UNIVERSITÉ DE MONTRÉAL

SYSTÈME D'AIDE À LA DÉCISION POUR LE RÉSEAU DE DISTRIBUTION

MOHAMED GAHA

DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION

DU DIPLÔME DE PHILOSOPHIAE DOCTOR

(GÉNIE INFORMATIQUE)

NOVEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

SYSTÈME D'AIDE À LA DÉCISION POUR LE RÉSEAU DE DISTRIBUTION

présentée par : GAHA Mohamed
en vue de l'obtention du diplôme de : Philosophiae Doctor
a été dûment accepté par le jury d'examen constitué de :

M. PIERRE Samuel, Ph.D., président
M. GAGNON Michel, Ph.D., membre et directeur de recherche
M. SIROIS Frédéric, Ph.D., membre et codirecteur de recherche
M. ANJOS Miguel F., Ph.D., membre
M. BARKAOUI Kamel, Ph.D., membre

À ma grand mère, femme d'exception
je te dédie ce travail...

REMERCIEMENTS

Pour commencer, je tiens à remercier toutes les personnes qui m'ont accompagnée de près ou de loin tout au long de ma thèse.

Je remercie Mr. Michel Gagnon pour son support inconditionnel, la qualité de son encadrement, sa disponibilité et son amitié.

Un immense merci à Mr. Frédéric Sirois pour son suivi détaillé et extrêmement riche. Il m'a permis de grandement améliorer mon manuscrit et mes connaissances électriques.

Je remercie également les membres du jury pour leurs recommandations et pour avoir accepté d'évaluer ce présent travail.

Je présente toute ma gratitude à ma famille qui a supporté les bons et moins bons moments durant cette thèse. À savoir, mes sœurs Khadija (i.e. Katy) et Saida pour leurs soutiens indéfectibles, ma mère Sabiha pour ses multiples encouragements et son optimisme, ma marraine Moufida pour son sens de l'humour et à mon père Chiha pour son soutien et ses encouragements oh combien importants. Patron, je réalise ici ton rêve de Laval 1991 ! Ezzedine, grand-père chéri, tu aurais grandement apprécié ma réussite.

J'ai une pensée particulière à mes amis à savoir, Taieb & Hana, Anis & Emna, Joe, Sami, Mohsen & saloua et leur fille Linda, Errah & Doha, MDD & Rim, Wael, Faten, Phill & Miriam, Rita, Ayed... et la liste est encore longue. Une pensée particulière à mes amis Mzabi et Mehdi pour leurs encouragements lors de notre voyage aux états-unis.

Pour finir, je souhaite remercier du plus profond de mon cœur ma princesse Myriam Marzouki. Elle a su me supporter durant les moments critiques et m'a soutenu quand j'étais à bout de force. Malgré la distance, saches que tu as été à mes cotés durant les moments où j'en avais le plus besoin. Ma chérie, ma femme, ma Myriam... je t'aime. Mohamed, Jalila, Feriel & Selim et Nawel & Faouzi merci d'avoir fait de ma Myriam ce qu'elle est aujourd'hui.

RÉSUMÉ

De nos jours, de nouvelles technologies issues du domaine de l'information et de la communication sont introduites progressivement dans les réseaux de distribution électrique. Ces technologies nécessitent des études poussées et des simulations précises afin d'en évaluer les forces et les faiblesses. Toutefois, la simulation des réseaux électriques demeure une tâche complexe qui nécessite de tenir compte de plusieurs facteurs : électriques, mécaniques, économiques, naturels, matériels et humains.

Pour pallier à la complexité inhérente à la simulation électrique, il est possible de recourir aux systèmes multiagents (SMA). Ils présentent de nombreux avantages. Ils offrent une grande flexibilité en permettant à des agents autonomes de collaborer pour atteindre des objectifs complexes. Le SMA, par opposition au système de simulation monolithique, présente l'avantage d'être une architecture souple et évolutive capable de traiter des opérations complexes.

Toutefois, le développement et la manipulation de ces systèmes sont des tâches réservées à des experts en informatique et en SMA. Or, dans le cadre du projet LEOPAR, mené à l'Institut de recherche d'Hydro Québec, nous avons comme principal objectif de développer un SMA accessible à des non-experts en informatique. Le but est de permettre aux décideurs et aux ingénieurs électriques de modifier et de faire évoluer le simulateur de la manière la plus aisée possible.

Pour ce faire, nous avons développée une architecture à mi-chemin entre les architectures de Tableau Noir et les SMA. Nous avons utilisé une zone distribuée de partage de données pour permettre la communication des agents. Le partage et l'échange d'informations se fait par la modification des données distribuées. Ce mécanisme réduit la complexité des agents et leur mode de communication.

De plus, nous avons spécifié un langage d'actions de haut niveau qui permet de décrire de manière déclarative les actions, leurs effets, leurs conditions et leurs relations. Ce langage d'actions est automatiquement traduit en logique non monotone (Answer Set Programming) afin de permettre la coordination des agents du simulateur. La traduction que nous proposons du langage d'actions surpasse largement les autres langages d'actions en termes de rapidité d'exécution lors de la planification. La combinaison de notre langage d'actions et de la logique

non monotone a permis le développement d'un système performant, qui offre la possibilité à des novices de rajouter, modifier ou supprimer des agents du simulateur.

Le simulateur multiagents que nous avons développé fonctionne adéquatement et permet, entre autre, de réaliser des simulations de type Monte-Carlo pour l'étude de la fiabilité des réseaux. Notre simulateur permet de quantifier, à l'aide des indices de performances, l'impact et l'apport de nouvelles technologies. Il est en mesure de reproduire avec une grande fidélité des phénomènes électriques, mécaniques et humains, tels que la surcharge électrique des câbles, le changeur de prise des transformateurs, les équipes humaines d'intervention, le temps de restauration variable et la reconfiguration du réseau.

Notre simulateur a été testé sur de véritables réseaux de distribution d'Hydro-Québec et a démontré sa capacité à traiter de grandes quantités de données. En comparaison à d'autres simulateurs électriques multiagents standards, notre système s'est avéré être tout aussi performant mais beaucoup plus facile à développer et à faire évoluer. Lors des simulations électriques, nous avons été en mesure de réaliser des études de fiabilité qui ont permis de déterminer les facteurs les plus importants influant les performances du réseau.

ABSTRACT

Nowadays, the information system technologies are increasingly used in power distribution systems to improve network reliability and performance. The impact of these structural changes is important and requires in-depth studies and investigations. A better understanding of the effect of these technologies is required to optimize the network. However, the simulation of power network is a complex task, where several technical issues need to be considered such as : electrical, mechanical, economical, natural and human aspects.

The idea is to develop a multi-agent system (MAS) that can process complex simulations. Such a system is extensible and modular and it is composed by numerous simple agents that can collaborate and interact in order to achieve complex objectives. Multi-agent systems are capable of reaching goals that are difficult to achieve by monolithic systems or individual agents, which can be complex and hard to maintain and extend.

Nevertheless, the development and the maintenance of a MAS is a complex task that has to be performed by experts on computer science and multi-agent systems. In the framework of the project LEOPAR, carried out by the *Institute de Recherche d'Hydro-Québec*, we have as a main objective to develop an accessible and comprehensive MAS. The project's aim was to allow managers to modify the behavior and the objectives of the simulator without the assistance of an expert.

To this end, we developed a simulator based on Blackboard and MAS. Our system relies on a common pool of data to share information between agents. This type of mechanism reduces the communication complexity and makes the development of agents easier.

In addition, we defined a new action language that allows to incrementally describe the agent's actions, effects, conditions and relations. Our action language is automatically translated into a non-monotonic logic (Answer Set Programming) in order to process the agent's actions. The translated answer set program has shown to be effective in providing action plans. The action language combined to answer set programming allowed us to develop a powerful and accessible simulator, enabling novice to add, change, and remove agents' behavior.

Our simulator works properly and allows, among other things, processing power network

assessments using a Monte-Carlo approach. It analyses the impact of introducing new types of technologies, by comparing performance indicators of the network. Moreover, it is able to simulate with accuracy a wide variety of phenomena as wire overloading, protection mechanism activation, tap changer changes, human intervening team patrols, restoration process and network reconfiguration.

It has been tested on realistic distribution network of Hydro-Quebec and it performed well in assessing networks. Our simulator is performing similarly to a *classical* multi-agents system, but with the benefit of being accessible and easy to use.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ANNEXES

LISTE DES SIGLES ET ABRÉVIATIONS

ACL	Agent Communication Language
ASP	Answer Set Programming
ATN	Architecture Tableau Noir
BDI	Belief Desire Intention
CAIDI	The Customer Average Interruption Duration Index
CIM	Common Information Model
FIPA	The Foundation for Intelligent Physical Agents
HLA	High Level Architecture
JADE	Java Agent Development Framework
KS	Knowledge Source
OWL	Web Ontology Language
LEOPAR	Laboratoire d'évaluation Opérationnelle des Processus et des Architecture de Réseau
UML	Unified Modeling Language
IREQ	Institut de Recherche d'Hydro-Québec
SAIDI	The System Average Interruption Duration Index
SAIFI	The System Average Interruption Frequency Index
SMA	Système Multiagents
XML	Extensible Markup Langugae

CHAPITRE 1

INTRODUCTION

1.1 Mise en contexte

L'institut de recherche d'Hydro-Québec (IREQ) œuvre dans le domaine de l'innovation technologique et consacre son savoir-faire à améliorer la performance opérationnelle d'Hydro-Québec, développer ses pôles de croissance et mieux servir ses clients. L'IREQ réunit des ingénieurs, des chercheurs, des gestionnaires et des universitaires dans le but de développer des technologies nouvelles pour une meilleure gestion du réseau électrique.

Récemment, une vision intelligente du réseau électrique est en voie de se concrétiser. Elle englobe l'utilisation de technologies nouvelles afin d'optimiser la production et l'acheminement de l'énergie au consommateur. En effet, l'avènement des technologies de télécommunication et des systèmes d'information contribuent à introduire progressivement de la haute technologie dans les réseaux électriques. C'est ainsi qu'est né le Smart-Grid. Il est une volonté de moderniser les réseaux électriques. Parmi les principaux objectifs du Smart-Grid, nous citons les avenues suivantes :

Automatisation du réseau : il s'agit d'apporter des améliorations technologiques dans les réseaux électriques afin de réduire la durée et la fréquence des interruptions (c.-à-d. de non-alimentation). Cette opération est possible par l'introduction d'équipements de sectionnements télécommandables à distance ou par une redondance appropriée du réseau électrique de transport et de distribution.

Diminution des pics de consommation : il s'agit de réduire la consommation durant certaines périodes de surconsommation. Des pointes de consommation se produisent lors de certaines conditions climatiques et conduisent à une surexploitation du réseau. Cet état rend le réseau plus sensible aux pannes et peut entrainer des coupures de courant de grande ampleur à l'image de la panne de 2010 aux États-Unis.

Développement des architecture maillée : il s'agit de transformer l'architecture conventionnelle du réseau, qui est généralement radiale, en une architecture maillée et interconnectée. Cette nouvelle approche est due à deux facteurs : la démocratisation de nouvelles sources d'énergie domestique (éoliennes domestiques, panneaux solaires domestiques, pile domestique) et la reconfiguration du réseau à distance.

Utilisation des compteurs intelligents : l'utilisation de technologies de pointe telles que

les compteurs intelligents permet de suivre en temps réel la consommation électrique. Ces compteurs permettent une meilleure compréhension du comportement du consommateur et ainsi une meilleure prédiction des demandes énergétiques. De plus, les compteurs intelligents ouvrent la voie à la facturation variable, par tranche horaire, afin d'inciter le consommateur à faire varier ses habitudes en fonction des tarifs en vigueurs.

Production distribuée : ce type de production réfère à de petites centrales de production d'énergie (entre 3 et 1000 kV) capables de fournir un apport en énergie à certaines sections du réseau. Généralement, ces sources d'énergie dépendent de facteurs climatiques tels que le soleil ou le vent et ne sont pas en mesure de fournir de l'énergie à la demande. Ce type de production nécessite un contrôle à distance afin d'établir un équilibre entre les quantités énergétiques produites et la consommation effective. Ces petites centrales jouent un rôle complémentaire en offrant une plus grande réactivité par rapport aux grandes centrales de production.

Face à ces défis, Hydro-Québec veut se doter des outils pour étudier, analyser et simuler l'impact de ces technologies sur le réseau. Les responsables d'Hydro-Québec souhaitent disposer d'outils informatiques, préférablement probabilistes, qui simulent le plus fidèlement possible les réseaux électriques. C'est ainsi qu'est né le projet LEOPAR (Laboratoire d'Évaluation Opérationnelle des Processus et des Architectures de Réseau) qui vise à doter l'IREQ de l'expertise, des moyens techniques et du savoir-faire requis pour analyser et évaluer les scénarios d'évolution du réseau de distribution.

Le projet LEOPAR fait partie de la division Innovation Stratégique, et vise le développement des connaissances et des compétences nouvelles afin de répondre à des besoins complexes et mal définis. Il réunit des experts provenant de différents domaines dans le but de développer des outils/systèmes informatiques capables d'aider les gestionnaires dans leur prise de décision stratégique.

1.2 Problèmatique

Le projet LEOPAR cherche à offrir aux gestionnaires un outil de décision pour répondre à des projections futures qui s'inscrivent dans le cadre du Smart-Grid (voir Figure 1.1). Parmi ces projets, nous citons :

La *conduite* qui vise à introduire des modules de contrôle capables de détecter et d'isoler les défauts de service ou coupures de courant d'un réseau de distribution. La télésurveillance permettra de reconfigurer à distance le réseau électrique et ainsi délimiter et isoler certaines

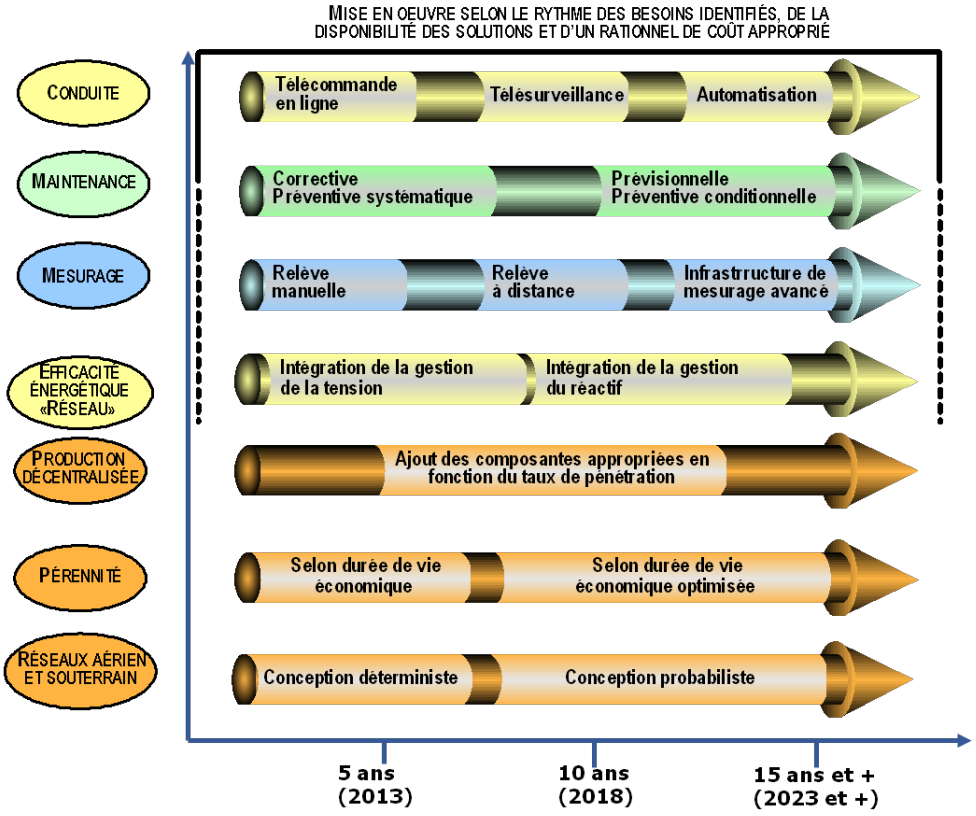


Figure 1.1 Vision à long terme d'Hydro-Québec (Simard, 2008)

sections défaillante. À long terme, il n'est pas exclu que les reconfigurations se fassent de manière automatique par l'ouverture et la fermeture des sectionneurs à distance via un système informatique.

La *maintenance* vise à permettre à Hydro-Québec de disposer des données à jour qui informent sur l'état exact du réseau. Ces données permettront de planifier plus efficacement des actions de maintenance. Cette approche nécessite une connaissance poussée des composants, de leur pérennité et de leur historique.

Le *mesurage et relève des compteurs électriques* permettra de collecter à distance la consommation électrique des entreprises et des particuliers. La vision d'Hydro-Québec est que dans un avenir proche il serait possible de déterminer le niveau de consommation client en temps réel, permettant une meilleure anticipation de la demande en énergie. Cette approche vise la mise en place de moyens de production réactifs capable de répondre efficacement à la demande énergétique.

La *production* décentralisée réfère à toutes les sources d'énergie réparties sur le réseau de distribution. Généralement ce type de production est de faible ou moyenne capacité. Nous citons, par exemple, les piles, les plaques photovoltaïques et les éoliennes qui connaissent un

intérêt grandissant de la part des particuliers. La production décentralisée concerne aussi les minis-centrales électriques distribuées sur le réseau. Elles sont capables de répondre rapidement aux besoins énergétiques. Elles compensent le manque de réactivité des grandes centrales qui nécessitent un temps de latence avant de pouvoir augmenter la production énergétique.

La *pérennité* du réseau vise à étudier la durée de vie d'un composant selon son historique d'utilisation, son taux de défaillance et son coût. Hydro-Québec cherche à déterminer la meilleure stratégie à adopter face à des composants en fin de vie. Il est primordial de déterminer si le composant est à réparer, à remplacer ou à changer par un nouveau composant de nouvelles générations.

Ainsi, la vision future du réseau d'Hydro-Québec fait intervenir différents domaines d'expertise et de compétences. Le projet LEOPAR, dans lequel s'inscrit ce doctorat, se veut être un moyen de mettre en commun les différentes expertises et compétences pour permettre la prise de décision stratégique. Il se compose d'une équipe multidisciplinaire formée de spécialistes en systèmes d'information, en simulations électriques et en processus de restauration. Notre objectif premier est de développer un système informatique capable de répondre à des questions de l'unité d'affaires en innovation (voir Figure 1.2). Une analyse détaillée du projet LEOPAR et une concertation avec les membres de l'IREQ, font ressortir les problématiques suivantes :

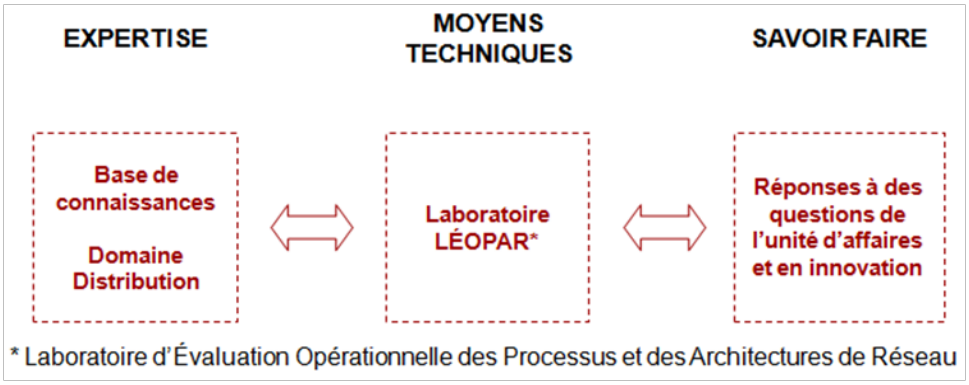


Figure 1.2 Vue globale du projet LEOPAR (Langheit, 2009)

- Les bases de données d'Hydro-Quebec sont nombreuses et l'information est parfois redondante et manque de structure. En effet, les données utilisent un format interne, des concaténations et des mots-clés spécifiques pour représenter le réseau électrique. De plus, l'absence de structure relationnelle rend l'interprétation des données complexe.
- Les besoins auxquels doit répondre le futur système sont imprécis, mal définis ou in-

connus. En effet, le système est appelé à simuler des scénarios électriques selon des objectifs changeants. Il est primordial de disposer d'un système de simulation évolutif capable de tenir compte des nouveaux besoins.

