un algorithme de programmation génétique, etc. Lorsque l'environnement peut fournir une récompense à chaque action réalisée par un agent, une alternative est d'employer une technique issue de l'Apprentissage par Renforcement (AR) pour permettre à l'agent d'apprendre à réaliser la meilleure séquence d'actions maximisant la somme des récompenses reçues. La taxinomie de ces algorithmes de résolution de problèmes de décision séquentielle est complexe, en raison des hypothèses et des paramètres sur lesquels ils sont fondés. Par exemple, un algorithme d'AR peut être classé selon plusieurs critères (Sutton & Barto, 1998; Busoniu *et al.*, 2010). Il peut :

- être basé sur *Value Iteration*, *Policy Iteration* ou *Policy Search*,
- être *model-free* ou *model-based*,
- être *off-policy* ou *on-policy*,
- être *off-line* ou *on-line*,
- utiliser une technique de sélection d'action parmi celles disponibles dans l'état de l'art (comme *Epsilon-Greedy*, *Softmax*...),
- utiliser ou non un approximateur de fonction.

Enfin, à cette complexité inhérente aux environnements et aux algorithmes de résolution s'ajoute celle de l'évaluation de la performance des algorithmes, qui peut tenir compte de l'espérance des récompenses reçues, du nombre de décisions prises pour atteindre l'objectif, de la complexité de l'algorithme en terme de temps de calcul ou par rapport à la quantité des données nécessaires pour atteindre une performance suffisante.

A notre connaissance, IPSEITY est actuellement la seule plateforme multiagent permettant à des chercheurs ou à des étudiants, entre autres :

- d'utiliser un des algorithmes "clés en main" disponibles pour résoudre des problèmes de décision séquentielle,
- d'interagir facilement avec un des environnements,
- d'utiliser des techniques *offline* à partir des interactions générées par un humain ou par un contrôleur,
- d'implémenter de nouveaux algorithmes et environnements,
- d'étudier l'influence de certains paramètres (comme les fréquences de décisions des agents, la fréquence de mise à jour de l'environnement, ou les paramètres des algorithmes de décision) sur la performance des agents.

En effet, RL-Glue [2] (Tanner & White, 2009), CLSquare [3], PIQLE [4](Comité & Delepoulle, 2005), RL Toolbox [5] (Neumann, 2005), JRLF [6], LibPGRL [7] ne supportent que des techniques d'AR mono-agent. Les boîtes à outils *Matlab* MDP [8] et MARL [9] supportent l'apprentissage par renforcement multiagent sous *Matlab*. Cependant, parce qu'ils dépendent de l'environnement *Matlab*, les environnements et algorithmes implémentés dans ces boîtes à outils ne sont pas aussi facilement extensibles que ceux d'IPSEITY, qui bénéficie entre autres des avantages liés à la programmation orientée-objet, comme l'encapsulation, l'héritage et le polymorphisme. De plus, ces boîtes à outils Matlab ne permettent pas d'interagir dans l'environnement.

2. http://glue.rl-community.org/wiki/Main_Page
3. http://ml.informatik.uni-freiburg.de/research/clsquare
4. http://piqle.sourceforge.net
5. http://www.igi.tu-graz.ac.at/gerhard/ril-toolbox/general/overview.html
6. http://mykel.kochenderfer.com/jrlf
7. https://code.google.com/p/libpgrl
8. http://www.inra.fr/mia/T/MDPtoolbox/
9. http://busoniu.net/repository.php

IPSEITY fournit un ensemble prédéfini d'environnements qui peuvent être simulés et éventuellement visualisés, un ensemble prédéfini d'algorithmes de prises de décision et un ensemble prédéfini d'outils pour faciliter leur étude. Nous présentons par la suite les principales fonctionnalités d'IPSEITY, ses concepts sous-jacents ainsi que son architecture générale. Le présent article constitue une version étendue de (Lauri *et al.*, 2013b) et (Lauri & Koukam, 2014).

2 Aperçu

IPSEITY est une plateforme *open source* spécialement dédiée pour faciliter le développement et la validation expérimentale de techniques d'Intelligence Artificielle appliquées à la résolution de problèmes de décision séquentielle pour des agents coopératifs ou compétitifs. Les agents peuvent évoluer dans un environnement multiagent statique ou dynamique, selon la terminologie de Russell et Norvig (Russell & Norvig, 2009). L'espace d'états, l'espace d'actions et l'espace temporel de l'environnement peuvent chacun être discret ou continu.

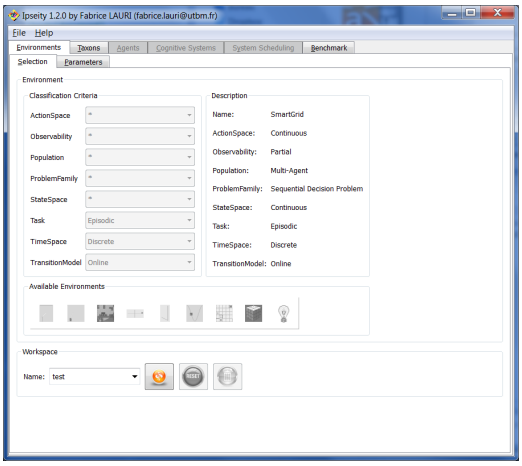
2.1 Concepts principaux sous-jacents

Dans IPSEITY, une population d'agents éventuellement dynamique interagit au sein d'un *environnement*. Un ensemble de groupes d'agents, appelés *taxons* dans IPSEITY, peuvent être définis pour un environnement donné. Les agents appartenant au même taxon partagent le même mécanisme de prise de décision et sont donc susceptibles de se comporter de manière similaire. Le comportement d'un agent est exhibé selon son *système cognitif*. Un système cognitif implémente le processus de décision qui permet à un agent de sélectionner l'action à réaliser à partir de ses perceptions. Une classe donnée de système cognitif, basée sur une combinaison de techniques d'AR par exemple, peut être associée à un taxon. Ceci signifie que tous les agents appartenant au même taxon utiliseront le même algorithme de prise de décision.

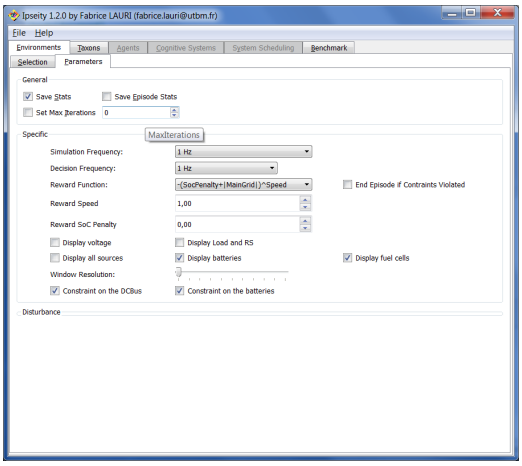
Différents *paramétrages* et *scénarios* peuvent être définis dans un *benchmark* pour permettre à un utilisateur d'étudier la qualité des décisions prises individuellement ou collectivement par les agents sous certaines conditions initiales. Au cours d'une simulation, l'ordre des scénarios prédéfinis peut être modifié par un *superviseur*, qui peut être l'utilisateur lui-même ou un agent. Les données générées pendant une simulation par l'environnement, les agents ou les systèmes cognitifs, et en particulier les statistiques représentant les mesures de performance des algorithmes exécutés, sont sauvegardées au sein d'un répertoire de travail, ou *workspace*. Ces données sont sauvegardées dans un format textuel et peuvent donc être ensuite exploitées pour générer des graphiques sous *Matlab* ou *Gnuplot*.