– Les utilisateurs du futur système sont des non-experts en informatique. Ils sont principalement des ingénieurs en électrique ou des gestionnaires et ne disposent pas de connaissances poussées en programmation informatique. Il en résulte que l'accessibilité de l'outil est un critère important pour permettre aux utilisateurs de faire évoluer aisément le système.

Ainsi, notre problématique est double. D'une part il s'agit de fournir un système modulaire capable de collecter des données à partir de différentes sources et de les traiter selon des besoins changeants. D'autre part, l'outil informatique doit être accessible à des non-experts.

Le critère d'accessibilité est vital dans le cadre du projet LEOPAR. Ce constat se base sur des expériences de projets passés tel que FIORD, un ancêtre du projet LEOPAR. Le projet FIORD est un logiciel développé en Prolog afin de simuler un processus d'automatisation des réseaux de distribution. Il a été fortement critiqué pour son manque de modularité et son manque d'accessibilité.

La problématique à investir est de déterminer l'architecture à d'adopter pour le futur système de simulation. Pour ce faire, deux questions centrales se posent :

1. Comment développer un système de simulation accessible, modulaire et capable de répondre à des besoins changeants ?
2. Quelle structure de données utiliser pour unifier les sources de données d'Hydro-Québec et les rendre accessibles au système de simulation ?

Ces deux questions forment le cœur de notre problématique. Nous apporterons des réponses à ces interrogations.

1.3 Objectifs

Le système à développer vise principalement le domaine des réseaux électriques de distribution. Or, développer un système de simulation peut se faire selon une approche monolithique et aboutir à un logiciel de simulation complexe, non modulaire et difficile à faire évoluer (voir Rathnam, 2004). L'alternative aux systèmes monolithiques consiste à développer *un système multiagents*(SMA) composé d'un ensemble de simulateurs autonomes coopérant pour atteindre un ou plusieurs objectifs. L'avantage d'une telle approche est la réutilisation de programmes ou simulateurs existants pour réaliser des simulations de haut niveau. Dans

le cadre du projet LEOPAR, pour développer un simulateur électrique multiagents, les trois objectifs suivants doivent être réalisés :

1. Développer un système multiagents doté d'une architecture extensible. Cette caractéristique est importante et a pour objectif de développer une architecture qui permet de rajouter, de modifier et de supprimer aisément des agents. À ce niveau, il est primordial que le système soit capable d'intégrer facilement de nouveaux agents pour répondre à des problématiques changeantes.
2. Identifier quel mécanisme utiliser pour coordonner et planifier l'action des agents. Cette étape permet de décrire selon une logique déterminée les types d'interaction que peuvent avoir les agents/simulateurs. Il est important de disposer d'un mécanisme de coordination polyvalent et facile d'utilisation pour des non-experts en informatique et systèmes multiagents.
3. Décrire selon un formalisme approprié le modèle de données des composants électriques d'Hydro-Québec. Ce modèle du domaine aura un double objectif : il sera le support informationnel au futur simulateur et il permettra d'unifier les sources de données d'Hydro-Quebec.

Ces trois étapes visent à doter l'Institut de recherche d'Hydro-Québec de l'expertise, des moyens techniques et du savoir-faire requis pour analyser et évaluer l'impact des scénarios d'évolution du réseau de distribution. Schématiquement, le futur système de simulation ressemble à terme à ce qui suit (voir Figure 1.3) :

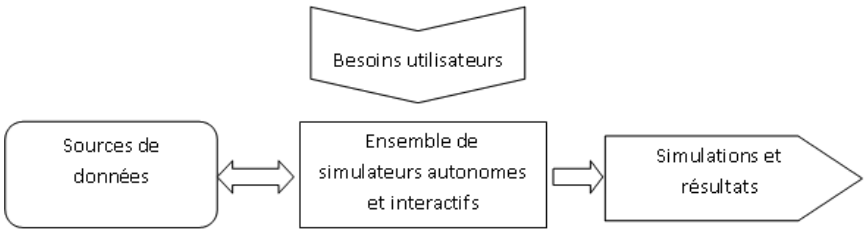


Figure 1.3 Vue globale du système de simulation

1.4 Hypothèses

En fonction des problématiques du projet LEOPAR et du besoin de l'IREQ de se doter d'un système de simulation, nous avons émis les hypothèses suivantes :

1. Nous pensons que le développement et l'utilisation des systèmes multiagents peut être problématique car ils nécessitent une grande connaissance des agents et de leur mode de communication. Nous croyons que la simplification du mode de communication des agents facilite considérablement la complexité du système.
2. Nous croyons que l'utilisation des langages multiagents est trop complexe pour des non-informaticiens. L'objectif d'accessibilité et d'extensibilité étant centraux, nous pensons qu'il est possible d'utiliser un langage de modélisation de haut niveau pour décrire les agents et leur mode de coordination.
3. Nous croyons que le recours aux ontologies et à la logique descriptive est un excellent moyen pour représenter et fusionner les connaissances d'Hydro-Québec. Nous supposons que l'ontologie sera en mesure de représenter des données hétérogènes et d'en vérifier l'intégrité.

1.5 Contributions

Le travail décrit dans cette thèse, en regard de ces hypothèses, apporte les contributions suivantes :

- Nous proposons une nouvelle architecture multi-agents accessible, extensible et performante. Notre architecture hybride est à mis chemin entre les systèmes multi-agents classiques et les architectures tableau-noir.
- Nous avons développé un nouveau langage de coordination et de collaboration des agents pour des non-spécialistes. Notre langage, formalisé sémantiquement, utilise les actions et leurs effets pour permettre la planification des agents.
- Nous avons développé un système de simulation multi-agents basé sur le web sémantique. Il permet de fusionner des données hétérogènes et d'en vérifier l'intégrité.

1.6 Organisation de la thèse

Nous structurons notre analyse comme suit : dans le chapitre 2, nous décrivons deux architectures collaboratives. Dans le chapitre 3 et 4 nous présentons les bases de connaissances du simulateur ainsi que les ontologies et les normes utilisées dans le domaine électrique. Les chapitres 5 et 6 introduisent les concepts de langage orientés agents, les langages d'actions et la programmation par ensemble de réponses, partie intégrante de la logique non monotone ASP (Answer Set Programming). Dans le chapitre 7, nous exposons les résultats et les observations obtenus et nous terminons la thèse par une conclusion générale et des perspectives futures dans le chapitre 8.

CHAPITRE 2

SYSTÈMES MULTIAGENTS

2.1 Introduction

Pour déterminer l'architecture multiagents la plus appropriée au projet LEOPAR, il est important d'analyser les forces et les faiblesses de ces systèmes. Pour ce faire, nous explorons deux approches : l'architecture des tableaux noirs (ATN) et les systèmes multiagents. Cette comparaison permettra d'identifier ce à quoi doit ressembler le futur système de simulation multiagents à développer.

2.2 Les systèmes monolithiques et multiagents

La simulation est un procédé informatique qui permet de modéliser un phénomène ou un système pour en maitriser la complexité. Elle permet de simuler un système complexe pour lequel il est difficile d'avoir une vue d'ensemble. Pour construire un simulateur, il existe deux approches. La première se nomme simulation *intégrée* ou *monolithique*. Le système se compose d'un seul bloc qui simule l'ensemble du phénomène. Cette approche, bien qu'intuitive, pose problème. L'implémentation d'un simulateur monolithique requiert la contribution et la synergie simultanée de divers experts appartenant à différents domaines (voir Alok Chaturvedi, 2006). Par ailleurs, le simulateur monolithique peut être suffisamment complexe pour empêcher toute extension ultérieure, d'où l'intérêt de recourir à une approche plus modulaire : les systèmes multiagents. C'est une approche qui vise à subdiviser un traitement en sous-éléments traités par des agents. Ces derniers se définissent comme étant des entités autonomes capables à la fois de percevoir et d'agir sur leur environnement dans le but d'atteindre des objectifs propres. Un agent est en mesure de réaliser des traitements de manière autonome ou en collaboration avec d'autres agents (voir Corkill, 2003).

Il existe deux types d'architectures multiagents : les systèmes de tableau noir ou (*Black-Board Systems*) et les systèmes multiagents standards. Ces systèmes ont en commun la capacité de simuler un phénomène à l'aide d'un ensemble d'agents simples et modulaires. Ce qui différencie ces deux types d'architecture réside dans la manière dont collaborent les agents. Les agents des architectures de tableaux noir collaborent afin d'atteindre un objectif global et commun. Dans les architectures multiagents, les agents sont fortement autonomes et poursuivent leurs propres objectifs. Ces différences sont analysées dans les sections suivantes, où nous exposons pour chacun des systèmes le mode de fonctionnement des agents.

2.3 Les architectures tableaux noirs

Une ATN permet de simplifier le traitement d'applications complexes ou difficiles à définir. Elle se compose d'ensemble de modules experts, nommés KS (*Knowledge Source*), spécialisés dans la résolution de tâches spécifiques. Ces KS interagissent à travers des données communes et sont appelés à les enrichir dès qu'ils le peuvent. La zone commune de partage des données se nomme *tableau noir*. La résolution de problème ou simulation se fait de manière incrémentale par l'apport successif de chaque KS. Les ATN offrent une grande flexibilité dans la résolution incrémentale de problème, car ils ne nécessitent pas la description des interactions possibles entre les KS. Ces derniers peuvent être ajoutés ou enlevés du système de manière transparente et flexible.

L'ATN agit de manière opportuniste ; dès qu'un KS reconnait une donnée à laquelle il peut contribuer, il peut s'exécuter (voir Corkill, 1991). Les premiers systèmes remontent à une trentaine d'années. On cite Hearsay-II (voir Erman et Lesser, 1980), HASP/SIAP (voir Nii *et al.*, 1982), CRYSALIS (voir Engelmore et Terry, 1979), RESUN(voir Carver et Lesser, 1991) et OPM (voir Engelmore et Terry, 1979). Il est possible de décrire les ATN selon la métaphore du tableau noir (voir Corkill, 1991). Un ensemble d'experts de divers domaines travaillent ensemble de manière coopérative pour résoudre un problème. Leur coopération est rendue possible par le biais d'une zone commune de travail qu'est le tableau noir. La résolution d'un problème débute quand des données initiales sont écrites sur le tableau. Chaque spécialiste KS observe les données du tableau et réagit en fonction de sa capacité à faire évoluer le problème. Dès qu'un spécialiste se croit capable de contribuer à la résolution du problème, il l'écrit dans le tableau.

Ce processus de contributions successives de la part des experts dans le tableau noir continue de la sorte jusqu'à la résolution du problème. Dans notre exemple, les spécialistes représentent les KS (*Knowledge Source*). Ces derniers peuvent être des systèmes experts, des simulateurs, des agents ou toute autre entité capable de traiter de l'information. Les KS ne communiquent pas entre eux, mais uniquement par l'entremise des données du tableau. Dans le cas où un KS identifie un modèle de données dans le tableau, il y contribue de manière autonome.

Dans chaque ATN, il existe trois types de composants : le tableau noir, un ensemble de sources de connaissances (KS) et un mécanisme de contrôle. Le tableau noir représente un bassin de données commun qui renferme le problème à résoudre ainsi que des solutions partielles possibles (nommées hypothèses). Les KS examinent l'état du tableau noir et créent ou modifient des hypothèses quand ils sont en mesure de le faire. Cette caractéristique permet au système de résoudre un problème donné selon une approche incrémentale et opportuniste.

Il s'agit d'une caractéristique importante pour la résolution de problèmes complexes. Un mécanisme de *contrôle* permet de guider la progression de la résolution du problème. Ce mécanisme coordonne l'exécution des KS dans le cas où plusieurs KS sont en compétition pour être exécutés. Le contrôleur évalue la priorité d'exécution de ces KS en déterminant la manière dont chaque KS peut contribuer à la résolution du problème pour choisir le plus pertinent (voir Rabin, 2002). Le choix du KS à exécuter se fait en fonction de la pertinence immédiate, mais aussi de l'objectif global visé. Le contrôleur est guidé par les données (Data-Driven), mais aussi par les objectifs (Goal-Driven) à atteindre (voir Carver et Lesser, 1992) (voir figure 2.1).

Ainsi, l'ATN a l'avantage d'être modulaire et appropriée aux approches incrémentales et émergentes. La réalisation de ce système peut être aisée dans le cas où il y a un nombre réduit de KS. Leur synchronisation a recours à un contrôleur simple qui obéit au phénomène d'opportunisme, c'est-à-dire le KS qui contribue le plus à la résolution du problème est exécuté. Toutefois, ces architectures deviennent rapidement difficiles à gérer dans le cas où il y a une grande interaction entre les KS, ce qui oblige à la mise en place d'un module de contrôle complexe et fortement spécialisé.

Une autre limite inhérente au contrôleur réside dans le fait qu'il ne tient pas compte des phénomènes complexes de synchronisation, d'exécution concurrente ou parallèle. En effet, de part sa nature émergente, le contrôleur sélectionne la meilleure action à exécuter en fonction des effets immédiats du KS et de sa contribution à l'achèvement de l'objectif final.

Toutes ces limites expliquent vraisemblablement le manque d'engouement pour le développement des tableaux noirs.

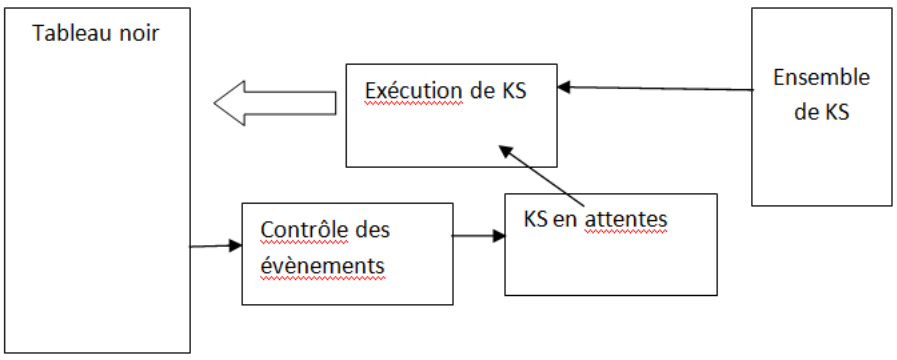


Figure 2.1 Architecture du tableau noir

La question qui ressort est de savoir comment tirer profit des architectures tableau noir pour réaliser un système de simulation modulaire. Quelle approche adopter pour développer

un système de simulation simple d'utilisation et fortement extensible ?
À ce niveau il est possible de déduire que l'avantage des ATN pour le projet LEOPAR réside dans :

1. La décomposition de la simulation en fonction des centres d'expertise. Chaque spécialiste développe son propre simulateur ou KS qui vient se greffer aisément dans une ATN.
2. La réutilisation de simulateurs ou KS préalablement développés. Il devient possible de réutiliser d'anciens simulateurs ou KS pour les intégrer dans des ATN.
3. L'ATN facilite l'interaction et la communication des agents, car elle se fonde sur la mise en commun des données.

2.4 Les architectures multiagents

Ces dernières années, les Systèmes multiagents standards (SMA) sont apparus comme étant un paradigme prometteur où l'objectif est de faire collaborer un ensemble d'agents autonomes, réactifs et proactifs. Cette approche architecturale octroie aux agents une grande autonomie en leur permettant de décider par eux-mêmes des actions à exécuter. L'architecture multiagents est considérée comme une avancée par rapport aux architectures tableau noir, car elle résout deux limites majeures : la centralisation des données et la centralisation de la coordination. Afin de mieux comprendre l'architecture d'un système multiagents, nous présentons la figure 2.2 :

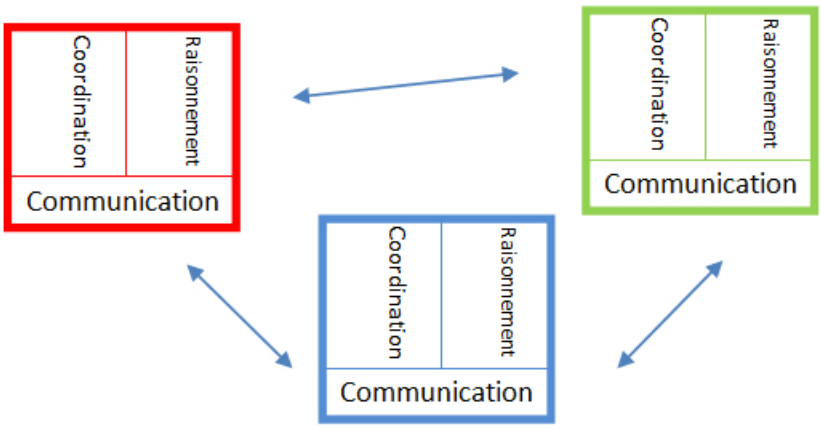


Figure 2.2 Architecture système multiagents

Théoriquement, dans les systèmes multiagents, les interactions se font par l'échange de messages entre les acteurs. La zone centrale de stockage des données de l'ATN est remplacée par des agents capables de raisonner et de partager de l'information. Le module de coordination n'est plus centralisé, il devient intrinsèque à chaque agent.

Les SMA octroient une plus grande autonomie aux agents, qui deviennent capables de communiquer selon différents niveaux d'abstraction. Or, octroyer une grande autonomie aux agents pose de nouveaux défis relatifs à la coordination, la communication et la transmission de messages. La communication devient un mécanisme centrale pour les agents, car elle est à l'origine de toutes sources de coordination et de collaboration.

Conceptuellement, les agents sont dotés de trois modules : un module de communication qui permet de recevoir et de transmettre de l'information, un module de coordination qui permet à l'agent d'entreprendre des actions individuelles ou en groupe et un module de raisonnent qui stocke et traite les connaissances dont dispose l'agent sur ses objectifs et sur son environnement. Analysons en détail ces modules pour comprendre leurs forces et faiblesses. Cette analyse va permettre de cerner les besoins du projet LEOPAR.

2.4.1 Le module de communication

L'interaction entre les agents est possible par le biais de la communication et l'échange d'informations. Les agents communiquent avec les utilisateurs, les ressources du système et les autres agents dans le but d'accomplir leurs objectifs. Pour collaborer, les agents ont besoin d'utiliser le même langage pour échanger des informations. C'est ainsi qu'en 1995 le langage FIPA ACL a été élaboré afin de standardiser la communication inter-agents.

Le langage FIPA identifie deux niveaux d'information lors de la communication. Premièrement, l'agent doit déterminer le type du message selon 22 types distincts. Par la suite l'agent détermine le contenu du message, sa grammaire et sa sémantique. Généralement le contenu du message est formalisé à l'aide d'une ontologie commune qui est un support par excellence pour désambigüiser une information (voir Wooldridge, 2009). Un message FIPA ACL ressemble à ce qui suit :

```
[inform
:sender    agent1
:receiver  agent2
:content   (powerTransformer hasPrice 150)
:language  SL
:onto1ogy  CIM14.owl
]
```

La performative « inform » :

La performative indique le type du message à transmettre. Elle est le résultat d'études basées sur l'acte du langage (*speech acts*) (voir Searle, 1969). Quand un agent décide de communiquer avec un autre agent, il doit déterminer le type de la communication. Pour le standard FIPA, il existe actuellement 22 types de performatives réparties en 5 catégories (information, requête, négociation, action et erreur). Parmi les performatives les plus usuelles nous citons : informer (*inform*), demander (*ask*), accepter (*accept*), et refuser (*decline*).

L'expéditeur et le destinataire du message, de « agent1 » à « agent2 » :

Une fois que le type du message est identifié, il y a détermination du destinataire du message. Cette étape est généralement réalisée par le module de coordination et plusieurs techniques existent pour permettre à un agent de cibler un interlocuteur. Dans le cas où l'agent est l'initiateur de la communication, il a la possibilité d'envoyer un message à tous les agents en mode diffusion (*broadcast*) pour identifier un destinataire. Cette diffusion peut être restreinte à une zone d'influence. Toutefois, il est possible dans un SMA d'avoir recours à un agent particulier (*Directory facilitator*) qui permet d'établir le lien entre l'agent et les services qu'il offre, à l'image d'un annuaire téléphonique. Ainsi, l'agent annuaire stocke et met à jour les agents et leurs compétences.