2.2 Fonctionnalités principales

La version actuelle d'IPSEITY permet à l'utilisateur de charger et de paramétrer un des environnements prédéfinis, de créer un workspace, d'effacer les données associées à un workspace ou de détruire un workspace (figure 1). Dans cette figure, l'environnement SmartGrid a été chargé à partir de l'onglet *Selection*, et ses paramètres sont modifiables dans l'onglet *Parameters*. Les paramètres disposés dans le groupe *General* sont communs à tous les environnements. L'implémentation d'un nouvel environnement ne nécessite pas de redéfinir ces paramètres. Les environnements actuellement prédéfinis sont les environnements classiques mono-agents étudiés dans le domaine du contrôle, tels que *Acrobot*, *Cartpole*, *DoubleIntegrator*, *InvertedPendulum*, *MountainCar*, ainsi que des environnements de jeu de réflexion tels que *RasendeRoboter* et *Rubik's Cube*. Les environnements multiagents actuellement disponibles sont l'environnement *SmartGrid* (Lauri *et al.*, 2013a) et un jeu vidéo de type grid-world appelé *Delirium2* (largement inspiré du jeu *Boulder Dash*). L'utilisateur peut également associer un système cognitif à un taxon, détruire cette association, et paramétrer les systèmes cognitifs chargés (figure 2). Dans l'exemple illustré par la figure, un système cognitif basé sur des techniques issues de l'Apprentissage par Renforcement a été chargé. Les paramètres spécifiques à ce type de techniques sont accessibles dans le groupe *Controls*. Avant de réaliser une simulation, l'utilisateur peut paramétrer le *System Scheduling* (figure 3a), charger un des fichiers de scénarios prédéfinis ou charger un module de supervision de benchmark (figure 3b). Dans le cas où ce module est basé sur l'interaction d'un utilisateur (comme c'est le cas par défaut), celui-ci peut sélectionner les scénarios impliqués dans la simulation, spécifier un fichier de paramétrages, modifier la vitesse de simulation, décider de commencer la simulation, de faire une pause ou de la stopper, décider de visualiser la simulation en sélectionnant éventuellement les fenêtres de rendu, ou modifier la fréquence de rafraichissement.

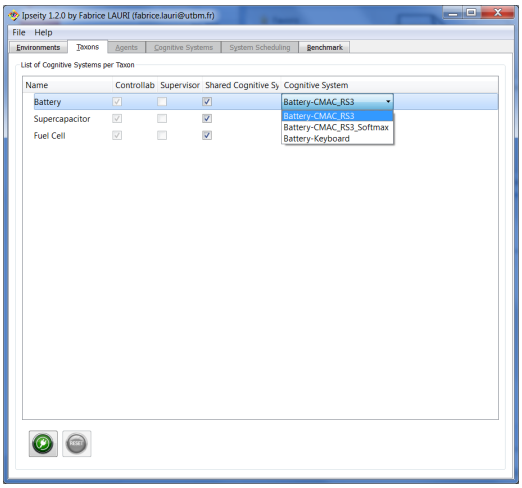


a) Sélection de l'environnement

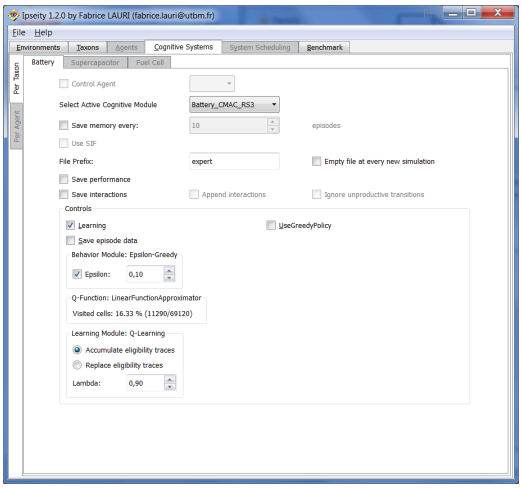


b) Paramétrisation de l'environnement

FIGURE 1 – Aperçus d'écran d'IPSEITY 1.2.0 relatifs à la gestion des environnements.

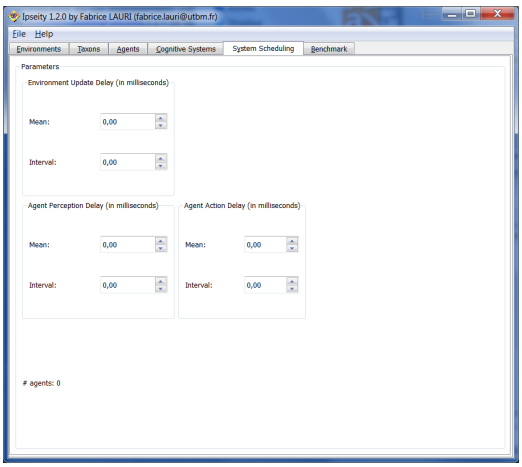


a) Association des systèmes cognitifs aux taxons

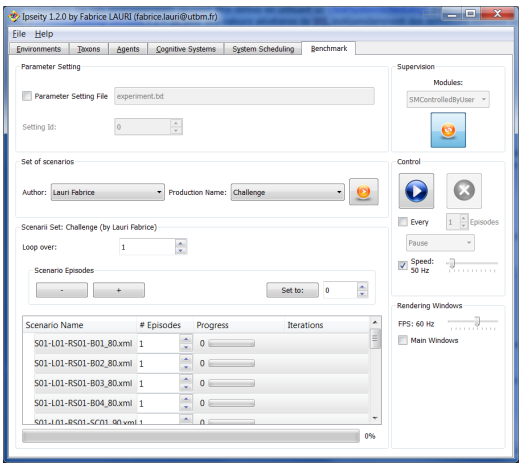


b) Paramétrisation des systèmes cognitifs

FIGURE 2 – Aperçus d'écran d'IPSEITY 1.2.0 relatifs à la gestion des systèmes cognitifs.



a) Paramétrisation du *System Scheduling*



b) Paramétrisation de la simulation

FIGURE 3 – Aperçus d'écran d'IPSEITY 1.2.0 relatifs à la gestion des simulations.

2.3 Propriétés

IPSEITY satisfait plusieurs des recomandations proposées par Kovacs *et al.* (Kovacs & Egginton, 2011). En particulier, IPSEITY possèdent les propriétés suivantes :

Flexibilité : IPSEITY utilise des concepts sous-jacents et des composants (c'est-à-dire des structures de données et des algorithmes) qui sont aussi flexibles que possibles. Par exemple, les perceptions et les réponses des agents sont représentées par des vecteurs de flottants 64-bits, permettant aux agents de pouvoir interagir dans des environnements discrets ou continus.

Modularité : IPSEITY utilise des modules, ou *plugin*, qui implémentent les concepts sous-jacents et les composants. Par exemple, les environments, les systèmes cognitifs, le *System Scheduling*, et la sélection des scénarios de simulation sont tous définis en tant que plugins.