Le langage utilisé « SL » :

Cette information spécifie le langage utilisé pour le contenu du message transmis. À noter qu'il existe différentes représentations telles que le RDF, SL ou le KIF. Seul le FIPA-SL est promu comme standard par le W3C et s'appuie sur une ontologie pour formaliser un message.

L'ontologie utilisée « CIM14.owl » :

Il est primordial que le message transmis d'un agent à un autre soit compris par les deux intervenants. La communication nécessite une compréhension identique de la terminologie utilisée. L'ontologie est utilisée afin de désambigüiser les termes et permettre de véhiculer une information précise d'un agent à un autre. Nous définissons l'ontologie comme une hiérarchie de termes reliés par des concepts logiques. Nous nous attarderons avec plus de détails sur la définition des ontologies dans le chapitre 3.

2.4.2 Le module de raisonnement et de prise de décision

La capacité de l'agent à raisonner est essentielle dans un SMA, car les actions entreprises par l'agent lui permettent d'atteindre des objectifs qui sont parfois en contradiction avec les objectifs d'autres agents. La prise de décision devient une étape complexe qui nécessite de la part de l'agent de prendre en compte ses propres contraintes ainsi que celles de l'environnement. Les techniques de raisonnement les plus courantes sont :

Raisonnement déductif :

Le raisonnement déductif utilise une représentation symbolique/logique des connaissances de l'agent afin de déduire un comportement cohérent. Le principal défi consiste en la traduction des objectifs de l'agent ainsi que l'environnement dans lequel il évolue en un formalisme symbolique/logique robuste. Cette connaissance formalisée est par la suite utilisée par l'agent afin de réaliser des raisonnements de déduction en un temps limité. Le raisonnement déductif est une approche puissante, car elle est bâtie sur de la logique mathématique. L'exemple suivant est une représentation logique déductive qui décrit la sortie d'un agent d'une chambre fermée :

$RobotInRoom \cap DoorLocked \cap \neg HaveKeys \rightarrow TakeKeys.$
$RobotInRoom \cap TakeKeys \rightarrow HaveKeys.$
$RobotInRoom \cap HaveKeys \cap DoorLocked \rightarrow DoorOpen.$
$RobotInRoom \cap DoorOpen \rightarrow RobotOutRoom.$

Parmi les techniques de raisonnement logique, il existe l'architecture BDI (Believe, Desire, Intention) qui offre une architecture structurée. Elle informe sur le pourquoi d'une action et sur les raisons de celle-ci. L'approche BDI introduit des états mentaux et l'agent dispose d'un ensemble d'objectifs à réaliser (les désirs), d'un ensemble d'actions à entreprendre (les intentions) et d'un ensemble de connaissances sur l'environnement (les croyances). Le Framework JADEX est un exemple d'environnement de développement d'architecture BDI.

La logique temporelle/modale introduite par le langage MetateM (voir Fisher, 2005) est une autre technique de raisonnement déductif. Elle utilise la temporalité pour représenter les connaissances de l'agent. Ce dernier dispose d'un ensemble de règles qui lui permettent de déduire des faits/actions futurs en fonction de fait/actions passés. Ainsi, la temporalité des événements joue un rôle important, car elle réfère à un historique pour établir un plan d'action de l'agent. L'action devient temporelle. À titre d'information, le langage MetateM introduit les concepts temporels *"tant que, jusqu'à, depuis, toujours, futur* et *passé"*.

Toutefois, les limites de l'approche déductive résident dans sa complexité. Elle nécessite des connaissances poussées en logique, raisonnement et représentation des connaissances. Le développement d'un agent déductif nécessite l'intervention d'un expert, car la complexité de la base de connaissances croît avec le volume des connaissances et règles logiques. De plus, en logique, il est compliqué de représenter des concepts d'incertitude, de dynamisme, de temporalité ou d'exception. La complexité des langages et formalisment associés au raisonnement déductif est en opposition avec l'objectif premier du projet LEOPAR : offrir un système accessible à des non experts. Il existe toutefois, une alternative au raisonnement déductif : le raisonnement réactif.

Raisonnement réactif :

La limite principale de l'approche symbolique/logique est imputable à la complexité du raisonnement nécessaire et conséquemment à la lenteur du temps de réaction des agents. Le raisonnement réactif vient en réponse à ces limites. Il stipule que certaines formes d'intelligence sont la résultante de réactions réflexes simples. Ce constat s'appuie sur l'observation du comportement de certains organismes vivants, à l'image des insectes ou racines de plantes. D'après les défenseurs du raisonnement réactif, l'intelligence peut-être le résultat de l'interaction d'actions simples ou réflexes.

Parmi les agents réactifs, nous citons l'architecture de subsomption (voir Brooks, 1990). Les agents disposent d'un réseau d'automates d'états finis qui associent des stimuli externes à des actions (voir figure 2.3). Les automates ne disposant pas de mécanisme de raisonnement symbolique/logique, il en résulte la nécessité d'un mécanisme pour prioriser une action par rapport à une autre. Comme les automates peuvent mener à l'exécution d'une ou plusieurs actions, la priorisation devient indispensable pour éviter un comportement incohérent de l'agent.

En résumé, le raisonnement réactif permet de développer des agents robustes. Le plus grand avantage d'un tel mécanisme est la rapidité de réaction des agents ainsi que la facilité d'implémenter rapidement des agents simples. Cette technique de raisonnement est particulièrement appréciée dans le domaine de la robotique ou toute action doit se faire en un temps limité. Simplicité, robustesse et rapidité sont les principaux avantages de cette technique.

Toutefois, le modèle réactif présente de nombreuses limites. Utiliser le raisonnement réactif dans le cadre du projet LEOPAR rendrait impossible la spécification de processus de collaboration entre les agents, car les actions des agents se basent uniquement sur un processus réflexe. Le raisonnement réactif ne permet pas de représenter des comportements complexes.

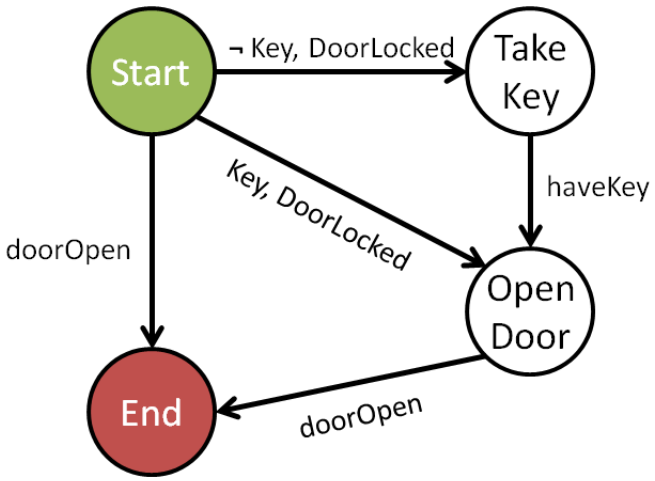


Figure 2.3 Exemple d'automates à états finis

Raisonnement hybride :

Cette approche vient en réponse à la fois aux limites du raisonnement déductif, qui est complexe et onéreux en temps de calcul, et à celles du raisonnement réactif, qui est trop simple pour élaborer des comportements complexes. Un agent hybride est un mélange de processus de raisonnement réactif et déductif. Tout le défi consiste à équilibrer ces deux types de raisonnement et de les transposer en un seul environnement multiagents.

Ainsi sont nés les agents hybrides dotés d'une hiérarchisation des niveaux d'interaction. Les niveaux conceptuels les plus bas contiennent des actions réactives en réponse à un stimulus particulier de l'environnement. Ce sont des actions réflexes. Les niveaux conceptuels les plus hauts sont des niveaux décisionnels plus complexes nécessitant un véritable processus de déduction qui implique la coopération, la coordination ou la planification.

Il est important de comprendre les avantages et les limites de ces architectures pour déterminer, selon les besoins de l'application, le meilleur type de raisonnement à adopter, car le meilleur langage orienté agent dépend de l'application à développer (voir Wooldridge, 2009).
À titre indicatif, le raisonnement hybride représente l'approche la plus commune. Parmi ces langages : 3APL, Jason, IMPACT, GO !, AF-APL (AGENT0 + BDI concepts).

2.4.3 Le module de coordination

Le module de coordination définit le protocole sur lequel se base l'agent pour réaliser une interaction avec d'autres agents. Il intervient quand l'agent n'est pas en mesure de réaliser de manière autonome un objectif particulier. Ainsi, le module de coordination détermine

comment l'agent doit agir pour solliciter l'aide et la collaboration d'autres agents.

Analysons le mode de coordination des systèmes multiagents afin de savoir dans quelle mesure il serait possible d'appliquer les mêmes approches pour notre système. Les principales techniques de coordination sont les suivantes :

Structure organisationnelle :

Cette méthode consiste à permettre à un ou plusieurs agents de haut niveau d'être responsables de la coordination d'un ensemble d'agents, et ainsi de jouer le rôle de chef d'équipe. C'est une relation de maître/esclave, qui est toutefois complexe et qui va à l'encontre de la philosophie de la décentralisation du module de coordination (voir Bellifemine *et al.*, 2007). Adopter une telle structure dans la cadre du projet LEOPAR nécessiterait de disposer d'un planificateur/coordinateur performant, qui soit en mesure de décrire facilement des phénomènes et des contraintes évoluées.

Contract net protocol :

Cette technique s'appuie sur une approche de mise en marché pour distribuer des tâches dans un groupe d'agents. Le protocole fait intervenir deux types d'acteurs, les acheteurs (*Participants*) et le vendeur (*Initiator*). Dans le cas où un agent a besoin de l'intervention d'autres agents, il les sollicite sous forme de contrat afin d'enclencher une coordination pour résoudre une sous-partie du problème. Cette approche se compose de quatre étapes : le vendeur envoie un appel de proposition (*call for proposal*), chaque acheteur renvoie une offre, par la suite le vendeur choisit la meilleure de toutes les offres et informe les agents du rejet des autres offres. Toutefois, ce protocole souffre de manque de robustesse, car il dépend du temps mis par les acheteurs pour répondre à une offre, ce qui peut résulter à un blocage (voir Bellifemine *et al.*, 2007; Alibhai et Sc, 2003).

Planification multiagents :

Cette technique de coordination résout le problème de coordination à l'aide de planificateurs. Une première approche consiste en la construction par chaque agent d'un plan d'action. Par la suite, un planificateur central est utilisé afin de vérifier la consistance de tous les plans parcellaires issues des agents et détermine si les plans sont consistants. La deuxième approche élimine le planificateur central. L'idée est de fournir aux agents une connaissance des plans des autres agents. Ainsi, l'agent va disposer de ses propres connaissances ainsi que celles des autres agents afin d'éviter de construire des plans qui sont en contradiction avec les autres agents. À titre d'exemple, l'équilibre de Nash permet à un agent d'agir en fonction de l'objec-

tif des autres agents (voir Bellifemine *et al.*, 2007). Ces techniques de planification présentent un grand intérêt pour le projet LEOPAR. Toutefois, le plus grand défi réside dans le choix du langage de planification à adopter, car il se doit d'être accessible, expressif et performant.

Les conventions sociales :

Les normes et les lois sociales sont une source de contrôle et de coordination et permettent aux agents de décider de la légalité d'une situation (voir Vlassis, 2007). Ces règles structurent les actions des agents à travers un cadre. Elles facilitent la prise de décision de la part de l'agent, car elles imposent l'obligation d'exécuter une série d'actions selon certaines situations. Il est important que les règles sociales soient partagées par la majorité des agents afin de permettre la convergence du système. Il n'en demeure pas moins qu'il est possible de permettre à un nombre restreint d'agents d'être des *hors-la-loi* afin d'introduire une certaine variabilité dans le système. Ce type d'organisation sociale s'applique idéalement pour des systèmes disposant d'un nombre élevé d'agents qui réalisent des interactions nombreuses, à l'image de la société humaine. Or, ce type d'approche ne convient pas au projet LEOPAR.

2.5 Les langages orientés agents

Pour faciliter le développement des systèmes multiagents plusieurs langages de programmation orientés agents ont vu le jour. Le tableau 2.1 énumère les principaux langages agents en fonction du type de raisonnement :

Une étude des langages orientés agents révèle que la limite première de ces langages réside dans leur complexité vis-à-vis des non-experts. Comme la majorité de ces langages utilisent la logique de premier ordre ou la logique modale temporelle, il est difficile pour un non-expert d'utiliser aisément ces langages (voir Gaha *et al.*, 2011). Par ailleurs, cette complexité est d'autant plus importante que le protocole de communication, généralement basé sur FIPA, ajoute un niveau de difficulté quant à la manière dont communiquent les agents. Dans notre cas, le choix du langage de programmation pour le futur système de simulation est important, car il se doit d'être extensible et accessible à des non-experts en système multiagents. Une description plus détaillées de ces langages se retrouve en partie annexe B.

2.6 L'architecture tableau noir et les systèmes multiagents

Les systèmes multiagents apportent de nouvelles possibilités et permettent de réaliser des systèmes distribués qui sont en mesure de surpasser les limites des ATN. Les systèmes mul-

Tableau 2.1 Comparaison de langages orientés agents

Langages agents	Raisonnement déclaratif	Raisonnement réactif	Raisonnement hybride	Formalisme
Minerva + Kabul	X			Logique non monotone
Concurrent MetateM	X			Logique temporelle et modale
Behavior nets		X		Réseau d'actions
3APL			X	Prédicat premier ordre et FIPA
Jason			X	Prédicat premier ordre
IMPACT			X	Prédicat premier ordre et instrcutions
GO			X	Programmation symbolique et logique
Agent0			X	Logique modale et instructions
AF-APL			X	Programmation logique et procédurale

tiagents présentent les avantages suivants par rapport aux ATN :

- La distribution des données : il n'y a pas de zone de stockage commune des données, comme c'est le cas pour les ATN. Le phénomène du goulot d'étranglement est évité.
- La coordination : chaque agent dispose d'un mécanisme de coordination interne qui lui permet de raisonner sur ses données et d'établir un plan d'action pour atteindre ses propres *objectifs*. Ainsi le processus de coordination devient distribué et propre à chaque agent.
- Les interactions : la communication entre les agents ne se fait plus par l'entremise de données. Les agents communiquent directement et échangent des informations. Il s'ensuit des interactions plus élaborées.

Toutefois, une analyse des architectures de tableau noir fait ressortir les avantages suivants par rapport aux systèmes multiagents :

- Une architecture simple : ce constat se fonde sur le fait que les interactions entre les agents se basent uniquement sur les données. Il n'existe pas de protocole de communication élaboré à respecter.
- Des agents simplifiés : les agents réalisent des raisonnements uniquement sur les données du tableau noir. Les agents ont une vue globale du problème et n'ont pas besoin de décider quels sont les résultats à partager, avec qui communiquer et comment répondre à une communication (voir Corkill, 2003) . Ces caractéristiques combinées au contrôleur du tableau noir permettent de résoudre des problèmes mal définis ou complexes.

Tableau 2.2 Comparaisons des architectures collaboratives

Système	Coordination	Données	Objectif	Extensibilité	Accessibilité	Performance
ATN	Centralisée	Centralisées	Global	Haute	Haute	Faible
SMA	Distribuée	Distribuées	Multiple	Moyenne	Moyenne	Haute
HLA	Distribuée	Distribuées	Global	Faible	Faible	Haute

Le tableau 2.2 présente différents systèmes collaboratifs. Nous y comparons les architectures de tableaux noirs, l'architecture HLA et les architectures multiagents classiques. Bien que nous n'ayons pas décrit le système HLA (High Level Architecture/IEEE Standard), nous l'incorporons dans le tableau récapitulatif car c'est une architecture récente. Elle offre un cadre ou les composants, rôles et types de messages sont identifiés et standardisés. Pour plus d'information sur cette architecture, nous invitons le lecteur à se référer à l'annexe [1] C. Nous comparons ces systèmes selon la nature de la coordination des agents, la manière dont les données sont réparties, les types d'objectifs des agents, la facilité d'intégrer de nouveaux agents, la facilité de compréhension du fonctionnement du système et finalement les niveaux de performance des architectures. L'objectif que nous nous fixons est de réaliser un système multiagents qui permette de :

- Offrir un système extensible : il s'agit de développer un système qui permette d'introduire et de retirer des agents hétérogènes sans compromettre le fonctionnement global du système de simulation multiagents.
- Offrir une architecture accessible : il s'agit d'offrir un environnement de simulation accessible aux non-experts en leur permettant de modifier aisément les objectifs du système.

1. Ce choix se base sur le fait que notre architecture ne s'appuie pas sur le modèle HLA.

– Offrir une architecture distribuée : il s'agit de permettre l'exécution distribuée et en parallèle des agents ainsi que la décentralisation des données pour éviter tout goulot d'étranglement et perte de performance du système.

Pour récapituler, le système que nous souhaitons développer (**Leopar++**) pour le compte de l'IREQ doit avoir les caractéristiques suivantes (voir tableau 2.3) :

Tableau 2.3 Propriétés du système de simulation Leopar++

Système	Coordination	Données	Objectif	Extensibilité	Accessibilité	Performance
Leopar++	Distribuée	Distribuées	Global	Haute	Haute	Haute

Le système Leopar++ se base sur une architecture nouvelle qui est connue sous le nom de multiagents tableau noir (voir Zhu *et al.*, 2010; Qiang et Liu, 2010; Hammami *et al.*, 2009). Elle jumèle deux architectures pour permettre une interaction facilitée et une grande autonomie des agents. En effet, l'approche des systèmes multiagents tableau noir vise à simplifier le développement des agents en usant d'une zone commune de données partagées pour faciliter la communication et l'interaction (voir Kao *et al.*, 2002; Dong *et al.*, 2005).

CHAPITRE 3

INTRODUCTION AUX BASES DE CONNAISSANCES

3.1 Introduction

Notre travail ambitionne d'apporter une contribution dans le domaine de la représentation et du traitement des connaissances afin d'offrir une architecture à mi-chemin entre les ATN et SMA. En définissant une plateforme robuste et polyvalente, il est possible de répondre aux besoins du projet LEOPAR en développant un système multiagents tableau noir robuste et accessible (voir Figure 3.1).

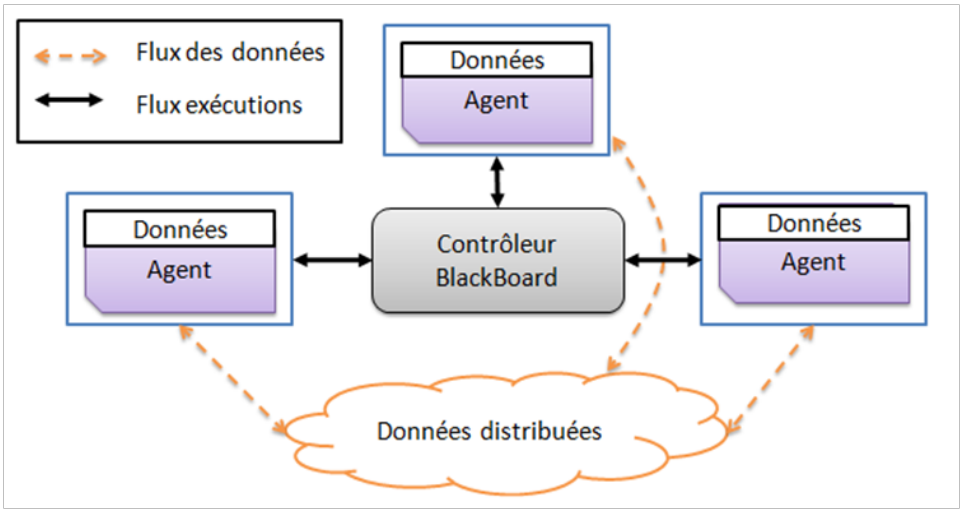


Figure 3.1 Architecture Leopar++

L'architecture du système Leopar++ intègre des technologies issues de différents domaines de recherche, à savoir les ontologies et la représentation des connaissances, les algorithmes de planification, les systèmes multiagents et les architectures de tableau noir. À l'image des systèmes informatiques, Leopar++ utilise et traite des données. Celles-ci sont fondamentales et leur structure influe sur les capacités du système à extraire et à traiter l'information. Un formalisme de représentation non adapté aux besoins d'un système informatique nuit à son bon fonctionnement, soit par une carence, soit par un excès d'expressivité. De ce fait, il est important de déterminer le formalisme, l'expressivité et le pouvoir d'inférence du langage de représentation des données.

3.2 Les bases de connaissances

Parmi les nombreux langages de représentation des connaissances, la logique descriptive est un formalisme puissant. Elle est utilisée dans des systèmes à base de connaissances pour désambigüiser, fusionner ou inférer de nouvelles données. Analysons de plus près ce formalisme et étudions son utilité pour le système Leopar++.

3.2.1 Logique descriptive et OWL (Web Ontology Language)

Le langage OWL est né de la volonté d'exprimer des connaissances consistantes et non contradictoires à l'aide de la logique descriptive. Les connaissances ainsi formalisées sont exploitables aussi bien par l'homme que par la machine (voir Eiter *et al.*, 2006). Standardisé par le W3C, le langage OWL permet une représentation formelle d'un ensemble de concepts reliés par des relations selon des quantificateurs logiques. La standardisation du langage, ainsi que le recours à la logique de description, permet une désambigüisation des concepts et des relations décrivant les connaissances du domaine.

Ainsi, le rôle premier d'une ontologie OWL, contrairement aux différents formalismes tels que le RDF, les bases de données ou le XML, est de donner une description logique aux connaissances représentées.

Plus généralement, une ontologie OWL représente, à l'aide de la logique descriptive, des connaissances sous forme de réseaux de nœuds et de liens (voir Eiter *et al.*, 2006). Il existe différentes familles de logiques descriptives qui offrent une expressivité variable. Parmi les familles de formalismes existants, nous citons le formalisme OWL2-RL, qui octroie un niveau d'expressivité plus important que son prédécesseur OWL-DL. Il permet d'exprimer les relations suivantes :

- **Hiérarchie** : il est possible de décrire des relations d'équivalence et d'inclusion des classes. Par exemple il est possible de décrire qu'un disjoncteur est sous-classe des composants électriques : $Disjoncteur \subseteq ComposantElectriques$
- **Nominal** : il est possible de spécifier une classe en énumérant ses membres et d'interdire à d'autres individus d'y appartenir. L'expression suivante indique qu'un conducteur électrique ne peut être que aérien ou souterrain. $TypeConducteurElectrique = \{souterrain, aerien\}$
- **Inverse** : il est possible de décrire une relation et son inverse. Par exemple, l'expression suivante : $alimente(a, b) \rightarrow estAlimente(b, a)$ indique que dès lors où a alimente b alors il est aussi possible d'affirmer que b est alimenté par a .
- **Quantificateur numérique** : il est possible de restreindre numériquement la cardina-

lité des relations entre les classes. Par exemple, $ComposantElectrique \subseteq Composant \cap \leqslant 1.aTerritoire.Territoire$, indique qu'un composant est localisé dans au maximum un territoire.

- **Chainage de propriétés** : il est possible de définir des chaines de propriétés. Par exemple, décrire que tous les composants du même poste sont localisés dans la même région que le poste. L'expression suivante $aPoste \circ aRegion \equiv aRegion$ indique que si $aPost(a, b); aRegion(b, c)$; alors il il est possible de conclure que $aRegion(a, c)$.

Il permet aussi de déclarer des propriétés de relations :

- **Transitivité** : il est possible de définir des relations de transitivité. Par exemple si $R(a, b) \wedge R(b, c) \rightarrow R(a, c)$.
- **Symétrie** : il est possible de définir une symétrie pour certaines relations : $R(a, b) \rightarrow R(b, a)$. Exemple la relation *etreConnecteA* est symétrique. Si un composant est connecté à un autre composant, donc il sont inter-connectés.
- **Asymétrie** : il est possible de définir une asymétrie pour certaines relations : $R(a, b) \rightarrow \neg R(b, a)$. Exemple la relation *appartenirAuPoste* est asymétrique. Si un composant appartient à un poste, l'inverse n'est pas vrai.
- **Réflexive** : informe qu'une relation est réflexive : $R(a, a)$. Par exemple la relation *avoirComposantSucceseur* peut être réflexive. La relation peut s'appliquer à la classe *Composant* de manière réflexive.
- **Non réflexive** : informe qu'une relation ne peut pas être réflexive : $\neg R(a, a)$. Par exemple la relation *appartenirReseau* est non réflexive. Un composant électrique ne peut pas appartenir à un autre composant.
- **Fonctionnelle** : il est permis de définir une relation fonctionnelle. Par exemple, la relation *appartenirPoste* est fonctionnelle.
- **Cardinalité qualifiée** : ce type de cardinalité permet d'imposer un nombre minimal ou maximal de relations de peut avoir une classe envers une autre classe. Par exemple, cette relation permet de décrire des composants qui ont connu plus de 2 pannes.
- **Typée** : il est possible d'identifier la classe source et destination d'une relation. Par exemple il est possible de typer la relation *appartenirAuPoste* pour qu'elle fasse référence uniquement aux classes *Composant* et *Poste*.

Tout comme la logique descriptive, le langage OWL se subdivise en deux parties : la Tbox et la Abox. La TBox renferme la définition des concepts et les relations qui lient les concepts. Elle contient tous les axiomes définissant le domaine. La ABox renferme les individus qui sont

les instances des concepts définis dans la TBox. Il est possible de faire un parallélisme entre la logique descriptive et la programmation orientée objet où les classes et instances de classes sont respectivement la TBox et la ABox. Ces deux parties sont traduites à l'aide du langage OWL via les triplets (sujet, prédicat et objet) inspirés du modèle des graphes (Figure 3.2).

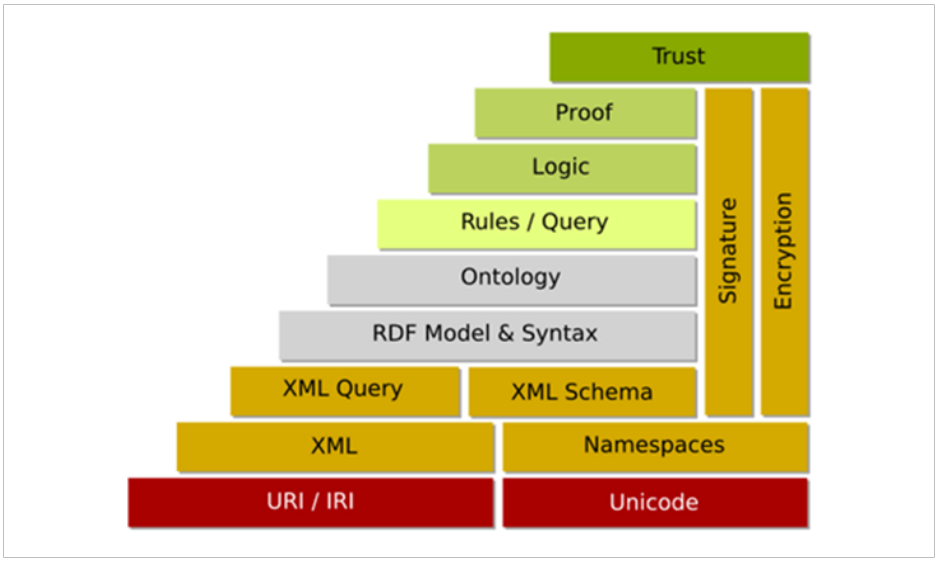


Figure 3.2 Les couches du web sémantique

Il ressort que la technologie OWL fait partie d'un ensemble de technologies et de langages construits en couches. Brièvement, nous décrivons ces technologies dans le tableau 3.1

Les ontologies font partie intégrante d'une succession de technologiques issues Web sémantique. Cette particularité octroie à l'ontologie OWL une grande puissance quant à la représentation et la manipulation de données hétérogènes et distribuées. Cette caractéristique est primordiale pour le projet LEOPAR qui vise à unifier les sources de données d'Hydro-Québec. Nous invitons le lecteur désireux de mieux comprendre les liens qui unissent le web sémantique aux ontologies OWL de se référer à l'excellent manuel en ligne (voir Gagnon, 2012).

3.2.2 Sémantique du langage OWL

Afin de mieux comprendre l'ontologie et ses implications, analysons de plus près le langage OWL et sa sémantique. Il est important de saisir la sémantique du langage, car nous allons avoir recours à la logique OWL pour définir des concepts électriques.

Tableau 3.1 Description des couches du web sémantique

Couches	Définition et Objectif
URI	Le *Uniform Ressource Identifier* est un système de nommage de ressource sur un réseau. Il permet d'identifier par exemple une ressource web physique ou abstraite.
XML	Le *eXtended Markup Language* est un langage de balisage pour l'échange de données informatiques. Les données sont représentées sous forme d'arbre afin de faciliter l'échange automatique de données complexes entre les systèmes.
RDF	Le *Ressource Description Framework* est un modèle de graphe permettant de représenter formellement des ressources Web sous une forme traitable par la machine.
Ontologie	La couche ontologique utilise la logique descriptive pour décrire des données. L'ontologie permet de raisonner sur les données pour en déterminer la consistance et la cohérence.
Règles	Les règles permettent d'octroyer une plus grande expressivité aux ontologies. Elles introduisent des conditions pour exprimer des connaissances conditionnelles.
Logique	Elle est une couche hypothétique qui vise à permettre de définir une logique particulière pour représenter des connaissances spécifiques telles que l'incertitude ou le raisonnement par défaut.
Preuve	La preuve est une couche hypothétique qui vise à permettre à un utilisateur ou système de comprendre le pourquoi d'une information. Les prémisses qui ont abouti aux informations inférées.
Confiance	Elle est une couche hypothétique qui vise à octroyer un niveau de confiance aux données et informations inférées. L'agent informatique doit être en mesure de prouver les conclusions et de garantir la fiabilité des informations utilisées.

Formellement, supposons que Δ non vide renferme les éléments du domaine à décrire. Supposons que **A** représente les concepts et **R** les relations entre les concepts. Admettons qu'une fonction d'interprétation I se définie de la manière suivante :

- $I(A) = A^I \subseteq \Delta$. L'interprétation des concepts fait partie du domaine Δ.
- $I(R) = R^I \subseteq \Delta^I \times \Delta^I$ avec $\Delta^I \times \Delta^I$ relation binaire. L'interprétation d'une relation est une relation binaire liant des concepts du domaine Δ.
- $I(\top) = \Delta^I$. L'interprétation de la totalité $\top$ représente tous les concepts du domaine Δ.
- $I(\bot) = \varnothing$. L'interprétation du rien représente l'ensemble vide.
- $I(\neg C) = \Delta^I \backslash C^I$. L'interprétation de la négation d'un concept $C \in A$ représente

tous les concepts du domaine Δ à l'exception de ceux contenus dans A. Par exemple $I(\neg Transformateur)$ représente tout ce qui n'est pas un Transformateur.
- $I(C_1 \cap C_2) = C_1^I \cap C_2^I$. Représente l'intersection binaire de deux concepts. Par exemple $ComposantDefectueux \cap Disjoncteur$ retourne tous les disjoncteurs qui sont défectueux.
- $I(C_1 \cup C_2) = C_1^I \cup C_2^I$. Représente l'union binaire de deux concepts. L'expression $Disjoncteur \cup Interrupteur \cup Sectionneur$ représente la classe de tous les éléments de sectionnements.
- $(\exists R.C)^I = \{x \in \triangle | \exists y : (x,y) \in R^I \wedge y \in C^I\}$. Représente l'existence d'une relation R avec un concept C. Par exemple $\exists aPoste.Poste$ décrit une propriété indiquant obligatoirement une relation d'appartenance à un poste.
- $(\forall R.C)^I = \{x \in \triangle | \forall y : (x,y) \in R^I \rightarrow y \in C^I\}$. Représente l'universalité d'une relation R avec un concept C. Par exemple $\forall aComposantSuivant.Composant$ informe que dans le cas où l'individu à une relation $aComposantSuiant$ alors elle doit obligatoirement être liée à un autre composant.
- $(\leqslant nR)^I = \{x \in \triangle | \ \#(\{y|(x,y) \in R^I\}) \leqslant n\}$. Représente un nombre maximum de relations R possibles avec un concept. Par exemple $\leqslant 3aComposantSuivant.Composant$ indique qu'un individu ne peut avoir plus que 3 connexions de type $aComposantSuiant$.
- $(\geqslant nR)^I = \{x \in \triangle | \ \#(\{y|(x,y) \in R^I\}) \geqslant n\}$. Représente un nombre minimum de relations R possibles avec un concept.
- $(R^-)^I = \{(a,b)|(b,a) \in R^I\}$. Représente l'inverse d'une relation R.

Intégrer des opérateurs logiques permet d'appliquer un mécanisme d'inférence sur l'ontologie du domaine. En effet, il est possible de réaliser des **inférences** sur une ontologie OWL. L'inférence désigne la capacité d'un programme informatique d'extraire de nouvelles connaissances qui s'opère aussi bien sur les concepts de la TBox et les individus de la ABox. Analysons ensemble les types d'inférences pouvant être réalisées :

- Subsomption (TBox) : elle permet de déterminer si les concepts C_1 et C_2 sont *subsumés* $I(C_1) \subseteq I(C_2)$. Par exemple, si $Composant \equiv Interrupteur \cap \exists aMecanisme.Sectionnement$. On peut conclure par inférence que le concept $Interrupteur$ est sous-classe du concept $Composant$: $Interrupteur \subseteq Composant$. Autrement dit, si un sous-ensemble de caractéristiques décrivant C_1 est nécessaire et suffisant pour décrire C_2, alors C_1 subsume C_2.
- Satisfaisabilité (TBox) : un concept C_1 est *satisfaisable* si $I(C_1) \neq \perp$. Dans le cas contraire un concept est dit non satisfaisable si $I(C_1) \subseteq \perp$, c'est-à-dire pour lequel aucune instance ne peut exister. Cette situation est normalement due à des erreurs de

conception ou de peuplement de l'ontologie. Par exemple, si on déclare que $Composant \cap Interrupteur \subseteq \perp$ et $Interrupetur \subseteq Composant$, alors par inférence on déduit la non-satisfaisabilité de la base de connaissances. Autrement dit, dire que $Interrupteur$ est à la fois sous-classe et dissocié de $Composant$ est non satisfaisable.

- Équivalence (TBox) : elle permet de détecter que si le concept C_1 est équivalent au concept $C_2 : I(C_1) \subseteq I(C_2)$ et $I(C_2) \subseteq I(C_1)$. Par exemple, si on déclare que $Voiture \subseteq Car$ et $Car \subseteq Voiture$ revient à affirmer que $Voiture \equiv Car$. Cette propriété est automatiquement inférée et permet de détecter une duplication de l'information.
- La disjointure (TBox) : elle permet de déduire qu'un concept C_1 est disjoint d'un concept $C_2 : I(C_1) \bigcap I(C_2) = \perp$. Par exemple, si on considère que $Interrupteur \subseteq Sectionneur$, $Cable \subseteq Conducteur$ et $Sectionneur \cap Conducteur = \varnothing$, alors il est possible de démontrer que les interrupteurs et les câbles sont disjoints : $Interrupteur \cap Cable = \perp$.
- La consistance (ABox) : elle est une inférence qui permet de réaliser un raisonnement sur les individus ou les instances de l'ontologie. La consistance permet de vérifier qu'un individu appartenant à un concept C_1 n'est pas en contradiction avec la définition de la classe. Admettons qu'une instance appartient aux deux concepts $\{I_1 \in Cable;\ I_1 \in Conducteur\}$. Il en résulte une inconsistance, car les deux classes sont disjointes telles qu'exposées dans le paragraphe précédent.
- La classification (ABox) : elle détermine si un concept appartient à une instance donnée. Selon les relations de l'instance, le raisonneur est en mesure d'en déterminer l'appartenance à un ou plusieurs concept. Formellement, si on reprend la définition $Interrupteur \subseteq Sectionneur$, il est possible de déduire que pour toutes les insertions $I_1 \in Interrupteur$, alors $I_1 \in Sectionneur$ est inférée.

Le pouvoir d'inférence des ontologies est une fonctionnalité puissante. Les règles logiques permettent au moteur d'inférence de détecter des inconsistances et d'inférer de nouvelles connaissances. Toutefois, le processus de raisonnement peut nécessiter beaucoup de temps car la complexité du modèle OWL2-RL est fonction du volume de données à traiter. La complexité est estimée à n^c avec n représentant le volume des données et c une constante à déterminer d'après le (W3).

En regardant de plus près les caractéristiques de l'ontologie, il est possible de la faire correspondre à une base de données relationnelle. Les concepts et les individus d'une ontologie représentent respectivement les tables et les enregistrements d'une base de données. Dans une ontologie, un concept est un contenant qui renferme des instances.

Toutefois, ce qui caractérise une ontologie est qu'elle dispose d'un pouvoir d'inférence.

L'ontologie utilise des connecteurs logiques (universel, existentiel, union, intersection) ainsi que des propriétés prédéfinies (domaine, image, transitivité, symétrie, réflexivité) qui fournissent un support sémantique aux données. Cependant, l'analogie s'arrête là, car l'ontologie ne dispose pas de mécanisme robuste pour la manipulation des données (écriture et lecture), elle demeure un support pour représenter l'information et non pour gérer l'information.

3.3 Les connaissances du domaine électrique

3.3.1 Le standard CIM

L'intérêt que présente une ontologie réside dans sa capacité à donner un sens logique aux données. Cette sémantique offre la capacité de raisonner sur les données pour soit en vérifier la consistance, soit inférer de nouvelles connaissances. Ces deux caractéristiques présentent un intérêt pour le système Leopar++ qui est appelé a extraire et à fusionner des données de différentes sources.

La construction d'une ontologie est une opération complexe qui nécessite l'intervention d'experts du domaine et de la représentation des connaissances. Pour développer une ontologie, qui va jouer le rôle de support informationnel pour le futur système de simulation Leopar++, il y a lieu de respecter trois étapes essentielles (voir Uschold et King, 1995; Pinto et Martins, 2004) :

1. L'identification du domaine d'application de l'ontologie afin de déterminer l'étendue du domaine et sa granularité.
2. La construction de l'ontologie à travers l'acquisition des connaissances du domaine, la conceptualisation et l'intégration d'ontologies existantes. Cette phase est cruciale car elle permet d'accélérer le développement de l'ontologie.
3. L'évaluation de l'ontologie et sa validation. Cette étape nécessite l'intervention d'un expert du domaine pour valider l'ontologie nouvellement créée.

En ce qui concerne les deux premiers points, nous avons entrepris une étude minutieuse du domaine électrique et avons identifié le modèle CIM (Common Information Model). Ce dernier est un standard pour l'échange de données électriques. Il constitue une excellente référence pour le développement d'ontologies du domaine électrique. Le modèle CIM est né du besoin d'échanger de l'information entre les compagnies d'électricité via l'utilisation d'un même langage. Les difficultés d'importation et d'exportation des données ont abouti à l'élaboration du standard CIM. Il est un format d'échange des données du domaine électrique.

Le CIM est basé sur les standards IEC 61970 et IEC 61968 pour permettre l'interopérabilité des données. Il offre une structure conceptuelle hiérarchique détaillée pour représenter des composants électriques et leurs caractéristiques. La spécification du modèle CIM utilise le modèle conceptuel UML (Unified Modeling Language), qui renferme des centaines de classes regroupées selon 12 catégories (voir Figure 3.3).