Facilité d'integration : Grâce à cette modularité, ces plugins (et en particulier ceux représentant des systèmes cognitifs) peuvent facilement être intégrés dans d'autres applications. Par exemple, IPSEITY peut être utilisé pour apprendre le comportement de certains agents. Une fois l'apprentissage terminé, les agents peuvent percevoir des informations provenant d'un système distant et lui envoyer leurs commandes de contrôle sur la base des comportements appris. Une telle intégration a été réalisée entre IPSEITY et JANUS (Gaud *et al.*, 2009) : des systèmes cognitifs ayant appris leurs comportements sous IPSEITY peuvent ainsi communiquer par sockets TCP-IP avec des agents évoluant dans un simulateur de microgrille développé sous JANUS.

Extensibilité : IPSEITY peut facilement être étendu par des plugins spécialisés pour le domaine d'application ciblé. Des extensions personnalisées incluent de nouveaux environnements, de nouveaux algorithmes de prise de décision ou de nouveaux modules de visualisation pour des environnements existants. L'implémentation de ces extensions est facilitée par le fait qu'IPSEITY bénéficie des avantages liés à la programmation orientée-objet (encapsulation, héritage et polymorphisme). En outre, des classes ont été développées en particulier pour faciliter l'interfaçage entre *C++* et des langages spécialisés comme *Prolog* ou *Python*.

Facilité d'analyse des données : IPSEITY enregistre dans un format textuel les données générées lors des simulations, telles que les états rencontrés et les actions réalisées par chaque agent à chaque instant, ou les mesures de performance des algorithmes. Ces données sont sauvegardées au sein d'un workspace, permettant ainsi à l'utilisateur d'exploiter ensuite ces données avec un logiciel spécialisé.

Interface graphique : IPSEITY fournit une interface intuitive (figures 1, 2 et 3) avec des icônes et widgets permettant facilement de paramétrer les variables impliquées dans la simulation d'un SMA, notamment celles de l'environnement et des systèmes cognitifs.

3 Framework théorique

Dans cette section, nous allons décrire plus en détails les concepts clés sur lesquels sont fondés IPSEITY. Les notations suivantes sont utilisées ci-dessous : $\mathbf{2} = \{0, 1\}$ est l'ensemble des booléens et $[M] = \{1, 2, \ldots, M\}$.

3.1 Agent

Tout agent i perçoit son environnement grâce à des *signaux de stimuli* $\mathcal{S}_i$ et il agit sur son environnement en utilisant des *signaux d'actions* $\mathcal{A}_i$. Les décisions d'un agent sont déterminées par son *système cognitif*.

3.2 Taxon

En biologie, un taxon (ou groupe taxonomique) peut être défini comme étant un groupe de populations d'organismes qui sont fortement liés entre eux phylogénétiquement et qui ont plusieurs caractéristiques en commun. Dans IPSEITY, un taxon k est défini par un tuple $\tau_k = (C_k, \mathcal{P}_k, \mathcal{U}_k)$. A chaque (nouvel) agent est associé un unique taxon. L'agent appartient à ce taxon pendant toute sa "vie virtuelle". Les agents appartenant au même taxon k ont été implicitement regroupés ensemble parce qu'ils possèdent les mêmes caractéristiques, et en particulier la même structure cognitive. Ils peuvent recevoir les mêmes types d'informations (percepts) à partir de *l'ensemble de perceptions* $\mathcal{P}_k$. Ils peuvent réaliser les mêmes types d'actions

prises dans *l'ensemble de réponses* $\mathcal{U}_k$. Leur processus de décision peut être implémenté par la même classe de système cognitif C_k. Tout agent i appartenant au taxon k utilise des percepts x tels que $x \in \mathcal{S}_i = \mathcal{P}_k$ et il sélectionne ses actions u tels que $u \in \mathcal{A}_i = \mathcal{U}_k$. Deux exemples de taxons qui n'ont pas les mêmes systèmes cognitifs et espaces de stimuli et de réponses sont *Homo Sapiens* (être humains) et *Antaresia Maculosa* (pythons réticulés).

3.3 Système multiagent

Un système multiagent M est considéré comme étant défini par le tuple $M = \left(\mathcal{S}, \mathcal{T}, \overline{f}, g, h, \Lambda, \Omega\right)$ où :

- $\mathcal{S}$ est l'ensemble des états environmentaux possibles. L'état $s \in \mathcal{S}$ décrit complètement l'environnement multiagent à un instant donné. Par souci de clarté, nous supposons que $\mathcal{S}$ est fini dans le reste de l'article.
- $\mathcal{T} = \{\tau_1, \ldots, \tau_M\}$ est l'ensemble de taxons.
- $\overline{f}$ est la fonction de transition entre états.
- $g : \mathcal{S} \to \mathbf{2}^{\mathbb{N}}$ est la fonction qui associe à chaque état une population d'agents. Tout agent i est uniquement identifié par cet entier i. Par exemple, $g(s) = \{1, 7, 11, 13\}$ signifie que les 4 agents 1, 7, 11 et 13 sont présents et actifs (c'est-à-dire qu'ils peuvent prendre des décisions) dans l'état s.
- $h : \mathbb{N} \to \mathbb{N}$ est la fonction qui associe à chaque agent le taxon auquel il appartient. $h^{-1}(\{k\})$ est l'image réciproque de h qui détermine l'ensemble des agents appartenant au taxon k.
- $\Lambda = (\lambda_k)_{k \in [M]}$ est une famille de fonctions $\lambda_k : \mathcal{S} \times h^{-1}(\{k\}) \to \mathcal{P}_k$. $\lambda_k(s, i)$ extrait un stimulus pour l'agent i à partir d'un état s, en considérant que $i \in h^{-1}(\{k\})$, c'est-à-dire que l'agent i appartient au taxon k.
- Ω décrit le *System Scheduling* (expliqué en détail dans la section 3.5).

Soit $\mathcal{A} = \mathcal{U}_1^* \times \cdots \times \mathcal{U}_M^*$ l'ensemble des actions jointes des agents dont la population peut être dynamique. La fonction $\overline{f}$ de transition entre états peut être déterministe ou stochastique et elle peut prendre en compte un temps discret ou un temps continu. Elle peut être généralisée à la fonction $\overline{f} : \mathbb{R} \times \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0; 1]$, où la probabilité d'atteindre l'état $s_{t'}$ au temps t' après avoir réalisé l'action $u_t \in \mathcal{A}$ dans $s_t \in \mathcal{S}$ au temps $t = t' - \Delta_{t'-t}$ est donné par $\overline{f}(\Delta_{t'-t}, s_t, u_t, s_{t'})$. Pour tout Δ_t, s_t et u_t, $\sum_{s' \in \mathcal{S}} \overline{f}(\Delta_t, s_t, a_t, s') = 1$ (puisque $\mathcal{S}$ est supposé fini). Cette fonction de transition $\overline{f}$ correspond au cas stochastique en temps continu. Pour le cas déterministe, $\overline{f}$ peut être remplacé par $f : \mathbb{R} \times \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbf{2}$, c'est-à-dire que pour tout Δ_t, s_t et u_t, $\overline{f}(\Delta_t, s_t, a_t, s')$ est égal à 1 si s' est l'état suivant lorsque l'action a_t est réalisé dans s_t et 0 sinon. Pour prendre en compte le temps discret, l'état s' est défini selon $\overline{f}$ sans prendre en compte le premier paramètre $\Delta_{t'-t}$ de la fonction, ou de manière équivalente, en supposant que $\Delta_{t'-t}$ est constant entre deux instants t' et $t < t'$.

Soit $\mathcal{G}_t = g(s_t)$ la population d'agents dans l'état s_t au temps t. La fonction de transition peut générer un état $s_{t'}$ au temps $t' > t$ en tenant compte :

- du temps écoulé $\Delta_{t'-t} \in \mathbb{R}$ entre l'instant t de la dernière mise à jour et l'instant t',
- l'état $s_t \in \mathcal{S}$, et
- l'action jointe $a_t \in \mathcal{A}_t$ produite par les agents dans la population $\mathcal{G}_t$, où $\mathcal{A}_t = \times(\mathcal{A}_i)_{i \in \mathcal{G}_t}$.

Soit $T = (t_1, t_2, \ldots) \in \mathbb{R}^{\mathbb{N}}$ une séquence valide d'instants, telle que $t_j > t_i$ pour tout $j > i$. T est composé des instants de mise à jour de l'environnement par le biais de la fonction de transition. La sequence $(s_t)_{t \in T}$ représente une trajectoire de l'environment, où s_{t_j} est obtenu à partir de s_{t_i} pour tout $j = i + 1$.

3.4 Système cognitif

Soit $T_i^{dec} = (t_{i,1}^{dec}, t_{i,2}^{dec}, \ldots)$ une séquence valide d'instants qui est composée des instants où l'agent i prend une décision dans un scénario donné. Le comportement de l'agent i appartenant au taxon $k = h(i)$ est exhibé par la séquence d'actions $(u_{i,t})_{t \in T_i^{dec}}$ telle que tout $u_{i,t} \in \mathcal{A}_i$ est généré par son *système cognitif* c_i, instance de la classe C_k des systèmes cognitifs.

Soit $T_i^{per} = (t_{i,1}^{per}, t_{i,2}^{per}, \ldots)$ une séquence valide composée des instants où l'agent i perçoit son environnement dans un scénario donné. Le stimulus $s_{i,t} \in \mathcal{S}_i$ de l'agent i appartenant au taxon $k = h(i)$ est extrait à partir de l'état s_t à l'instant $t \in T_i^{per}$ comme suit :

$$s_{i,t} = \lambda_k(s_t, i) \tag{1}$$

Le système cognitif c_i de l'agent i est composé d'une *fonction d'intégration* $\Phi_i : \mathcal{X}_i \times \mathcal{S}_i \times \mathcal{X}_i \to [0;1]$ et d'une *fonction de décision* $\Pi_i : \mathcal{X}_i \times \mathcal{A}_i \to [0;1]$. $\Phi_i(x, s, x')$ représente la probabilité de transition entre le *contexte spatio-temporel* x et le contexte spatio-temporel x' en considérant le stimulus s. $\Pi_i(x, u)$ représente la probabilité que l'action $u \in \mathcal{U}_i$ est réalisée lorsque le contexte spatio-temporel $x \in \mathcal{X}_i$ est rencontré. Le contexte spatio-temporel d'un agent représente son état mental à un instant donné et il peut inclure ses expériences passées. Si l'agent i n'a pas mémorisé ses stimuli et actions passés, $\mathcal{X}_i = \mathcal{S}_i$ et $\Phi_i(x, s, x')$ donnent une probabilité de 1 pour $x' = s$ et 0 pour les autres stimuli. Si l'agent i est capable de mémoriser et d'exploiter un historique des N stimuli rencontrés précédemment et de ses $N-1$ actions préalablement réalisées, alors $\mathcal{X}_i = \underbrace{(\mathcal{S}_i \times \mathcal{A}_i \times \cdots \times \mathcal{S}_i)}_{N}$.

3.5 *System Scheduling*

Le *System Scheduling* Ω définit la fréquence de mise à jour de l'environnement, la fréquence de perception de chaque agent et la fréquence de prise de décision de chaque agent.

Il peut être défini par le tuple $\Omega = (\upsilon, \mu, \nu, \Delta_t)$ où :

- $\upsilon : \mathbb{R} \to \mathbf{2}$. $\upsilon(t)$ indique si l'environnement est mis à jour à l'instant t.
- $\mu, \nu : \mathbb{R} \times \mathcal{G} \to \mathbf{2}^{\mathcal{G}}$, with $\mathcal{G} \subset \mathbf{2}^{\mathbb{N}}$. $\mu(t, G)$ (respectivement $\nu(t, G)$) spécifie le sous-ensemble d'agents qui perçoivent (respectivement prennent une décision) à l'instant $t \in \mathbb{R}$ à partir de la population d'agents $G \in \mathcal{G}$.
- Δ_t est la fréquence de simulation spécifiée par l'utilisateur.

Selon cette définition, le *System Scheduling* peut être considéré comme étant un *thread* qui est exécuté à chaque fois que Δ_t secondes se sont écoulées. Lorsqu'il est exécuté à l'instant t, ce *thread* :

- met à jour l'environnement en appliquant la fonction de transition $\overline{f}$ si $\upsilon(t) = 1$;
- permet aux agents spécifiés par $\mu(t, G_t)$ de percevoir leur environnement, où $G_t = g(s_t)$ est la population d'agents à l'instant t extraite de s_t ;
- permet aux agents spécifiés par $\nu(t, G_t)$ de prendre des décisions au sein de leur environnement.

Le *System Scheduling* Ω et la fonction de transition $\overline{f}$ doivent être définis de concert selon le type d'environnement désiré, afin de répondre de manière satisfaisante aux questions suivantes :

1. Est-ce que le temps s'écoule de manière continue ou de manière discrète ?
2. Est-ce que les états suivants sont générés de manière déterministe ou de manière stochastique ?
3. Est-ce que l'environnement peut se mettre à jour alors que des agents sont encore plongés dans leur processus de décision ? En d'autres termes, est-ce que l'environnement est statique ou dynamique (du point de vue de l'agent) ?

Quelques classes d'environnements sont présentées ci-dessous selon le *System Scheduling* et la fonction de transition définis.

3.5.1 Environnements statiques à temps discret

Un environnement statique à temps discret peut être défini par le *System Scheduling* $(\upsilon_0, \mu_0, \nu_0)$, où :

- $\upsilon_0(t) = \begin{cases} 1 & \text{si} \quad \exists k \in \mathbb{N}^*,\ t \geq k\ \Delta_t \\ 0 & \text{sinon} \end{cases}$
- $\mu_0(t, G) = \begin{cases} G & \text{si} \quad \upsilon_0(t) = 1 \\ \emptyset & \text{sinon} \end{cases}$ et
- $\nu_0(t, G) = \begin{cases} G & \text{si} \quad \upsilon_0(t) = 1 \\ \emptyset & \text{sinon} \end{cases}$

Dans un tel environnement, à chaque période de temps Δ_t, l'environnement est mis à jour en tenant compte des décisions de tous les agents. En outre, la fonction $\overline{f}$ est supposée être définie sans prendre en considération la période de temps $\Delta_{t'-t}$ écoulée entre les instants t et t'.