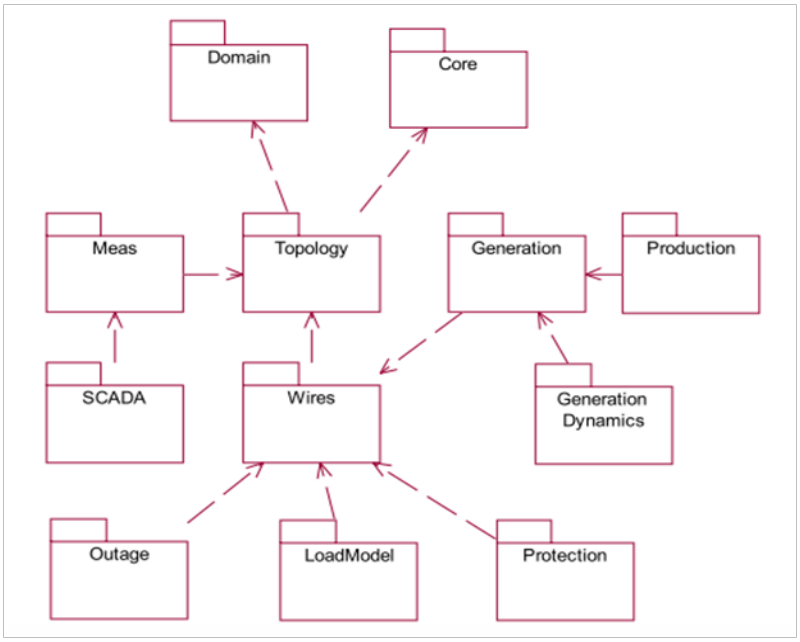


Figure 3.3 Les catégories du CIM

Chacune de ces catégories renferme des centaines de classes relatives aux câbles, aux charges, aux systèmes de production, aux éléments de mesures, aux profils de charges, etc. Les classes comportent des liens hiérarchiques, des attributs, des méthodes ainsi que des relations de compositions et agrégations. Le langage CIM est réaprti dans les groupes suivants :

- Meas : identifie des types de mesures, des valeurs de mesures, des accumulateurs de mesure, des mesures analogiques, etc.
- Wires : identifie des disjoncteurs, des lignes, des impédances, des transformateurs, des interrupteurs, des sectionneurs, etc.
- Core : définit le voltage de base, le nœud de connectivité, la courbe de données, etc.
- LoadModel : définit un modèle de jours, des modèles de charges, des saisons et des groupes de charges.
- Production : définit le type de combustible des systèmes à combustion, les caractéristiques des machines synchrones, des turbines, des hydro pompes, etc.

Le modèle CIM étant originellement représenté sous un formalisme de diagrammes de classes UML, il en résulte que l'expressivité du modèle englobe les concepts suivants :

- Classes : représentent une abstraction du monde réel.
- Attributs : représentent les données de la classe.
- Méthodes : représentent les traitements de la classe.
- Agrégation/Composition : représentent les relations de dépendance entre les classes. Cette relation comporte des cardinalités.
- Héritage : représentent les relations d'héritages entre les classes.

Ces caractéristiques inhérentes à UML limitent l'expressivité des données représentées et offrent un faible pouvoir d'inférence. C'est dans cette optique qu'une première traduction du CIM en modèle OWL a été réalisée par le CIMTool Group. Cette première traduction du modèle en OWL offre une plus grande souplesse que la représentation traditionnelle faite en UML/XML. À l'aide de la représentation OWL des composants électriques, il devient possible de :

- Communiquer plus efficacement entre les systèmes informatiques. Les données échangées sont sous le format de triplets octroyant une grande souplesse quant à la manipulation des données par rapport au formalisme XML.
- Fédérer différentes sources de données hétérogènes en un seul formalisme à l'aide de la puissance du web sémantique sur lequel se base le modèle OWL. Il est possible d'intégrer et de fédérer des formalismes ou standards comme la géoposition Open GIS, les données issues de progiciels OAGIS, de standards d'interface MultiSpeak, de modèles de données ouvert de maintenance MIMOSA, de standards du modèle objet OMG, etc. (voir Figure 3.4).
- Intégrer dans un parc informatique de nouveaux outils compatibles avec le modèle CIM.
- Offrir une meilleure couverture des données électriques à l'aide d'un langage qui englobe les composants électriques, les modèles de charges, les mesures, les systèmes de collectes de données, etc.

3.3.2 Le CIM à travers des exemples

Attardons-nous à analyser le modèle CIM à travers la conceptualisation d'un exemple. Cette analyse permet de déterminer les éléments réutilisables qui feront partie de la base de

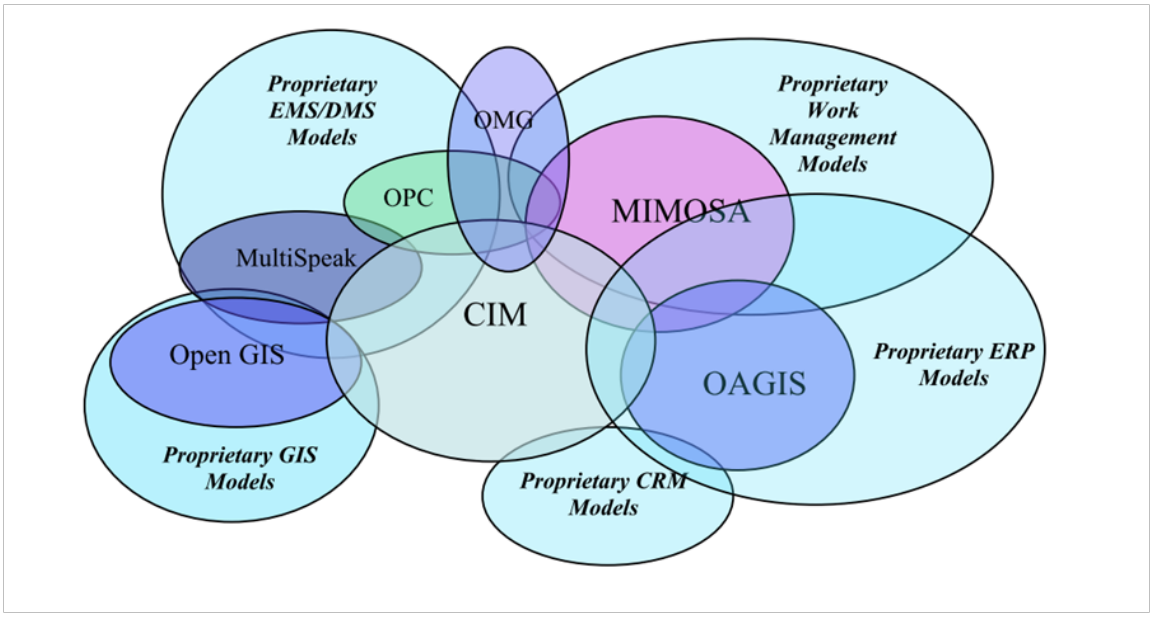


Figure 3.4 Vue globale du CIM

connaissance du simulateur Leopar++. Pour représenter une ligne électrique, le langage CIM introduit artificiellement des nœuds conceptuels pour permettre la création d'un lien entre les composants électriques. Ces nœuds sont *Terminal* et *ConnectivityNode* et ont pour tâche de connecter les composants tout en évitant les listes d'éléments. Explicitement, l'exemple suivant démontre le rôle joué par les nœuds conceptuels lors de la traduction d'un réseau électrique en modèle CIM (voir Figure 3.5) (voir McMorran, 2007).

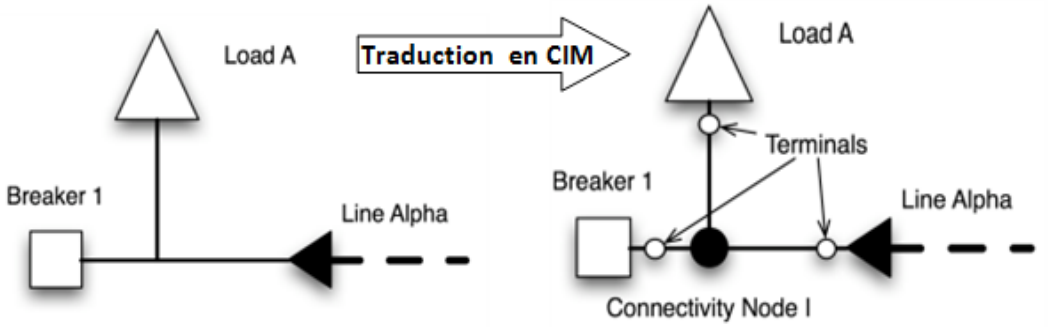


Figure 3.5 Exemple de réseau en CIM

Dans l'exemple précédent, pour connecter un disjoncteur (*Breaker*) et une charge (*LoadA*), l'insertion d'un noeud de connexion (*ConnectivityNode*) et des terminaux *Terminal* permet

de lier les composants du réseau selon l'approche CIM. Une fois que les nœuds conceptuels ont été introduits, il faut déterminer le nom réservé des composants selon la nomenclature des mots réservés CIM.

La traduction CIM/UML en CIM/OWL est une transcription directe du modèle de conception UML. Il s'ensuit que certains concepts sont inappropriés comme les nœuds conceptuels *terminals*. La traduction d'un grand réseau de distribution en CIM/OWL génère un volume de données très important qui est difficile, voir impossible à traiter par les outils de manipulation des ontologies. Utiliser le CIM/OWL pour représenter les connaissances d'Hydro-Québec devient une opération à risque qui nécessite une révision du modèle CIM.

CHAPITRE 4

LA BASE DE CONNAISSANCES DU SYSTÈME LEOPAR++

4.1 Introduction

L'objectif premier de la base de connaissances du système Leopar++ est de fusionner et rendre accessible des données réparties dans différentes sources. La base de connaissances permet, à l'aide de la logique descriptive, de vérifier l'intégrité des données fusionnées. Il devient important de mettre en place un mécanisme d'extraction, de fusion et de vérification des données. Ces données seront par la suite exploitées par les agents du système Leopar++.

C'est à ce défi que nous répondons dans la suite de ce chapitre. Nous analysons dans une première partie le processus de création de la base de connaissances. Dans une deuxième partie nous étudions le processus d'exploitation des données de la part des agents.

Dans le cadre du projet LEOPAR, la base de connaissances se compose de données provenant de différentes bases de données relationnelles. Il est primordial de mettre en place un ensemble de mécanismes pour : (1) *extraire* les données, (2) *fusionner* les données et (3) *vérifier* la consistance des données fusionnées.

4.2 L'extraction des données

Les données du réseau de distribution sont stockées dans diverses bases de données de type MySQL. Ainsi, la première étape à réaliser pour produire une base de connaissances est d'extraire les données des bases de données pour former la ABox.

Nous avons testé différents outils *freeware* qui convertissent une base de données relationnelle en base de connaissances RDF ou OWL. Les principaux outils sont les suivants (voir Tableau 4.1)

Le tableau récapitulatif 4.1 indique que les outils D2RQ et Datamaster sont les seuls à supporter le langage OWL. Toutefois, seul le D2RQ est en mesure d'être déployé de manière autonome en dehors de l'environnement Protégé. Ainsi, nous avons focalisé nos efforts sur D2RQ pour extraire des bases de données d'Hydro-Québec les données nécessaires à l'exécution de la simulation électrique. Une sous section de l'architecture D2RQ se présente comme suit (voir Figure 4.1).

Le module *D2RQ Engine* offre un accès à un contenu OWL via l'API Jena. Il établit la correspondance entre les bases de données relationnelles et les triplets RDF/OWL à travers

Tableau 4.1 Comparaisons d'outils de convertisseur BD en base de connaissances

Outils	RDFs	OWL	Environnement	Remarques
Spyder/Spinner	X		Java	Utilise le langage D2RML pour lier les bases de données au modèle RDF.
DataMaster	X	X	Protégé v3.X	Est une extension pour l'outil Protégé. Il permet d'extraire les enregistrements des bases de données.
D2RQ	X	X	Java	Utilise un langage propriétaire pour établir la relation entre la base de données et des données RDF/OWL.

un fichier nommé *D2RQ mapping File*. Il informe des relations entre les tables de la base de données et les instances de classe de la TBox.

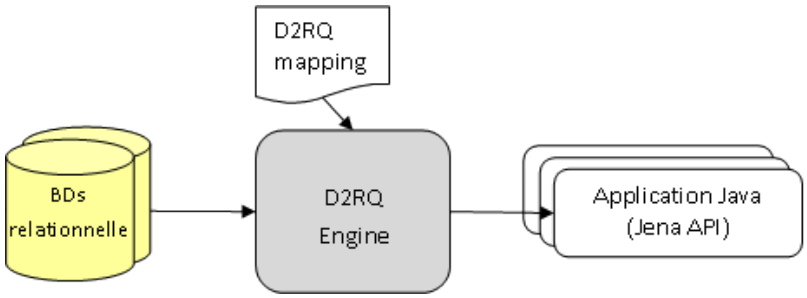


Figure 4.1 Architecture de D2RQ

Pour le système Leopar++, nous avons défini le fichier de mappage de manière à établir un lien entre les bases de données et les instances des classes du CIM. Nous exposons une sous-section du fichier de mappage afin de saisir brièvement la manière dont s'opère le mappage entre bases de données et instances de la base de connaissances de la ABox :

La Figure 4.2 établit les liens entre les bases de données et les instances de concepts. Ce fichier exprime un lien entre la table *fiche_10_poste* de la base de données et le concept *Poste* de la base de connaissances. Chaque enregistrement de la colonne *sigle_poste* de la table *fiche_10_poste* de la base de données d'Hydro-Québec représente un individu du concept *Poste* de la base de connaissances.

```
map:Poste a d2rq:ClassMap;
     d2rq:dataStorage map:database;
     d2rq:uriPattern "poste/@@fiche_10_poste.sigle_poste|urlify@@";
     d2rq:class ird:Poste;
     .
# Attributs
map:Poste_code_municipalite a d2rq:PropertyBridge;
     d2rq:belongsToClassMap map:Poste;
     d2rq:property ird:code_municipalite;
     d2rq:column "fiche_10_poste.code_municipalite"
     .
```

Figure 4.2 Exemple de fichier de mappage

En ce qui concerne les attributs, nous constatons qu'une instance du concept *Poste* a une propriété qui la lie à une municipalité. Cette dernière est une information issue de la colonne *code_municipalité* de la table *fiche_10_poste* de la base de données.

Pour établir un lien de relation entre deux individus de concepts différents, il faut utiliser l'identifiant unique du concept du poste *map* : *Poste*. La figure 4.3 expose une sous-section du fichier de mappage. Cet exemple indique que *Ligne_poste* est une relation entre la classe *Ligne* et la classe *Poste*. Pour chaque instance de la table *fiche_20_ligne* est associée une relation *Ligne_poste* qui pointe vers la colonne *poste* de la table *fiche_10_poste* :

```
# Relations
map:Ligne_poste a d2rq:PropertyBridge;
     d2rq:belongsToClassMap map:Ligne;
     d2rq:property ird:Ligne.poste;
     d2rq:refersToClassMap map:Poste;
     d2rq:join "fiche_20_ligne.responsabilite = fiche_10_poste.responsabilite";
     d2rq:join "fiche_20_ligne.sigle_poste = fiche_10_poste.sigle_poste"
     .
```

Figure 4.3 Exemple de relation de mappage

Une présentation sommaire du mode de fonctionnement de D2RQ fait ressortir deux constats. Premièrement, D2RQ permet de créer des instances à partir des bases de données. Deuxièmement, D2RQ ne permet pas définir une TBOX et ainsi disposer d'une base de connaissances complète. De ce fait, il est impossible de raisonner sur la base de connaissances,

car D2RQ ne génère que la ABox avec aucune sémantique liée aux concepts. La ABox est générée automatiquement par D2RQ en lui rajoutant une TBox pour pouvoir raisonner sur la base de connaissances fédérée.

4.3 La fusion

La figure (voir Figure 4.4) expose une première section de l'architecture que nous avons mise en place pour extraire les données et former la ABox. Néanmoins, il faut compléter la base avec les concepts de la TBox.

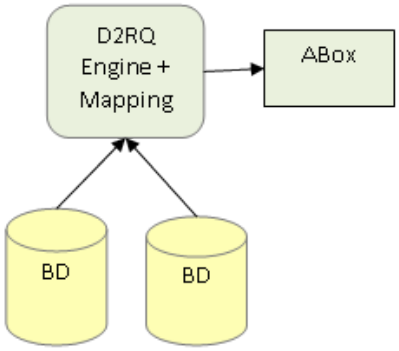


Figure 4.4 Mécanisme d'extraction des données de Leopar++

Il est primordial de fusionner automatiquement les concepts aux instances, et ce afin de disposer d'une base de connaissances où il est possible de vérifier la consistance des données fédérées. Pour ce faire, nous utilisons l'API Jena, élément central de notre architecture, pour fusionner la ABox et la TBox.

Nous utilisons le standard CIM comme TBox vu l'importance qu'il revêt au sein de la communauté. Or, une étude détaillée de ce standard nous a permis d'identifier deux limitations majeures empêchant l'utilisation directe du CIM comme TBox. Ces limitations sont les suivantes :

- Certains concepts sont superficiels. Les nœuds conceptuels *Terminal* augmentent artificiellement la base de connaissances. À l'origine, ces concepts étaient introduits pour éviter les structures de liste.
- Certains concepts sont trop détaillés. Les transformateurs électriques nécessitent, en CIM, trois concepts différents pour être représentés. Or certains concepts peuvent être fusionnés en un seul concept global.

– Certains concepts agissent comme des propriétés. Le concept *BaseLineVoltage* du CIM est un concept qui permet de spécifier la tension des composants. Il serait plus avantageux de le remplacer par des propriétés de concepts.

Nous en concluons que l'utilisation du CIM comme modèle conceptuel pour définir la *TBox* est sous-optimal, car le langage est trop détaillé et sa sémantique est limitée. La conversion d'un réseau de distribution en modèle CIM crée un volume de données important et rend difficile la manipulation de la base de connaissances. Le langage CIM/OWL augmente artificiellement la quantité des données formalisées. Nous avons opéré les changements suivants sur le modèle CIM/OWL [1] :

Suppression du concept terminal :

Nous supprimons le concept abstrait *Terminal*. Ce dernier a été introduit dans le modèle CIM pour éviter des structures de listes lors de la connexion des ($ConductingEquipment$). Or, il augmente artificiellement le volume des données et son utilité est limitée. Nous pensons que la suppression du composant *terminal* du modèle CIM est une opération avantageuse qui permet de réduire considérablement le volume des données générées. La logique descriptive suivante présente la différence entre le modèle original CIM (Version 0) et le modèle modifié que nous proposons (Version 1). Nous accompagnons cette description par un graphique démonstratif.

Version 0 (CIM original) :

$ConnectivityNode \subseteq IdentifiedObject \cap \forall.hasTerminal.Terminal \cap$

$1.hasContainer.ConnectivityNodeContainer.$

$Terminal \subseteq IdentifiedObject \cap \forall.hasRegulatingControl.RegulatingControl \cap$

$1.hasConnectivityNode.ConnectivityNode \cap$
$1.hasConductingEquipment.ConductingEquipment.$

$ConductingEquipment \subseteq Equipment \cap \leqslant 1.hasBaseVoltage.BaseVoltage \cap$

$\forall.hasTerminal.Terminal.$

Version 1 (CIM modifié) :

$ConnectivityNode \subseteq IdentifiedObject \cap$

1. Nous exposons uniquement les relations entre classes et non pas les relations de propriétés.

$$\geqslant 1.hasConductingEquipment.ConductingEquipment \cap$$
$$1.hasContainer.ConnectivityNodeContainer \cap$$
$$\forall.hasRegulatingControl.RegulatingControl.$$

$ConductingEquipment \subseteq Equipment \cap \leqslant 1.hasBaseVoltage.BaseVoltage.$

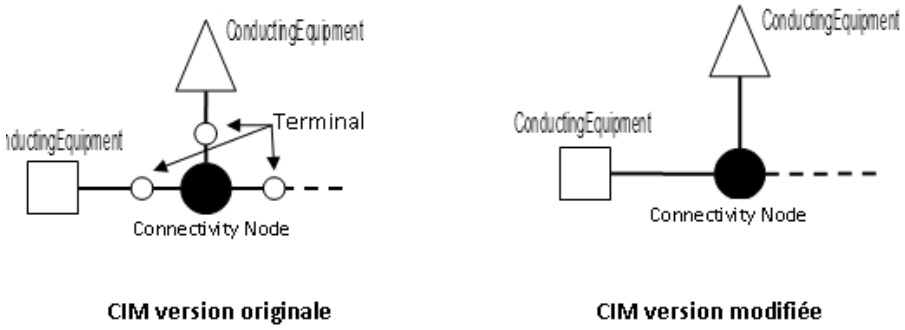


Figure 4.5 Exemple de simplification du CIM

Fusion des concepts PowerTransformer :

Durant cette étape nous fusionnons toutes les propriétés du *TransformerWinding* dans le concept *PowerTransformer*. Cette opération permet d'avoir moins de concepts et par conséquent diminuer le volume des données. Dans la version originale du CIM, les enroulement ($TransformerWinding$) primaires et secondaires du transformateur sont des concepts à part entière. D'après nous, ces concepts représentent des propriétés du transformateur et peuvent être fusionnés dans le concept $PowerTransformer$. Cette fusion permet de réduire le nombre de concepts et les relations les liants. Analysons ensemble comment nous arrivons à intégrer les concepts $TransformerWinding$ dans le concept $PowerTransformer$.