3.5.2 Environnements dynamiques à temps continu

Dans un environnement dynamique à temps continu, pour une période de temps Δ_t, l'environnement est mis à jour en tenant compte des décisions de quelques agents. Dans un tel environnement, peut être employé

le *System Scheduling* $(\upsilon_1, \mu_1, \nu_1)$ tel que :

$$\upsilon_1(t) \quad = \quad \begin{cases} 1 & \text{si} \quad \exists k \in \mathbb{N}^*,\ t \geq k\ \Delta_t \\ 0 & \text{sinon} \end{cases} \tag{2}$$

et tel que certains agents n'agissent pas immédiatement après avoir perçu leur environnement, c'est-à-dire que la contrainte suivante est respectée :

$$\begin{aligned} &\exists t_0, t_1 \in \mathbb{R}, \exists i \in G_0 = g(s_{t_0}) \cap G_1 = g(s_{t_1}), \\ &(\upsilon_1(t_1) = 1) \wedge (t_1 > t_0) \wedge (i \in \mu_1(t_0, G_0)) \wedge \\ &\qquad (i \notin \nu_1(t_0, G_0)) \wedge (i \in \nu_1(t_1, G_1)) \end{aligned} \tag{3}$$

Dans un environnement à temps continu, la fonction $\overline{f}$ est définie en tenant compte de la période de temps $\Delta_{t'-t}$. Par exemple, ce paramètre pourrait être utilisé pour calculer l'accéleration puis la vitesse appliquée au centre de gravité d'un agent mobile.

3.5.3 Environnements "chaotiques"

Dans un environnement qualifié ici de "chaotique", l'environnement est mis à jour à des intervalles de temps irréguliers. De tels environnements peuvent être définis en utilisant un *System Scheduling* avec $\upsilon(t) = 1$ pour des valeurs aléatoires de t, indépendamment des définitions données aux fonctions μ et ν.

3.6 Benchmark

Les comportements des agents peuvent être étudiés sur la base des résultats obtenus en simulation à l'aide d'un *benchmark*. Pour étudier de la manière la plus complète possible la qualité des décisions prises par les agents lorsqu'ils sont confrontés à différentes situations, plusieurs *scénarios* peuvent être préalablement définis puis utilisés au sein d'un benchmark. Pour étudier l'influence des paramètres d'un environnement donné, du *System Scheduling* ou des algorithmes de résolution sur la qualité des décisions prises par les agent, plusieurs *paramétrisations* (ou *parameter settings*) peuvent être successivement testées dans un benchmark.

3.7 Scénario

Un scénario σ peut être défini par le tuple $\sigma = (l, \tilde{\mathcal{S}})$, où $l \in \mathbb{N}$ est l'identifiant unique du scénario et $\tilde{\mathcal{S}} \subset \mathcal{S}$ représente l'ensemble des états initiaux de l'environnement. Lorsqu'un scénario σ est sélectionné au cours d'une simulation, un état initial $s_0 \in \tilde{\mathcal{S}}$ est utilisé pour initialiser l'environnement.

3.8 Paramétrisation

Tous les plugins dans IPSEITY peuvent être paramétrés. Un paramètre est caractérisé par son nom (visible dans IPSEITY en laissant la souris sur le widget associé), son type et sa valeur. Par exemple, un paramètre commun à tous les environnements est le nombre maximum d'itérations simulés (paramètre *MaxIterations* de type entier, illustré dans la figure 1b). Actuellement, la portée des paramètres d'un plugin est globale, c'est-à-dire que l'application IPSEITY ainsi que tous les autres plugins peuvent consulter la valeur de ces paramètres ou la modifier. Lors d'une simulation, si l'on désire tester l'influence de certains paramètres sur les performances des agents, un fichier de paramétrisation peut être utilisé. Ce fichier décrit, pour un ensemble de paramètres, les valeurs qu'ils prendront successivement pendant la simulation. Par exemple, le fichier suivant :

```
MaxIterations=1000
NbrExpertSamples=(100:100:1000)
InteractionFilePrefix=rcal_$NbrExpertSamples$
Lambda=(0.0 0.1 0.5 1.0)
```

indique que le paramètre *MaxIterations* est initialisé à la valeur 1000 tout au long de la simulation, *NbrExpertSamples* prendra successivement les valeurs $100, 200, 300, \ldots, 900, 1000$, le paramètre *InteractionFilePrefix* sera initialisé avec la chaîne de caractère "rcal_" suivie de la valeur courante de *NbrExpertSamples*

et le paramètre *Lambda* prendra successivement une des 4 valeurs spécifiées entre parenthèses, pour chaque valeur prise par les autres paramètres, et en particulier par *NbrExpertSamples*. Ainsi, dans cet exemple, chaque scénario sera simulé 40 fois, car il existe 40 combinaisons de valeurs qui peuvent être prises conjointement par *NbrExpertSamples* et *Lambda* (10 valeurs possibles pour *NbrExpertSamples* multipliées par 4 valeurs possibles pour *Lambda*).

3.9 Workspace

Les interactions des agents, les informations relatives à l'environnement (trajectoires des états), les mesures de performance des agents et les valeurs des paramètres des plugins sont sauvegardés au sein d'un *workspace*.

4 Architecture logicielle

Tous les concepts précédents sont implémentés au sein de classes dans IPSEITY. Nous allons maintenant présenter sous forme de diagrammes de dépendances entre classes l'architecture de l'application IPSEITY, d'un environnement multiagent puis d'un système cognitif tels que définis précédemment. Pour des raisons de clarté de présentation, nous avons choisi de ne pas inclure de diagrames de classes. Le lecteur désirant connaître les attributs et les méthodes membres des différentes classes disponibles dans IPSEITY est invité à télécharger l'archive des sources disponible sur le site dédié. Il pourra alors consulter la documentation *DOxygen* également présente dans l'archive.

4.1 Architecture générale de la plateforme

IPSEITY permet de simuler des environnements multiagents héritant de la classe *AbstractEnvironment*, comme illustré à la figure 4.

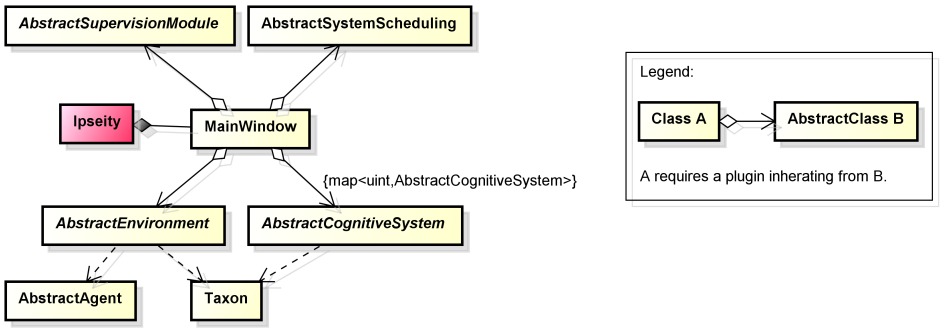


FIGURE 4 – Architecture générale d'IPSEITY (seules les classes principales sont représentées).

Plusieurs agents, chacun héritant de la classe *AbstractAgent*, interagissent dans un environnement. Les décisions d'un agent sont déterminées par son système cognitif, héritant de la classe *AbstractCognitiveSystem*. Plusieurs taxons peuvent être définis dans un environnement. La structure *map* permet d'associer à l'identifiant d'un taxon le système cognitif qui sera utilisé par tous les agents appartenant au taxon. C'est le module de supervision, héritant de la classe *AbstractSupervisionModule*, qui est chargé de définir le benchmark employé lors d'une simulation, c'est-à-dire l'ensemble des scénarios et l'ensemble des paramétrages effectivement utilisés. Le *System Scheduling*, héritant de la classe *AbstractSystemScheduling*, permet de définir les fréquences de mise à jour de l'environnement, ainsi que les fréquences de perception et de décision des agents. Ces fréquences peuvent être paramétrées par l'utilisateur dans l'application IPSEITY. La fenêtre principale de l'application IPSEITY (implémentée dans la classe *MainWindow*) permet de visualiser en temps-réel l'environnement multiagent et éventuellement les paramètres de certains systèmes cognitifs. Elle assure également l'appel des fonctions spécifiques pour mettre à jour l'environnement, pour transmettre les signaux de perception aux systèmes cognitifs associés aux agents et pour transmettre à l'environnement les signaux d'actions générés par les systèmes cognitifs des agents.

4.2 Architecture générale d'un système multiagent

Comme illustré à la figure 5, un environnement multiagent chargé par l'application IPSEITY est un plugin *MutiAgentEnvironment* qui hérite (indirectement) de la classe *AbstractEnvironment*. *MutiAgentEnvironment*

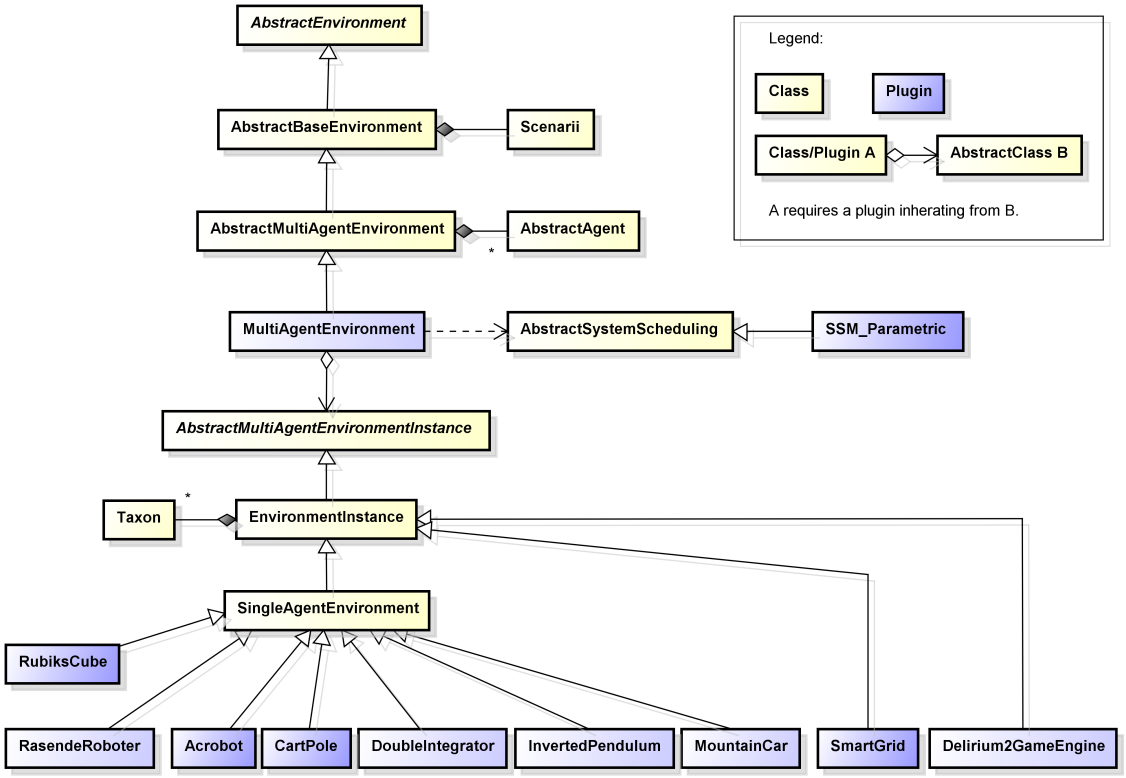


FIGURE 5 – Architecture des environnements uniagents et multiagents dans IPSEITY

hérite de la classe *AbstractMultiAgentEnvironment* qui fournit les structures de données et les méthodes pour manipuler des agents. La classe *AbstractMultiAgentEnvironment* hérite de la classe *AbstractBaseEnvironment* qui est chargée entre autres de charger et de gérer des scénarios lors de l'exécution d'une simulation. Le plugin *MutiAgentEnvironment* implémente des fonctions dont la définition est similaire dans tous les environnements, comme par exemple la gestion des taxons, l'ajout d'influences d'agents ou la mise à jour de l'environnement par l'intégration des influences des agents. *MutiAgentEnvironment* utilise le *System Scheduling* chargée préalablement par l'application IPSEITY. Ce plugin requiert le chargement d'une instance d'environnement multiagent, qui doit hériter de la classe *AbstractMutiAgentEnvironmentInstance*. Cette classe fournit les méthodes qui doivent être implémentées par un environnement particulier pour entre autres se mettre à jour à partir des influences reçues. Tous les environnements disponibles actuellement dans IPSEITY héritent indirectement de cette classe. Les environnements uniagents (respectivement multiagents) héritent de la classe *SingleAgentEnvironment* (respectivement *EnvironmentInstance*), qui fournit des structures de données (entre autres celles des taxons) et implémentent certaines des méthodes requises par *AbstractMutiAgentEnvironmentInstance* pour faciliter la définition de tels environnements.

4.3 Architecture générale d'un système cognitif

L'application IPSEITY utilise des systèmes cognitifs qui héritent de la classe *AbstractCognitiveSystem* (voir figure 6). Chaque système cognitif dans IPSEITY peut éventuellement être composé de plusieurs modules héritant de la classe *AbstractCognitiveModule*. Lorsqu'un agent doit prendre une décision, chacun des modules de son système cognitif est exécuté. Le module spécifié comme actif sélectionne l'action (de classe *Response*) que l'agent réalisera effectivement dans l'environnement. Les modules oisifs peuvent utiliser l'action choisie par le module actif pour éventuellement mettre à jour certaines de leurs structures

internes. Cette fonctionnalité est particulièrement utile lorsqu'un utilisateur interagit directement dans un environnement et que ses actions sont utilisées par un module d'apprentissage *on-line*.

Un exemple de module de système cognitif est le plugin *RLCognitiveModule*, basé sur une technique *on-line* d'apprentissage par renforcement. Ce module est actuellement composé de trois sous-modules : un *module comportemental*, qui sélectionne des actions à partir de perceptions, une *mémoire (de Q-valeurs)*, qui stocke les valeurs associées aux couples états-actions, et un *module d'apprentissage*, qui met à jour le contenu de la mémoire à partir des informations de perception obtenues après la réalisation d'une action dans l'environnement. Actuellement, *Epsilon-Greedy* et *Softmax* sont les modules (plugins) prédéfinis