Version 0 (CIM original) :
$PowerTransformer \subseteq Equipment \cap \forall.hasWinding.TransformerWinding.$

$TransformerWinding \subseteq ConductingEquipment \cap$

$$1.hasPowerTransformer.PowerTransformer \cap$$
$$\forall.hasTapChanger.TapChanger\ .$$

Version 1 (CIM modifié) :
$PowerTransformer \subseteq ConductingEquipment \cap \leqslant \forall.hasTapChanger.TapChanger \cap$

$1.hasPowerTransformer.PowerTransformer.$

Transformation du concept BaseVoltage :

Nous fusionnons les propriétés du concept *BaseVoltage* avec *ConductingEquipment*. À l'image du $PowerTransformer$, nous estimons que le concept $BaseVoltage$ du modèle original CIM revêt le rôle de propriété. Ainsi, nous fusionnons les propriétés du $BaseVoltage$ au concept $ConductingEquipment$ pour compresser le volume des données.

Version 0 (CIM original) :

$ConductingEquipment \subseteq Equipment \cap \leqslant 1.hasBaseVoltage.BaseVoltage \cap$

$\forall.hasTerminal.Terminal.$

$BaseVoltage \subseteq IdentifiedObject \cap$

$\forall.hasConductingEquipment.ConductingEquipment \cap$
$\forall.hasVoltageLevel.VoltageLevel.$

Version 1 (CIM modifié) :

$ConductingEquipment \subseteq Equipment \cap$

$\geqslant 1.hasConnectivityNode.ConnectivityNode \cap$
$\forall.hasVoltageLevel.VoltageLevel$.

Notez que le concept du $ConductingEquipment$ renferme dorénavant toutes les propriétés des concepts $BaseVoltage$ *et* $VoltageLevel$. Bien que ces modifications semblent minimes, nous avons obtenu une base de connaissances moins volumineuse que la version originale définit par le *CIMTool group*. Le tableau suivant (voir Tableau 4.2) reprend les modifications apportées et compare la version originale et modifiée du modèle OWL/CIM pour un réseau de distribution donné.

Avec ces modifications, nous obtenons une base de connaissances moins volumineuse tout en gardant le même niveau d'information. Cette diminution s'explique par la suppression et/ou la fusion de certains concepts et propriétés. Notez que le pourcentage de compression total n'est pas cumulatif car les modifications apportées sont inter-dépendantes. Par exemple la fusion des *PowerTransformer* fait diminuer certains *Terminals*. L'étape de compression et d'optimisation de la base de connaissances est vitale, car le test de consistance de la base est tributaire du volume de données à traiter. Plus la base de connaissances est volumineuse, plus le mécanisme d'inférence est couteux en temps de traitement.

Tableau 4.2 Comparaison des versions du CIM/OWL

Tâche	CIM Version 0	CIM Version 1	Réduction
Fusion du PowerTransformer	36.5MO	24.6	32%
Transformation du BaseVoltage	36.5MO	31.9MO	12%
Suppression du Terminal	36.5MO	23.4MO	35%
Total	36.5MO	12.2MO	66%

Les modifications apportés au CIM ont nécessité une excellente compréhension du modèle, des concepts et de leurs relations. L'optimisation que nous apportons au modèle CIM/OWL est nouvelle en son genre car elle permet une représentation plus en accord avec la logique descriptive. La nouvelle version optimisée du CIM fourni un excellent modèle de représentation des connaissances de notre système de simulation Leopar++. Bien que les modifications apportées paraissent élémentaires, elles permettent de diminuer considérablement le volume des données et ouvre la voie à l'utilisation effective du CIM/OWL comme modèle de représentation des connaissances. Notre approche est novatrice dans le domaine électrique car elle combine différentes technologies et mécanismes pour extraire, fusionner et valider une base de connaissances électriques. Nous affirmons que moyennant quelques modifications mineurs du modèle CIM, celui-ci devient optimisé pour la représentation et le traitement des connaissances électriques.

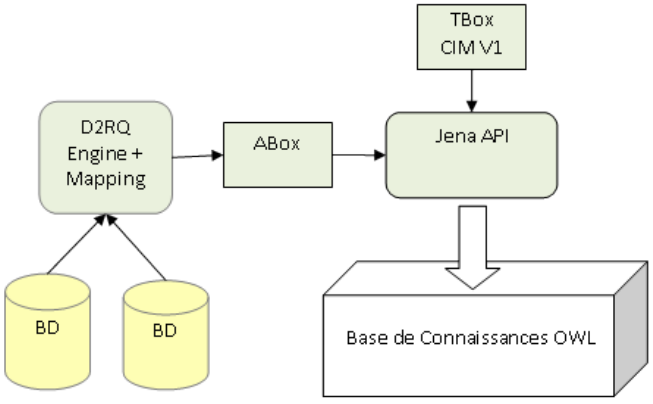


Figure 4.6 Architecture de la base de connaissances de Leopar++

Ainsi, la nouvelle version optimisée du CIM forme dorénavant la TBox de notre base de connaissances. La figure 4.6 expose comment Leopar++ fédère les instances de la ABox préalablement formées à l'aide de D2RQ et la TBox en une seule base de connaissances. C'est

à l'aide de l'API Jena que nous réalisons la fusion entre les concepts et les instances.

4.4 La vérification de la base de connaissances OWL

Une fois la base de connaissances créée, une dernière étape de vérification est réalisée. En effet, c'est à l'aide de la sémantique des concepts qu'il est possible de raisonner sur les données et de déterminer la subsomption, la satisfaisabilité, l'équivalence, la disjointure, la consistance de la base de connaissances. Cette étape est primordiale, car la fusion de données est source d'inconsistance.

Le recours aux raisonneurs permet de vérifier la consistance des données. Nous vérifions la consistance des données fédérées à l'aide du raisonneur Pellet parfaitement intégré à Jena. Pellet est un raisonneur incrémental capable de détecter les incohérences d'une base ainsi que d'extraire de nouvelles connaissances à partir de base de connaissances. Dans le cas de Leopar++, nous avons été en mesure de détecter certaines incohérences :

- L'identifiant unique des composants électriques fait parfois référence à deux ou plusieurs composants.
- Tous les réseaux ne sont pas liés à un poste.
- Tous les réseaux n'ont pas un disjoncteur principal.
- Il arrive qu'un composant électrique ne soit pas lié à un type connus, une adresse civique, ou à aucun disjoncteur en amont.
- Il arrive qu'une ligne ait une tension de 0.0 volts.

Ces erreurs ont été détectés grâce à l'utilisation de la version optimisée du CIM/OWL. Des tests de consistance ont été effectués avec la version originale du CIM et s'il avéré que pour certaine réseaux volumineux (composé de 1022 composants), le temps d'exécution de Pellet a augmenté de 400

4.5 Distribution de la base de connaissances de Leopar++

La base de connaissances nouvellement formée renferme les données fédérées de différentes sources de données. Le rôle final de celle-ci est d'être exploitée par les agents du système Leopar++. Toutefois, la base de connaissances de OWL n'est pas directement utilisable par les agents. Une analyse détaillée de OWL relève les limites majeures suivantes empêchant l'utilisation de la base par des agents :

- Les agents sont appelés à interagir avec la base de connaissances OWL. Le nombre d'agents pouvant être élevé, l'accès multiple à la base de connaissances pose problème et aboutit inévitablement à la création de goulots d'étranglement.
- Les agents pouvant s'exécuter en parallèle, il devient nécessaire de disposer d'un mécanisme d'accès concurrentiel aux données. Permettre à des agents d'accéder à des données évitant l'inter-blocage est vital.
- Les agents de notre système peuvent être distribués sur différentes machines. Il devient important de disposer de base de connaissances distribuée tout en évitant la duplication inutile des données.

Pour ce faire, nous avons décidé de convertir automatiquement une ontologie en Objets Java déployés sous l'environnement JavaSpaces. La génération de code Java à partir d'ontologie n'est pas nouvelle et il existe des outils qui permettent de convertir automatiquement des ontologies. Le tableau suivant 4.3 reprend ces outils et en résume les caractéristiques :

Tableau 4.3 Comparaison d'outils de conversion de bases de connaissances en objets Java

Outils	Environnement	Remarque
OntoJava	Protégé	Génère des objets Java à partir d'ontologie et de règle RuleML.
Bean Generator	Protégé	Permet de générer des objets compatibles sous le format de JavaBeans à partir d'une ontologie.
Ontology Creator	Protégé	Génère des classes Java à partir d'une ontologie.
Jastor	Java	Génère des classes à partir d'une ontologie. Les instances ne sont pas prises en compte.

La majorité de ces outils sont destinés à être utilisés avec l'environnement Protégé et ne sont pas autonomes. L'outil Jastor est le seul à pouvoir s'exécuter de manière autonome. Cependant il souffre de deux limitations :

- L'outil Jastor ne traduit pas des instances de la ABox,
- Le code objet généré n'est pas personnalisable en ce qui concerne les types, les méthodes, les variables globales et les contraintes.

Pour cette raison nous avons développé notre propre API pour traduire à la fois les concepts et les instances de l'ontologie. Nous nous appuyons sur les méthodes proposées

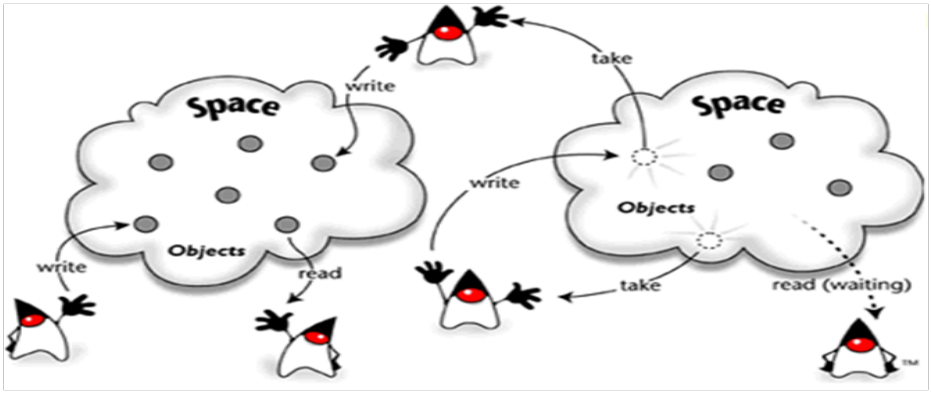


Figure 4.7 Architecture du JavaSpace

en (voir Kalyanpur *et al.*, 2004) pour la traduction d'une ontologie en objets Java. Notre traduction est plus simple que celle proposée dans Jastor car notre conversion ne tient pas compte des contraintes logiques des propriétés. Les objets Java générés vont uniquement servir comme support pour stoker les données. À titre d'information nous exposons dans la figure 4.8 la conversion de certains concepts.

Pour rendre accessible les objets Java nouvellement formés à partir de la base de connaissances, nous utilisons une technologie particulière qui se nomme JavaSpaces (voir Mamoud, 2005). C'est une implémentation de haut niveau pour le partage de données afin de les rendre accessibles à des programmes distribués. L'approche JavaSpaces a pour but de faciliter le partage de données entre applications résidant dans des environnements hétérogènes. En effet, JavaSpaces utilise un protocole d'accès aux données spécifique afin d'éviter les inter-blocages et préserve la consistance des données. Cette technologie permet de sauvegarder des données dans un espace commun accessible par des applications à distance. La figure 4.7 expose l'architecture d'une zone JavaSpaces représentant des applications Java accédant à des données sauvegardées dans JavaSpaces.

Le mécanisme JavaSpace tire ses origines d'un langage de coordination LINDA (voir Bjornson *et al.*, 1997) et présente l'avantage de :

- Offrir une plateforme qui simplifie la conception et l'implémentation de systèmes distribués.
- Permettre aux agents d'accéder à une zone commune de données distribuées pour enregistrer des données ou pour communiquer.
- Préserver la consistance des données lors de l'accès concurrent aux données grâce à un langage spécifique nommé LINDA.

Le langage LINDA est destiné à la programmation parallèle et introduit des opérations de synchronisation dans un espace de données. Le modèle LINDA est bâti sur les concepts de tuples qui sont une structure de données renfermant un identifiant unique et des données typées. Les tuples représentent des objets Java et forment un espace de tuples. La manipulation des tuples dans l'espace de tuples se fait par l'une des instructions suivantes en conformité avec le langage LINDA :

- *Write()* : permet d'introduire un nouvel objet Java dans l'espace de tuples.
- *Take()* : permet d'extraire des objets de l'espace.
- *Read()* : permet de lire le contenu de l'objet.
- *Notify()* : permet de surveiller l'espace de tuples afin de détecter une donnée particulière.

Grâce à ces instructions, il est possible de sauvegarder des données distribuées dans des espaces de tuples. Ces données sont passives et les agents ne sont pas en mesure de modifier l'objet directement dans la zone d'espace qu'après avoir retiré et inséré de nouveau l'objet.

Les espaces de tuples permettent à plusieurs agents Java d'accéder simultanément à un ou plusieurs espaces de tuples et ce de manière transparente à l'agent. Cette caractéristique permet de créer une zone commune de partage de données distribuées qui permet de garantir le transfert, la persistance, la communication et la synchronisation entre agents.

Pour conclure, nous avons été en mesure de combiner et d'optimiser des technologies différentes afin de développer une base de connaissances électrique. Nous apportons la preuve qu'il est possible d'extraire, de fusionner, de vérifier et de rendre accessible des connaissances électriques volumineuses à des fin de traitement. Ce présent chapitre décrit une approche nouvelle pour créer une base de connaissances électriques et la rendre disponible aux systèmes de simulation.

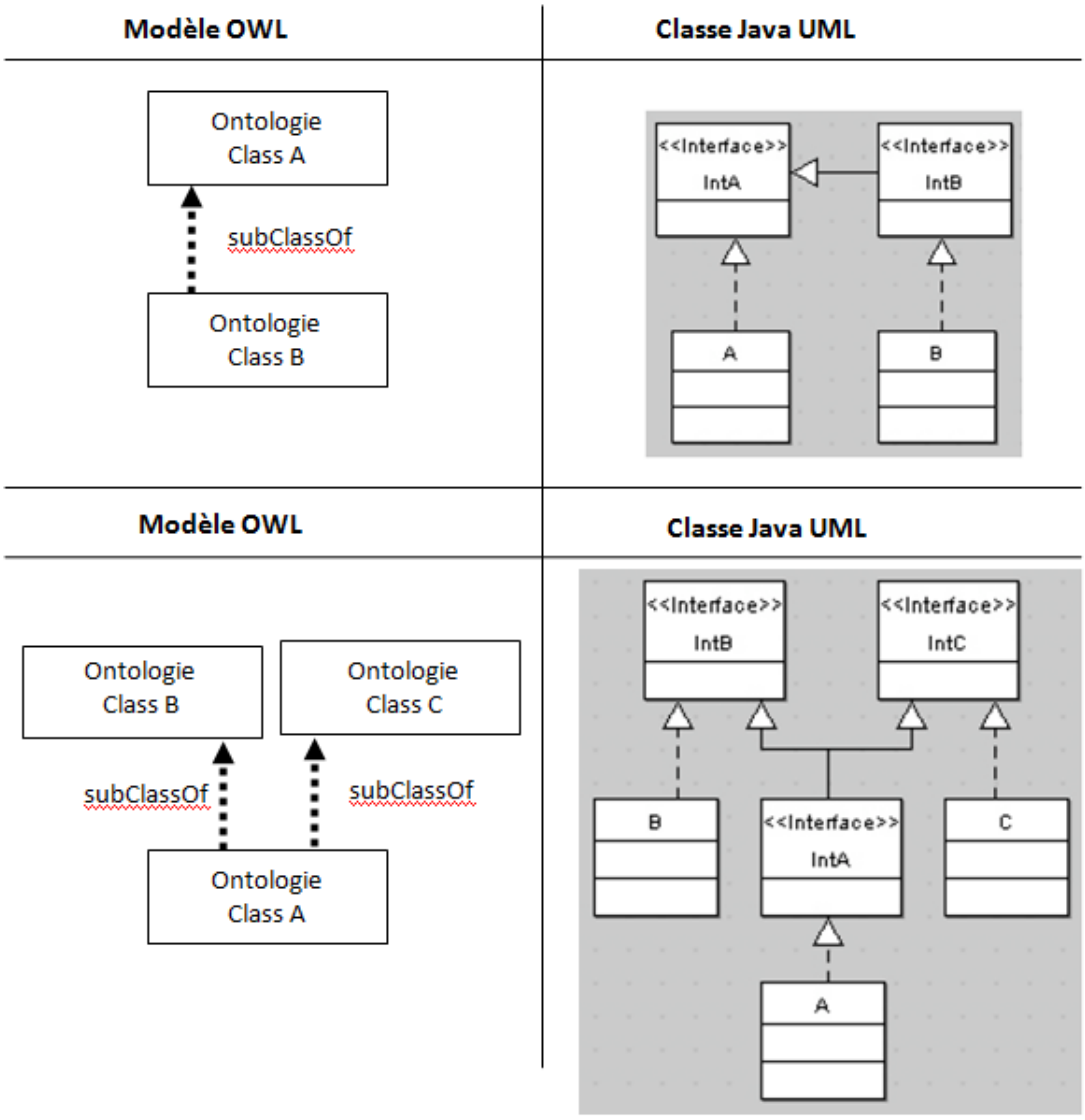


Figure 4.8 Exemple de conversion OWL en Java

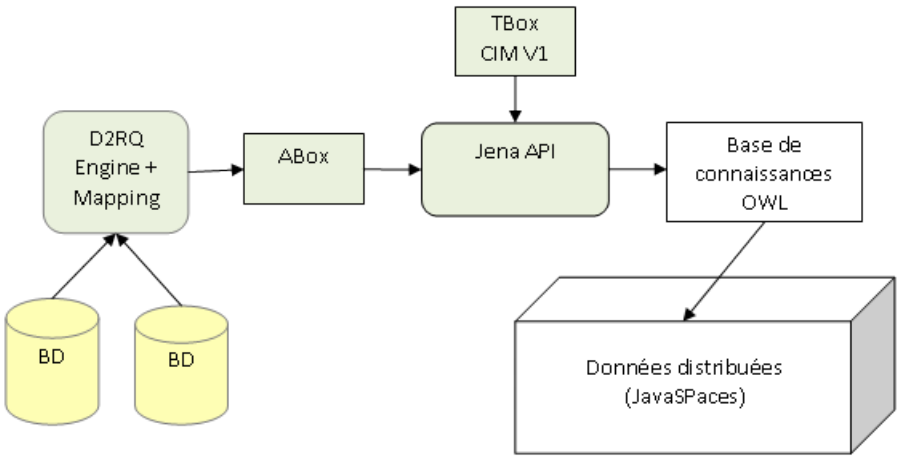


Figure 4.9 Architecture de la zone de données partagées

CHAPITRE 5

LOGIQUE NON MONOTONE ET LANGAGES D'ACTIONS

5.1 Introduction

Bien que l'ATN offre une extensibilité et accessibilité accrue, il existe certaines limites à cette approche. En effet, les ATN ont été délaissés au profit des systèmes multiagents pour plusieurs raisons (voir Hunt, 2002) :

- La zone commune d'échange de donnée représente un point d'accès unique aux KS du système. Le partage de cette zone entre plusieurs agents KS est problématique car il a risque d'accès concurrent aux ressources.
- Le processus de coopération est complexe et la détermination du KS à exécuter est une opération hautement coûteuse en temps de calcul. Les ATN souffrent d'un manque de performance face aux systèmes multiagents classiques, car le module de coordination n'est pas spécialisé dans la résolution d'un objectif particulier. En l'absence de processus de coordination efficace qui guide la résolution du problème, les ATN explorent des hypothèses inutiles qui ne contribuent pas à la résolution du problème global.
- La zone commune de partage de donnée des ATN est une zone de données globale qui renferme toutes les données relatives au problème : les hypothèses, les données, les décisions et les requêtes. Les agents KS sont programmés de manière à répondre à des modèles d'information particuliers. Dans le cas d'une exception ou un dysfonctionnement des agents KS, il arrive que les données du tableau noir deviennent chaotiques et rendent le système incontrôlable. Les ATN sont très sensibles aux erreurs dues au manque de coordination et de planifications des agents.