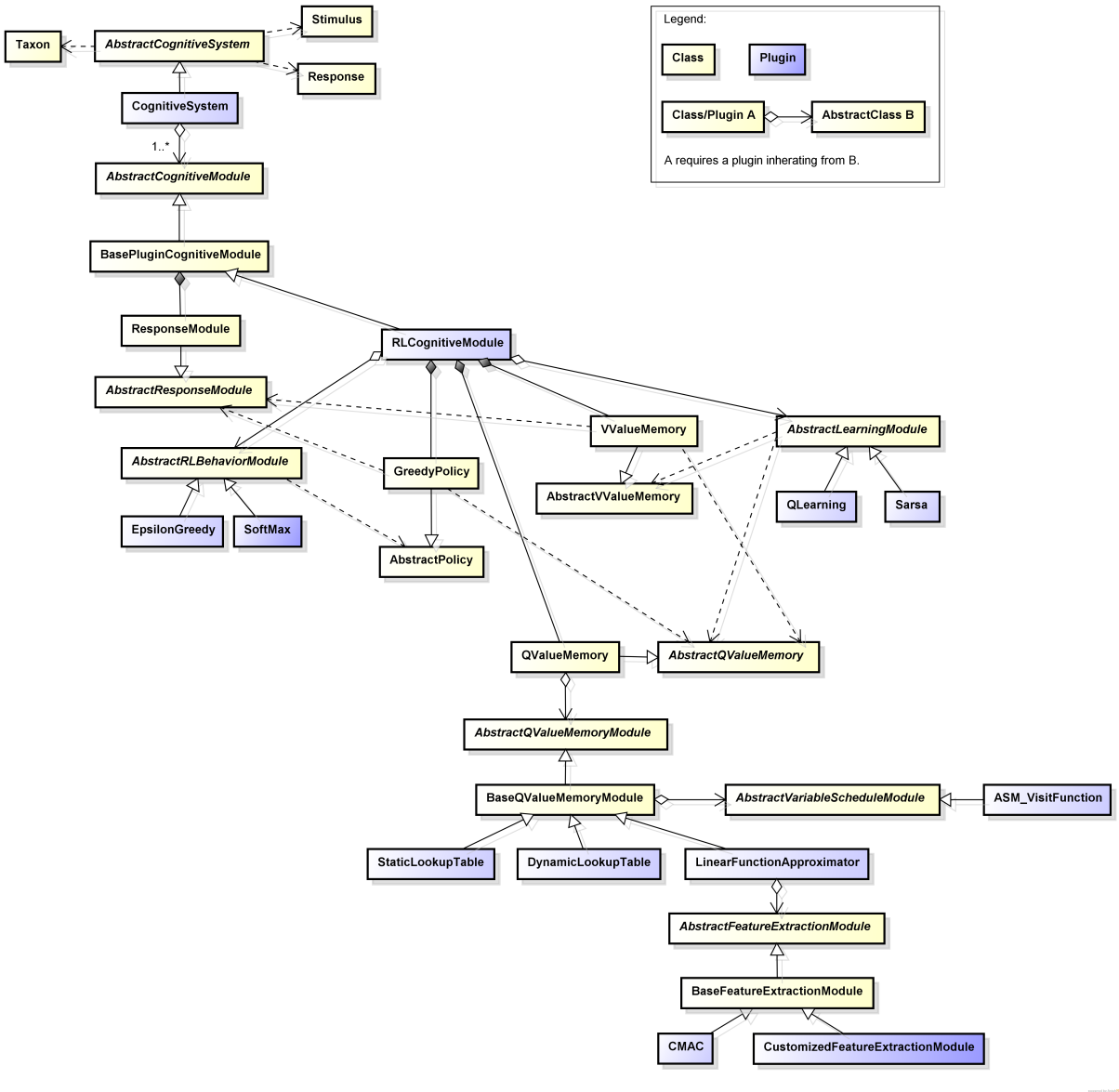


FIGURE 6 – Architecture d'un système cognitif "*Reinforcement Learning*"

qui peuvent être utilisés en tant que modules comportementaux, *Q-Learning* et *Sarsa* peuvent être utilisés comme modules d'apprentissage. La mémoire des Q-valeurs peut être instanciée par un plugin implémentant soit une *lookup table* statique (lorsque la taille de l'espace états-actions est connue), soit une *lookup table* dynamique (lorsque la taille de l'espace états-actions n'est pas connue mais néanmoins finie), ou un approximateur de fonction linéaire utilisant un *module d'extraction de features* comme CMAC (Albus, 1975) par exemple.

5 Exemples d'environnements multiagents

Nous présentons ci-après les deux environnements multiagents disponibles dans la plateforme IPSEITY.

5.1 L'environnement *SmartGrid*

La plateforme IPSEITY a été utilisé avec succès comme plateforme de simulation pour valider expérimentalement des techniques d'apprentissage par renforcement multiagent appliquées à la gestion de l'énergie dans une microgrille (Lauri *et al.*, 2013a). Dans ces travaux, la microgrille est considérée comme étant un environnement multiagent statique à temps discret, avec des espaces d'états et d'actions continus. Une microgrille est composé de plusieurs sources (batteries, supercondensateurs, panneaux photovoltaïques, éoliennes, réseau principal) et éventuellement de plusieurs charges. Les batteries et les supercondensateurs peuvent être contrôlés afin d'équilibrer la production et la demande tout en minimisant l'apport en énergie du réseau principal. Étant contrôlables a priori de manières différentes car ne possédant pas la même dynamique de fonctionnement, ils sont représentés par deux taxons (voir figure 2a). La figure 7 présente la fenêtre de rendu de cet environnement lors du contrôle autonome d'une batterie à l'aide du système cognitif basé sur de l'apprentissage par renforcement et illustré à la figure 2. Les scénarios utilisés dans cette

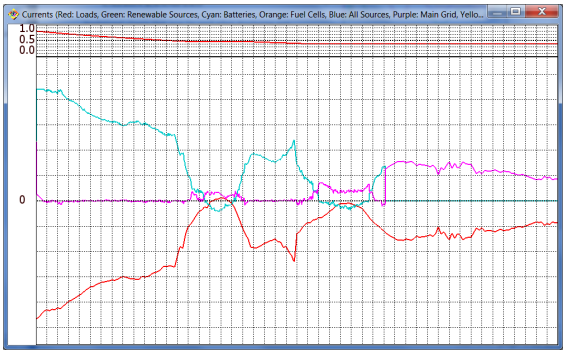


FIGURE 7 – Aperçu de la fenêtre de rendu temps-réel de l'environnement SmartGrid composé d'une batterie, d'une charge et du réseau principal. En haut, la courbe rouge indique la variation de l'état de charge de la batterie. En bas, les courbes rouge, violette et cyan indiquent respectivement l'intensité demandée par la charge, l'intensité délivrée par le réseau principale et celle délivrée par la batterie au cours du temps. La batterie est ici contrôlée par une technique d'apprentissage par renforcement.

étude ont été définis en faisant varier le nombre et le type de sources contrôlables et de charges, ainsi que les profils de production en énergie des sources renouvelables et les profils de consommation des charges. Pour cette étude, les mêmes scénarios furent utilisés pour la phase d'apprentissage et pour la phase d'évaluation. Plusieurs techniques basées sur de l'apprentissage par renforcement multiagent furent comparées, notamment en étudiant l'influence du partage des Q-valeurs parmi les agents appartenant à un même taxon sur la rapidité d'apprentissage. L'utilisation d'IPSEITY a permis notamment de faciliter la mise en œuvre des simulations, qui implique une gestion paramétrée d'un scheduling alternatif sur les agents, et la génération des résultats associés obtenue de manière automatique à partir d'un benchmark composé de plusieurs scénarios.