D'un autre côté, les systèmes multiagents offrent une approche différente des ATN en octroyant aux agents une grande autonomie. Le problème de ces systèmes se résument comme suit :

- Il est complexe de guider le système vers la résolution d'un objectif global. Parce que dans les systèmes multiagents chaque agent dispose d'objectifs propres. Il arrive que le système se retrouve dans une configuration d'inter-blocages. Il y a lieu de bien identifier l'objectif de chaque agent pour permettre au système de converger vers la résolution

d'un problème.

- Les SMA ne sont pas accessibles aux non spécialistes. En effet, les langages orientés agents nécessitent une bonne connaissances de la logique de premier ordre et/ou modale pour exprimer l'état mental de l'agent. Ce dernier est appelé à raisonner sur ses connaissances de l'environnement et des autres agents pour permettre la prise de décision.
- La communication inter-agents est complexe car elle est basée sur un protocole sophistiqué. L'échange de données entre les agents nécessite de respecter un protocole de communication rigoureux où les interactions ont besoin d'être correctement identifiées, validées et représentées.

Au regard des ATN et des SMA, il est important de disposer d'un mécanisme de coordination performant capable de décrire *facilement* les interactions des agents ainsi que les objectifs à atteindre.

Le système Leopar++ utilise un coordinateur central pour planifier l'exécution des agents. Le coordinateur utilise un *langage d'actions* de haut niveau qui permet de décrire selon une approche incrémentale les actions des agents. Il est accessible à des non-spécialistes et informe sur le mode de fonctionnement des agents. Il renseigne sur les effets et les conditions d'exécution des agents et permet au coordinateur de planifier l'exécution des agents selon les objectifs définis par l'utilisateur.

Notre coordinateur se veut robuste, polyvalent et incrémental, afin de répondre aux besoins de l'IREQ. Il diffère en cela des contrôleurs des ATN et système multiagents. Il est le premier du genre à utiliser un langage d'action pour coordonner des agents. La figure 5.1 présente une vue de haut niveau du coordinateur de Leopar++.

L'architecture que nous proposons est connue sous le nom de multiagents tableau noir (voir Zhu *et al.*, 2010; Qiang et Liu, 2010; Hammami *et al.*, 2009). Elle vise à simplifier le développement des agents à l'aide d'une zone commune de données partagées pour faciliter la communication et l'interaction (voir Kao *et al.*, 2002; Dong *et al.*, 2005). Toutefois, ces systèmes ne sont pas suffisamment détaillés pour être utilisables directement dans le futur système de simulation que nous souhaitons développer. Ainsi, nous apportons deux contributions majeures à l'architecture multiagents tableau noir :

1. Un langage de coordination spécifique aux systèmes multiagents basés sur le tableau noir. Les langages d'actions n'ont jamais été utilisé sur des systèmes multiagents en général.
2. Une description détaillée de l'architecture mutliagents tableau noir à base de connais-

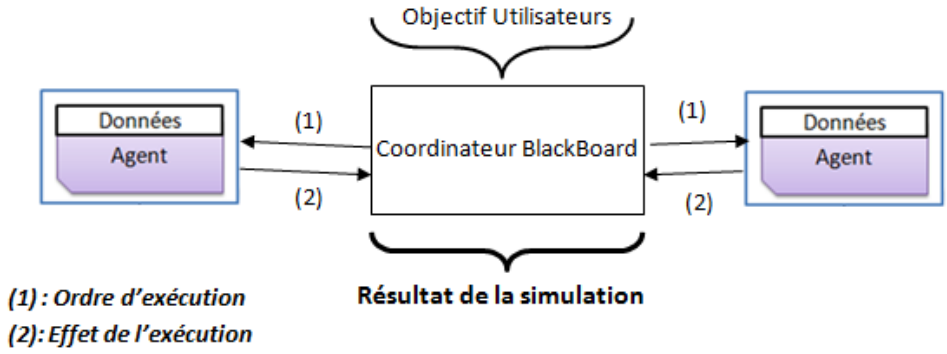


Figure 5.1 Vue de haut niveau du coordinateur Leopar++

sances.

5.2 Les langages d'actions

Pour coordonner les agents d'un système, il faut disposer d'un langage qui permet de décrire les agents et leurs comportements. Nous avons analysé les langages orientés agents dans le chapitre 3 et une première remarque est ressortie : la limite première de ces formalismes réside dans leur complexité vis-à-vis des non-experts. Comme la majorité de ces langages utilisent la logique de premier ordre ou la logique modale temporelle, il est difficile pour un non-expert d'utiliser aisément ces langages. Par ailleurs, cette complexité est d'autant plus importante que le protocole de communication, généralement basé sur FIPA, rajoute un niveau de difficulté quant à la manière dont les agents communiquent entre eux.

Or, ces dernières années, des recherches ont ouvert la voie à des langages abstraits et de haut niveau qui permettent de décrire selon une approche incrémentale les actions des agents. Les *langages d'actions* sont des langages **orientés action** et c'est en cela qu'il diffèrent des langages orientés agents, qui sont orientés implémentation.

Le langage d'actions est un modèle formel introduit dans les années 1990 par Gelfond (voir Gelfond et Lifschitz, 1993). Il traduit l'effet des actions pour un système de transition et permet de décrire des actions et leurs effets selon un formalisme facile à comprendre. De nos jours, il existe une grande variété de langages d'actions, par exemple les langages d'action A, B, C (voir Lifschitz et Turner, 1999) qui se distinguent par l'expressivité et les opérateurs permis. Ces langages sont utilisés pour la planification, la modélisation de phénomènes naturels, le diagnostic de pannes, la robotique, etc. Cet attrait est dû à l'accessibilité du langage

et à sa simplicité, comme nous allons l'exposer dans les paragraphes suivants.

5.2.1 Langage d'actions A

L'alphabet de ce langage d'actions comporte deux ensembles disjoints et non vides de symboles F et A, respectivement l'ensemble des **faits** (*fluents*) et des **actions**. Un fluent représente l'état d'un objet : par exemple, $on(blocA, table)$ permet de spécifier que l'objet *blocA* est sur la table. L'ensemble F décrit des états d'objets et plus généralement l'état de l'environnement (*state of the world).* Un fluent peut être vrai ou faux. Dans le cas où le fluent est faux il est précédé par une négation booléenne « ¬ » décrivant la non-véracité du fluent. Formellement, un fluent f est vrai à l'état σ si $f \in \sigma$. Respectivement, $\neg f$ est faux à l'état σ, si $f \notin \sigma$.

En dehors des fluents, il existe des actions qui agissent sur les fluents et modifient les états de l'environnement. Par exemple, l'action de déplacer un bloc B sur une table, $moveOn(blocB, table)$, entraine l'apparition des effets suivants $\sigma' = \{on(blocB, table)\}$. Plus généralement, un langage d'actions vise à traduire la transition entre les états. Le langage d'actions A s'appuie sur l'expression suivante pour définir le modèle de transition :

$$a \ causes \ f_1, \ldots, f_m \ if \ F.$$

Où $a \in A$ est une action et $\{f_1, \ldots, f_m\} \subseteq F$ sont des fluents. Cette expression définit que si F est vrai à un état σ, alors l'action ajoute les faits de $\{f_1, \ldots, f_m\}$ à l'état successeur σ'. Par exemple, il est possible de définir l'expression suivante selon le langage A :

$$moveOn(X, Y) \ causes \ on(X, Y), \ \neg clear(Y) \ if \ clear(X), clear(Y).$$

Cette expression spécifie quand il est possible de déplacer un bloc d'un endroit à un autre. Elle informe que pour déplacer un bloc X sur un bloc Y, ils doivent être libres : clear(X), clear(Y). Les effets de l'exécution de l'action sur l'environnement sont définis après le mot clé *causes*. L'expression informe que l'exécution de l'action $moveOn(X, Y)$ entraine que le bloc X est sur le bloc Y, $(on(X, Y))$, et que le bloc Y n'est plus libre $(\neg clear(Y))$.
Le langage d'actions A comprend une expression qui décrit l'état initial du système de transition :

$$initially(f_1, \ldots, f_m)$$

Cette expression définit les fluents qui sont vrais durant l'état initial du système. Pour revenir à notre exemple précédent, il est possible de spécifier :

$$initially(on(blocB, table)).$$
$$initially(on(blocA, table)).$$

La planification dans ce genre de langage revient à déterminer la séquence d'actions qui, à partir de l'état initial, arrive à satisfaire un ensemble de fluents F finaux.

5.2.2 Le langage d'actions B

Ce langage étend le langage d'actions A en permettant l'introduction d'effets indirects. Cette opération est permise par l'introduction de *lois statiques*. Les expressions du langage sont les suivantes :

$$causes\ f_1, \ldots, f_m\ if\ F. \qquad (Loi\ statique)$$
$$a\ causes\ f_1, \ldots, f_m\ if\ F. \qquad (Loi\ dynamique)$$
$$initially(f_1, \ldots, f_m)$$

Par rapport au langage d'actions A, le langage B permet de décrire des lois statiques qui sont indépendantes des actions. Les lois statiques stipulent que lorsque F est vrai durant un état, alors $\{f_1, \ldots, f_m\}$ sont vrais dans le même état. Les lois dynamiques informent que si F est vrai durant un état s, alors $\{f_1, \ldots, f_m\}$ sont vrais dans l'état suivant s'. La traduction en langage B de l'exemple des blocs donne :

$$moveOn(X, Y)\ causes\ on(X, Y)\ if\ clear(X), clear(Y).$$
$$causes\ \neg clear(Y)\ if\ on(X, Y).$$

Il en résulte que l'expression de l'action de déplacer un bloc est subdivisée en deux expressions composées de lois statique et dynamique. Ce partage offre une meilleure lisibilité des expressions, où il est possible de se focaliser sur les effets directs de l'action entreprise.

5.2.3 Le langage d'actions C

Le langage C est similaire au langage d'actions B, mais offre en plus la possibilité d'exprimer des actions concurrentes ou en parallèle et permet de spécifier si les lois d'inertie sont applicables ou non à un effet. Dans notre cas l'inertie se définit comme la capacité à un fluent d'être vrai jusqu'à ce qu'une action vienne modifier son état. Les expressions du langage sont :

$$causes\ G\ if\ F. \qquad (Loi\ statique)$$
$$causes\ G\ if\ F\ after\ U. \qquad (Loi\ dynamique)$$

Dans notre cas, les symboles G et F sont des fluents, tandis que U représente soit des actions soit des fluents. La loi statique exprime que si F est vrai durant un état donné s, alors les fluents G sont vrais durant l'état suivant s'. La loi dynamique exprime que si U est vrai à un état donné s et que F est vrai durant l'état suivant s' alors G devient vrai durant l'état s'. Cette loi est puissante car elle permet de généraliser les expressions du langage A et du langage B :

Pour exprimer l'effet d'une action, le langage C utilise l'expression suivante :

$$U\ causes\ \ G\ if\ F\ \equiv\ causes\ G\ if\ True\ after\ U \cap F$$

Le principe d'inertie, qui est implicite dans les langages A et B, devient une expression explicite dans le nouveau langage C :

$$inertial(F) \equiv\ causes\ F\ if\ F\ afterF$$

Il est aussi possible de rendre une action non exécutable et ainsi réduire le champ des actions possibles :

$$non\_executable(U)\ if\ F \equiv causes\ False\ after\ F \cap U$$

Il est aussi permis d'introduire l'incertitude de l'exécution d'une action. L'expression suivante stipule qu'il n'est pas certain que U s'exécute dans le cas où F est vrai, avec F est un ensemble de fluents :

$$U\ may\ cause\ F\ if\ G \equiv\ causes\ F\ if\ F\ after\ G \cap U$$

En dehors de ces langages dits classiques, il existe plusieurs autres versions de langages qui introduisent des propriétés spécifiques. Nous citons le langage C_k (voir Eiter *et al.*, 2004) qui introduit des caractéristiques d'incertitude sur les fluents. Il existe aussi le langage C_{TAID} qui fait référence à des phénomènes biologiques l'inhibition, l'activation, la permissivité d'une action.

5.3 Le formalisme ASP

Les langages d'actions offrent un excellent moyen pour décrire les actions d'un agent dans le système Leopar++. Il permettent de coordonner les agents en se basant uniquement sur les effets de l'action et sont de ce fait faciles d'utilisation pour des non experts. Ils offrent un excellent support pour décrire et planifier des actions des agents.

Toutefois, les langages d'actions sont des langages de modélisation. Ils ne disposent pas de pouvoir d'inférence. Ils nécessitent une traduction en un programme informatique pour permettre le traitement et la planification des actions. Or, les langage d'actions sont traduisibles en logique non monotone à des fins de planification. Parmi les logiques non monotones, la programmation déclarative ASP (Answer Set Programming) offre un langage puissant et expressif. Analysons le mode de fonctionnement d'ASP, car nous allons avoir recours à ce formalisme dans la suite du document.

Le langage ASP est un formalisme déclaratif destiné à la représentation de problèmes NP difficiles (voir Eiter *et al.*, 2004; Alferes et Pereira, 1996). Le formalisme ASP utilise le raisonnement non monotone, qui est une famille de logique particulière. Ce type de formalisme introduit le principe de raisonnement par défaut à l'aide d'un opérateur particulier, appelé **négation par défaut**. Cette propriété est différente de la négation classique de la logique de proposition. La négation par défaut symbolise l'absence d'information tandis que la négation classique informe sur la fausseté de l'information. Une autre différence majeure qui distingue la logique classique de la logique non monotone est l'introduction de la disjointure. Ces ajouts permettent au formalisme ASP de décrire des phénomènes plus riches, tels que les effets d'une action, l'incomplétude d'une information, les exceptions, la non-monotonie, l'incertitude de l'effet d'une action, etc. (voir Gelfond, 2007).

Ainsi, pour pouvoir utiliser un langage d'actions dans le cadre du projet LEOPAR, il est important de comprendre le langage ASP. Ce dernier va permettre de rendre le langage d'actions décidable. Analysons de plus la logique non monotone ASP pour en saisir le mode de fonctionnement.

5.3.1 Syntaxe du langage ASP

Soit trois ensembles disjoints $\sigma^{pred}, \sigma^{cte}, \sigma^{var}$ de prédicats, constantes et variables d'un vocabulaire θ. L'ensemble des variables est infini et les ensembles de constantes et prédicats sont finis. Nous posons comme norme que chaque item de l'ensemble σ^{pred} et σ^{cte} sont sous la forme d'une chaîne de caractères qui commence par une minuscule. L'ensemble σ^{cte} représente soit une valeur numérique ou une chaine de caractères. L'ensemble σ^{var} des variables est représenté à l'aide d'une chaine de caractères qui commence par une majuscule.

Supposons qu'un atome se définit comme étant une expression de la forme $p(t_1, \ldots, t_k)$ avec k représente l'arité de l'atome. Le symbole p est un prédicat tel que $p \in \sigma^{pred}$. Les $(t_1, \ldots, t_k)$ représentent des termes. Un terme représente une constante de σ^{cte} ou une variable de σ^{var}. Des atomes avec des arités $k = 0$ sont nommés des atomes propositionnels ou propositions.

Un littéral l est un atome de p ou la négation de l'atome $\neg p$. Le symbole « $\neg$ » représente une négation classique. Le complément de p est $\neg p$ et vice versa. Un littéral représente une propriété d'une variable. La négation par défaut (NAF) est dénotée par un « *not* ». Elle est une extension de la négation classique et dénote qu'un littéral *not* l est vrai si rien ne permet de prouver que l est vrai.

Exemple :
$time = 5$. // représente une constante.
$aime(A, B)$. // représente un atome avec A et B représentent des variables.
$province(qc)$. // représente un atome avec qc représente une constante.

En général, une règle r issue de ASP a la forme suivante :

$$a_1 \cup \ \ldots \cup \ a_n \ \leftarrow \ b_1, \ldots, b_k, \ not \ b_{k+1}, \ldots, not \ b_m \ .$$

Les littéraux $\{a_1, \ \ldots, a_n\}$ sont appelés la partie de gauche ou tête H(r) d'une règle. La partie $\{b_1, \ldots, b_k, \ not \ b_{k+1}, \ldots, not \ b_m\}$ est appelée le corps ou partie droite de la règle avec $B^+(r) = \{b_1, \ldots, b_k\}$; $B^-(r) = \{not \ b_{k+1}, \ldots, not \ b_m\}$ et $B(r) = B^+(r) \cup B^-(r)$. Une règle r avec un seul littéral dans la partie tête (n=1) est appelée une règle normale. Dans le cas où $(k = m = 0)$, la règle représente un fait. Finalement, une règle r avec aucun atome dans la partie tête : $H(r) = \varnothing$ *ou* $(n = 0)$, représente une contrainte d'intégrité. Pour $(n > 1)$, la règle est dite disjonctive. L'exemple suivant expose un programme en langage ASP :

$noeud(n1)$. $//(faits)$
$noeud(n2)$. $//(faits)$
$noeud(n3)$. $//(faits)$
$lien(n1, n2)$. $//(faits)$
$lien(n2, n3)$. $//(faits)$
$couleur(bleu)$. $//(faits)$
$couleur(rouge)$. $//(faits)$

$\leftarrow\ noeud(X, Couleur),\ noeud(Y, Couleur),\ liens(X, Y).\quad //(contrainte)$
$aCouleur(X)\ \leftarrow\ nCouleur(X, Y),\ noeud(X),\ couleur(Y).\quad //(regle)$
$nCouleur(X, bleu) \cup nCouleur(X, rouge)\ \leftarrow\ noeud(X),\ not\ aCouleur(X).\quad //(disjonction)$

Ce petit programme en langage ASP représente un problème de coloriage de graphe. Les faits décrivent 3 nœuds liés non coloriés. Il existe deux couleurs, le rouge et le bleu. La contrainte d'intégrité informe que deux nœuds liés ne doivent pas avoir la même couleur. La règle disjonctive informe qu'un nœud X non colorié doit avoir l'une des couleurs suivantes {bleu, rouge}. Dans cet exemple, la règle disjonctive joue un rôle important dans le choix de la couleur du nœud. Nous remarquons à ce niveau que le programme ASP décrit un problème et ses contraintes, mais ne décrit pas la manière de résoudre le problème de coloriage. C'est à ce niveau que réside la puissance des langages descriptifs.