5.2 L'environnement *Delirium2*

Delirium2 est un jeu vidéo de type *grid-world* inspiré du jeu *Boulder Dash* crée par Peter Liepa et Chris Gray en 1983 sur Atari800. Dans ce jeu, le joueur personifie un mineur qui évolue en creusant des tunnels dans des sous-terrains. Un sous-terrain est représenté par une grille de cellules. Chaque cellule peut être vide ou contenir le mineur, de la terre à creuser, un diamant, une pierre, un mur, un monstre ou la sortie. L'objectif du mineur est de collecter un certain nombre de diamants tout en évitant les monstres. Une fois le nombre de diamants requis, le mineur doit se diriger vers la sortie (alors accessible) pour pouvoir résoudre un autre sous-terrain. Parce que le mineur crée un espace vide à chaque fois qu'il se déplace, les diamants et les pierres peuvent tomber en cascade. Lorsqu'un diamant ou qu'une pierre tombe sur le mineur ou sur un monstre, celui-ci explose. L'explosion peut se répandre sur plusieurs cases. Le mineur perd

lorsqu'il explose. Certains sous-terrains peuvent ne pas contenir immédiatement des diamants. Dans ce cas, le mineur doit éliminer certains monstres qui se transforment en diamants lorsqu'ils explosent. La figure 8 présente un aperçu d'écran de la fenêtre de rendu temps-réel de cet environnement lors de la simulation d'un sous-terrain.

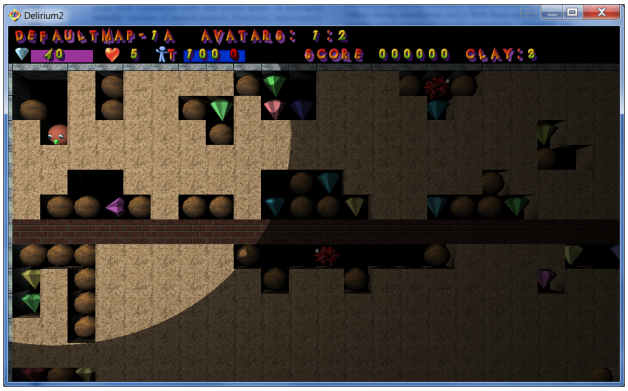


FIGURE 8 – Exemple d'un sous-terrain dans l'environnement Delirium2. 40 diamants doivent être collectés par le mineur. La sortie (non visible ici) est localisée en bas à droite.

Le problème auquel doit faire face le mineur peut être considéré comme étant un problème de décision séquentielle déterministe avec un espace discret d'états et un espace fini d'actions. La tâche du mineur peut être représentée par un MDP si l'on considère qu'il a une connaissance complète de son environnement, ou par un POMDP si ses connaissances sont partielles. En raison d'un ensemble conséquent de scénarios (plus d'une centaine) de complexités diverses, de la possibilité donnée aux utilisateurs d'interagir directement dans cet environnement, et de l'existence d'un contrôleur prédéfini (mais perfectible), cet environnement est particulièrement intéressant pour tester et comparer des algorithmes d'apprentissage par renforcement et de planification, en utilisant éventuellement un ensemble de données générées au cours de ces interactions. Une étude comparant les performances, avantages et limites d'un certain nombre d'algorithmes appartenant à des familles très diverses (Q-Learning avec ou sans approximateur de fonction, LSPI, Fitted Q-Iteration (Busoniu *et al.*, 2010, p. 63), RCAL (Piot *et al.*, 2014), LRTA*) est actuellement en cours de réalisation [10]

6 Conclusion

Un aperçu des fonctionnalités, des concepts sous-jacents et de l'architecture d'IPSEITY a été présenté dans cet article. IPSEITY est modulaire et largement extensible. Cette plateforme peut être téléchargée gratuitement sur le site `www.ipseity-project.com` sous licence *open source* GNU *GPLv3*. Des démonstrations montrant les fonctionnalités principales d'IPSEITY ainsi que des exemples d'utilisation sont également disponibles sur ce site.

Les développements en cours dans IPSEITY incluent l'implémentation de techniques *off-line* d'apprentissage par renforcement telles que *LSPI*, la redéfinition de la classe mère *AbstractEnvironnement* pour permettre l'utilisation de techniques *model-based*, l'implémentation de techniques de planification temps-réel tels que *LRTA**, ainsi que des facilités de développement sur GPU. Une fonctionnalité importante prévue à moyen terme consiste à proposer une interface Web permettant d'accéder aux résultats d'expériences en cours, en vue de faciliter la collaboration au sein de la communauté. Évidemment, toute personne désirant participer et contribuer à ce projet est cordialement encouragée à se faire connaître et à contacter le premier auteur.

10. Des résultats préliminaires avec RCAL et Fitted Q-Iteration avec extra-trees (Geurts *et al.*, 2006) sont actuellement disponibles et pourront éventuellement être présentés et discutés.

Références

ALBUS J. (1975). A new approach to manipulator control : The cerebellar articulation model controller (cmac). *Journal of Dynamic Systems, Measurement and Control*, **97**(3).

BUSONIU L., BABUSKA R., DE SCHUTTER B. & ERNST D. (2010). *Reinforcement Learning and Dynamic Programming Using Function Approximators*. CRC Press.

COMITÉ F. & DELEPOULLE S. (2005). PIQLE : a platform for implementation of Q-learning experiments. In *NIPS workshop : reinforcement learning benchmarks and bake-offs II*.

GAUD N., GALLAND S., HILAIRE V. & KOUKAM A. (2009). An organisational platform for holonic and multiagent systems. In *Programming Multi-Agent Systems*, volume 5442 of *Lecture Notes in Computer Science*.

GEURTS P., ERNST D. & WEHENKEL L. (2006). Extremely randomized trees. In *Machine Learning*, volume 36, p. 3–42.

KOVACS T. & EGGINTON R. (2011). On the analysis and design of software for reinforcement learning, with a survey of existing systems. *Machine Learning*, **84**, 7–49.

LAURI F., BASSO G., ZHU J., ROCHE R., HILAIRE V. & KOUKAM A. (2013a). Managing Power Flows in Microgrids using Multi-Agent Reinforcement Learning. In *Agent Technologies in Energy Systems (ATES)*.

LAURI F., GAUD N., GALLAND S. & HILAIRE V. (2013b). Ipseity – A Laboratory for Synthesizing and Validating Artificial Cognitive Systems in Multi-Agent Systems. In *European Conference on Machine Learning (ECML)*.

LAURI F. & KOUKAM A. (2014). Ipseity : an open-source platform for synthesizing and validating artificial cognitive systems in MAS. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*.

NEUMANN G. (2005). *The Reinforcement Learning Toolbox, Reinforcement Learning for Optimal Control Tasks*. PhD thesis, Institut for Grundlagen der Informationsverarbeitung (IGI).

PIOT B., GEIST M. & PIETQUIN O. (2014). Boosted and Reward-regularized Classification for Apprenticeship Learning. In *Autonomous Agents and Multi-Agent Systems (AAMAS)*.

RUSSELL S. & NORVIG P. (2009). *Artificial Intelligence : A Modern Approach (Third edition)*. Prentice Hall.

SUTTON R. & BARTO A. (1998). *Reinforcement Learning : An Introduction*. Cambridge, MA.

TANNER B. & WHITE A. (2009). RL-Glue : language-independent software for reinforcement learning experiments. *Journal of Machine Learning*, **10**, 2133–2136.