5.3.2 Sémantique du langage ASP

Le langage ASP s'appuie sur un solutionneur pour déterminer l'ensemble des réponses du programme. Dans un premier temps, il y a élimination des variables à travers un processus d'affectation nommé $ground(P)$. Admettons que pour un programme P, l'univers de Herbrand, dénoté HU_p, est l'ensemble de toutes les constantes $c \subset \sigma^{cte}$ du programme P. Supposons que la base de Herbrand d'un programme P, qu'on dénote HB_p, représente l'affectation de valeurs pour tous les littéraux $(a_1 \ldots\ a_n, b_1, \ldots, b_k)$ à partir des prédicats du programme P et de HU_p. L'affectation de valeurs des littéraux d'une règle $r \in P$ est obtenue par le remplacement de toutes les variables par une constante de HU_p. L'exemple suivant illustre cette opération d'affectation de valeur ou $ground(P)$:

Supposons le programme P suivant :
$lier(X, Z) \leftarrow lier(X, Y), lier(Y, Z).$
$lier(a, b).$
$lier(b, c).$
$lier(c, d).$

La fonction ground(P)retourne les resultats suivants :
$lier(a, b) \leftarrow lier(a, b), lier(a, b).$
$lier(b, c) \leftarrow lier(b, c), lier(b, c).$
$lier(c, d) \leftarrow lier(c, d), lier(c, d).$
$lier(a, c) \leftarrow lier(a, b), lier(b, c).$
$lier(b, d) \leftarrow lier(b, c), lier(c, d).$

...
$lier(a, b)$.
$lier(b, c)$.
$lier(c, d)$.

Afin de résoudre un programme P du modèle ASP, le solutionneur ASP identifie un ensemble de littéraux $X \subseteq HB_p$ qui soient consistants. Les littéraux sont dites consistantes si et seulement si $X \subseteq HB_p$ ne comporte pas des atomes tel que $X \nsubseteq \{p, \neg p\}$ avec $p \in HB_p$ (voir Baral, 2003). Par la suite le solutionneur identifie parmi les littéraux X un sous ensemble **minimal** de littéraux S qui satisfait l'ensemble des règles r du programme P. On dit que des littéraux S satisfont une règle r dans P quand :

- $B^+(r) \subset S$: l'ensemble B^+ fait partie de l'interprétation S.
- $B^-(r) \cap S = \varnothing$: aucune variable de l'ensemble interprétation ne fait partie de l'ensemble $B^-(r)$.
- $S \in H(r)$: les littéraux solutions sont éléments de de $H(r)$.
- $H(r) \cap S \neq \varnothing$: au moins un des littéraux de l'ensemble $H(r)$ fait partie de l'ensemble S.

Un modèle *positif* du programme P est une interprétation $I \subseteq HB_p$ tel que I satisfait toutes les règles r du programme P. Pour étendre cette définition dans le cadre des programmes avec négation par défaut (*not*), il y a lieu de transformer le programme P en fonction de l'ensemble I. Le nouveau programme P^I est obtenu par l'application des règles suivantes :

1. Supprimer toutes les règles r de P telles que $B^-(r) \cap I \neq \varnothing$.
2. Supprimer tous les $B^-(r)$ des règles restantes.

Au final, si le solutionneur détermine que l'ensemble réponse du nouveau programme P^I est identique à S dans ce cas, S correspond à l'ensemble réponse du programme P.

Par exemple, pour le programme ASP suivant :
$p \leftarrow a$.
$a \leftarrow not\ b$.
$b \leftarrow not\ a$.

On obtient l'ensemble de littérals suivants :
$X = \{a,\ b,\ p\}$
On obtient les deux ensembles réponses suivants : $S_1 = \{a,\ p\}$
$S_2 = \{b\}$

$P^{S_1} = \{a\ \leftarrow;\ p\ \leftarrow\ a\}$; l'ensemble réponse de P^{S_1} est identique à S_1.
$P^{S_2} = \{b\ \leftarrow;\ p\ \leftarrow\ a\}$; l'ensemble réponse de P^{S_2} est identique à S_2.
Dans cet exemple, l'ensemble réponse S_1 et S_2 sont les ensembles réponses du programme P.

Voici un autre exemple plus complexe qui démontre la manière de résoudre un programme ASP. Admettons le programme P suivant :
$a\ \leftarrow not\ b.$
$b\ \leftarrow not\ c.$
$d\ \leftarrow .$

On obtient les éléments suivants :
$X = \{a,\ b,\ c,\ d\}$
$S_1 = \{d,\ b\}$

$P^{S_1} = \{b \leftarrow;\ d \leftarrow\}$; l'ensemble réponse de P^{S_1} est identique à S_1.
Il est possible de démontrer que l'ensemble $S = \{a,\ d\}$ n'est pas un ensemble de réponse car $P^S = \{a \leftarrow; b \leftarrow;\ d \leftarrow\}$; l'ensemble réponse de P^S est **différent** de S. Donc ce n'est pas un ensemble réponse.
Dans cet exemple l'ensemble réponse S_1 est l'unique ensemble réponse du programme P.
Pour un exemple plus complexe, on pose le programme suivant (voir Schindlauer, 2008). Ces règles permettent de décrire que tous les oiseaux volent exception faite des pingouins :

$fly(X)\ \leftarrow\ bird(X),\ not\ ab(X).$
$ab(X)\ \leftarrow\ penguin(X).$
$bird(X)\ \leftarrow\ penguin(X).$
$bird(tweety).$
$penguin(skyppy).$

On obtient un seul et unique ensemble de réponses :
$S_1\{bird(tweety),\ penguin(skyppy),\ bird(skyppy),\ fly(tweety),\ ab(skyppy)\}.$

L'intérêt pour le formalisme ASP s'est intensifié ces dernières années, car d'une part il permet de décrire des problèmes complexes selon une logique déclarative puissante et d'autre part il est doté de solutionneurs performants. De nombreux domaines utilisent le langage ASP pour solutionner des problèmes de planification, de raisonnement à base de modèle, de diagnostic, de représentation des connaissances (voir Balduccini et Gelfond, 2003; Baral, 2003; Schindlauer, 2008), de configuration (voir Gelfond, 2007), de Bounded Model Checking (voir Tang et Ternovska, 2007), etc.

Toutefois le langage ASP demeure complexe pour un non-informaticien. Utiliser le langage ASP comme modèle de coordination des agents pour le système Leopar++ nécessite des connaissances poussées en programmation et représentation des connaissances.

C'est pour cette raison que nous comptons utiliser le langage d'actions comme intermédiaire entre le langage ASP et les besoins de l'utilisateur final. En effet, le langage d'actions est un langage de haut niveau qui utilise des expressions simples afin de planifier les actions des agents. Il présente l'avantage d'être facile d'utilisation et de compréhension et il est aisément traduisible en langage ASP. Les langages d'actions simplifient grandement les tâches de modélisation et ne nécessitent pas de bonnes connaissances en programmation.

5.4 Liens entre langage d'actions et ASP

Comme décrit précédemment, un langage d'actions utilise les fluents et les actions pour décrire les états du monde. Il a une expressivité variable dépendante du domaine d'application. Parmi les langages d'action, le langage A_k^c (voir Tu *et al.*, 2007) s'avère particulièrement adapté à la description des actions des agents du système Leopar++. Il permet de décrire les phénomènes suivants :

- Des effets dynamiques : l'exécution d'une action implique l'apparition d'un nouveau fluent dans l'environnement.
- Des effets statiques : un fluent implique l'apparition d'un nouveau fait dans l'environnement. Il ne nécessite pas l'exécution d'une action.
- Des conditions d'exécutions : des conditions sont nécessaires pour exécuter une action.
- Des états initiaux : des fluent décrivent des états initiaux.
- Des états finaux : des fluent décrivent des états finaux à atteindre.

Le langage d'actions A_k^c se définit par les axiomes suivants (voir Tableau 5.4) :

Tableau 5.1 Les axiomes du langages A_k^c

N°	Axiomes	Remarques
(1)	$initially(l)$	Définit l'état initial du système
(2)	$finally(l)$	Définit l'état final du système
(3)	$executable(a, F)$	Définit les préconditions d'une action
(4)	$causes(a, l, F)$	Définit les préconditions et les effets d'une actions
(5)	$if(l, F)$	Définit une condition

Supposons que a représente une action, F un ensemble de fluents $\{f_1, ..., f_n\}$, et l un litéral. Les axiomes (1) et (2) expriment l'état initial et final du système. Dans notre cas, l'état final représente l'état à atteindre, les objectifs à réaliser. L'axiome (3) informe que l'action a est exécutable lorsque les fluents de F sont vrais. L'axiome (4) exprime les effets dynamiques d'une action. Il exprime que l'action a ne s'exécute que lorsque les fluents F sont vrais, ce qui entraine la véracité du littéral l. Les fluents F est optionnel dans l'équation (4).

L'axiome (5) représente un effet statique. Les fluents F deviennent vrais dans le cas où les littéraux l sont vérifiés. Cet axiome ne fait pas intervenir des actions pour modifier l'état de l'environnement et le changement provient uniquement de l'environnement.

Les agents définis par le langage A_k^c sont automatiquement traduits en langage ASP. Un solutionneur ASP détermine un plan d'exécution conforme aux contraintes du domaine et aux objectifs finaux fixés par l'utilisateur.

Analysons de plus près la traduction du langage d'actions A_k^c selon le formalisme ASP (voir Tableau 5.2). Nous invitons le lecteur à se focaliser sur l'exécution des actions.

Tableau 5.2 Traduction du langage A_k^c

Expression	Traduction ASP
$initially(l)$	$holds(l, 1).$
$executable(a, F)$	$executable(a, T) \leftarrow holds(f1, T), ..., holds(fn, T), time(T).$ Avec $T = [0..n]; n \in N^+ \ ; \ f_i \in F$
$causes(a, l, F)$	$holds(l, T + 1) \leftarrow occurs(a, T), \ holds(f1, T), ..., holds(fn, T), time(T).$
$if(l, F)$	$holds(l, T) \leftarrow holds(f1, T), ..., \ holds(fn, T), \ time(T).$
$finally(l)$	$finally(l).$

Les contraintes d'intégrité du langages A_k^c sont traduites dans le tableau suivant (voir Tableau 5.3).

Tableau 5.3 Les contraintes d'intégrité du langage d'action A_k^c

Contraintes		Traduction ASP
Définition de l'objectif : un plan n'existe que lorsque tous les littéraux finaux « finally » se sont réalisés « holds » à l'instant T.		$negGoal(T) \leftarrow time(T), finally(X),$ $not\ holds(X,T).$ $goal(T) \leftarrow time(T), not\ negGoal(T).$ $existe\_plan \leftarrow goal(T), time(T).$ $\leftarrow not\ existe\_plan.$
Inertie et dynamisme des états : les deux premières expressions définissent qu'un état est vrai/faux tant qu'il n'y a pas un état contraire. La troisième expression indique qu'une action peut créer un nouvel état.		$holds(F,T+1) \leftarrow literal(F), time(T),$ $holds(F,T), not\ holds(\neg F, T+1).$ $holds(\neg F,T+1) \leftarrow literal(F), time(T),$ $holds(\neg F,T), not\ holds(F,T+1).$ $holds(F,T+1) \leftarrow literal(F), time(T), action(A),$ $executable(A,T), occurs(A,T), causes(A,F).$
Contrainte intégrité des états : exprime qu'à un instant donné, il ne peut exister un état et son contraire.		$\leftarrow holds(F,T), holds(\neg F,T), time(T),$ $literal(F).$
Condition d'exécution d'une action : une action ne s'exécute que si elle est définie comme étant exécutable. La condition d'unicité de l'action est préservée par « $\neg occurs$», qui assure qu'une seule action ne peut s'exécuter durant l'instant T.		$occurs(A,T) \leftarrow action(A), time(T),$ $executable(A,T), not\ \neg occurs(A,T).$ $\neg occurs(A,T) \leftarrow action(A), action(AA),$ $time(T),\ occurs(AA,T).$

À titre d'exemple, admettons que nous disposons de la description suivante en A_k^c (voir Tableau 5.4).

La résolution du modèle ASP du tableau 5.4 retourne l'ensemble réponse suivant :
$n = 2$
$T = [0..n];\ n \in N^+$
$S_1 = \{occurs(actionB,1),\ \ holds(fluentA,1),\ holds(fluentB,2),\ holds(fluentC,2),$
$\neg goal(1),\ goal(2)\}.$

À travers ces exemples, il en ressort que le formalisme ASP présente deux avantages importants. Premièrement, ASP est un langage déclaratif et non procédural. Il est en mesure de décrire de manière incrémentale un problème et de l'enrichir de manière progressive.

Tableau 5.4 Exemple de modèle A_k^c

Modèle A_k^c	Traduction en ASP
$initially(fluentA)$.	$holds(fluentA, 1)$.
$executable(actionB, fluentA)$.	$executable(actionB, T) \leftarrow holds(fluentA, T), time(T)$.
$causes(actionB, fluentB, \{\})$.	$holds(fluentB, T + 1) \leftarrow occurs(actionB, T), causes(actionB, fluentB), time(T)$.
$if(fluentB, fluentC)$.	$holds(fluentB, T) \leftarrow holds(fluentC, T), time(T)$.
$finally(fluentC)$.	$finally(fluentC)$.
Déclaration des littéraux & actions :	$literal(fluentA)$. $literal(fluentB)$. $literal(fluentC)$. $action(actionB)$.

Cette caractéristique le rend parfaitement adapté aux besoins des langages d'actions qui sont de nature déclarative. Deuxièmement, le langage ASP introduit deux opérateurs puissants que sont la négation par défaut et l'union. Ils permettent de définir des besoins spécifiques. Ainsi, l'utilisation de ASP et des langages d'actions forment un excellent duo pour le système Leopar++. D'une part, ils permettent de décrire aisément les agents du système, d'autre part, ils permettent de coordonner et de planifier les agents en fonction d'objectifs changeants.

CHAPITRE 6

LA COORDINATION DES AGENTS DE LEOPAR++

6.1 Langage d'actions A^c_{k-ext}

Dans ce chapitre, nous présentons le langage d'actions A^c_{k-ext} (voir Gaha *et al.*, 2011) afin de permettre la coordination des agents du système Leopar++. Le module de coordination du simulateur, élément central de notre architecture, se définit à l'aide du langage d'actions A^c_{k-ext} extension de A^c_k. Il est axé sur la facilité d'utilisation et intègre des fonctionnalités liées aux architectures tableau noir et multiagents : les sous-objectifs, le contrôle, les incertitudes, les interdictions d'exécution et le parallélisme (voir Corkill, 2003; Hewett et Hewett, 1993). Le langage A^c_{k-ext} se compose d'un ensemble fini d'axiomes, comme l'expose le tableau 6.1 :

Tableau 6.1 Les axiomes du langages A^c_{k-ext}

N°	**Axiomes**	**Remarques**
(1)	$initially(l)$	Définit l'état initial du système
(2)	$finally(l)$	Définit l'état final du système
(3)	$executable(a, F)$	Définit les préconditions d'une action
(4)	$causes(a, l, F)$	Définit les préconditions et les effets d'une action
(5)	$if(l, F)$	Définit une condition
(6)	$obviate(a, F, L)$	Permet d'interdire l'exécution d'une action
(7)	$uncertain(a, \{F_i\})$	Identifie les effets incertains d'une action
(8)	$control(L, F)$	Permet d'annuler la poursuite d'un objectif
(9)	$subGoal(a, F, L)$	Identifie des sous-objectifs à exécuter
(10)	$prefer(a, F)$	Identifie des actions prioritaires
(11)	$parallel$	Définit le mode d'exécution

Les axiomes (1-5) sont identiques aux axiomes du langage A^c_k. Ils expriment respectivement l'état initial du système, les conditions d'exécution d'une action, les effets dynamiques d'une action, les effets statiques d'une action et les états finaux du système.

Les axiomes (6-10) représentent des *connaissances de contrôle*. Ils apportent des informations qui affinent et optimisent le processus de planification. Supposons que le symbole a représente une action, F un ensemble de fluents, L un ensemble de fluents **finaux** et l un littéral.

L'axiome (6) informe que l'action a ne peut pas s'exécuter quand les fluents F sont vrais et L sont des objectifs finaux. Cet axiome a pour but d'accélérer la procédure de planification afin de minimiser le nombre d'actions à prendre en compte. Sa sémantique est différente de l'axiome *non_executable* du langage C car elle intègre le paramètre L. De plus, la traduction de *obviate* en ASP est différente de *non_executable.*[1] .

L'axiome (7) introduit le non déterminisme. Il exprime que l'exécution de l'action a entraine qu'uniquement un sous-ensemble des fluents $\{F_1, ..., F_i\}$ est Vrai, avec $F_i \subseteq F$. L'axiome *uncertain* permet de représenter un environnement dynamique où l'effet des actions est incertain. À titre d'exemple, nous définissons l'action de "lancer un dé à six faces" comme étant une action incertaine. La valeur retournée par le dé est inconnue à l'avance, avec $F_i = \{\{1\}, \{2\}, \{3\}, ..., \{6\}\}$.

L'axiome (8) permet d'annuler la poursuite des objectifs L dans le cas où les fluents F sont vérifiés. L'intérêt de cet axiome est de suspendre la poursuite de certains objectifs pour des situations précises. Par exemple, il peut arriver que le système se retrouve à faire des exécutions cycliques répétées. Pour y remédier il faut supprimer certains des objectifs initialement fixés pour éviter des exécutions infinies.

L'axiome (9) exprime des sous-objectifs. Il indique que l'action a est à exécuter dès l'instant où les fluents F sont vrais et que les littéraux L font partie de l'objectif final. Cet axiome augmente la performance de la planification en imposant au solutionneur ASP l'exécution de sous-objectifs.

L'axiome (10), permet de mettre une préférence sur certaines actions selon certaines contraintes F. Cette souplesse permet de spécifier que certaines actions sont préférables à d'autres et qu'elles doivent s'exécuter dès que possible. Le paramètre F est optionnel et son omission indique que l'action est préférée pour toutes les conditions.

Finalement, l'axiome (11), permet de spécifier la manière dont s'exécutent les actions. Dans le cas où *parallel* apparait dans le programme, les actions peuvent s'exécuter en parallèle tant que les contraintes d'intégrité sont préservées. Dans le cas contraire, les actions s'exécutent en mode séquentiel, qui représente le mode de planification par défaut.

Il est à noter que les axiomes que nous définissons établissent des relations entre les données et les actions uniquement. Il n'y pas de relation entre les actions. Cette spécificité est nécessaire, car nous définissons un modèle de coordination basé uniquement sur les données. Ainsi, nous respectons la philosophie des architectures tableaux noirs, qui offrent la possibilité aux utilisateurs d'introduire de nouveaux agents en ne tenant compte que des données manipulées. Cette caractéristique facilite l'ajout, la modification et la suppression de nouveaux

1. Le langage C traduit la règle non_executable comme étant une contrainte d'intégrité (voir Gelfond et Lifschitz, 1998). Dans notre cas, nous traduisons *obviate* comme une règle normale.

agents dans le système.

6.1.1 Traduction de A^c_{k-ext} en ASP

Dans cette section nous exposons la traduction du langage d'actions A^c_{k-ext} en langage ASP. Comme le langage d'actions est uniquement un langage de représentation et ne permet pas de raisonner sur les actions, il est nécessaire de le traduire en un autre langage, en l'occurrence ASP. La phase de traduction se réalise automatiquement à l'aide des règles suivantes :

Traduction de obviate(a, F, L)

L'axiome *obviate* rend une action non exécutable, et ce, tant que l'ensemble des fluents $F = \{f_1, \ldots, f_m\}$ sont vrais et que les fluents $L = \{l_1, \ldots, l_n\}$ sont des objectifs finaux. Le prédicat *obviate* joue un rôle important lors de la détermination de l'action exécutable. Il se retrouve dans l'axiome *executable*, défini plus loin, et indique si une action peut s'exécuter ou pas :

$$obviate(a, T) \leftarrow holds(f_1, T), ..., \ holds(f_m, T), \ finally(l_1), \ ..., \ finally(l_n), \ time(T).$$

Traduction de subGoal(a, F, L)

Le concept de sous-objectif est très puissant. Il permet de délimiter l'espace de recherche de l'action à exécuter si $L = \{l_1, \ldots, l_n\}$ sont des objectifs finaux et $F = \{f_1, \ldots, f_m\}$ sont vrais. Noter que dès l'instant où un sous-objectif est vérifié, il interdit, par l'introduction du prédicat *occSubgoal*, l'exécution d'une autre action dans le mode séquentiel. Comme nous le verrons plus loin, le prédicat *occSubgoal* est testé dans la règle de *causes*.

Si F est un ensemble vide :

$$2\{occurs(a, T), \ occSubgoal(T)\}2 \leftarrow finally(l_1), ..., \ finally(l_n), \ time(T).$$

sinon

$$2\{occurs(a, T), \ occSubgoal(T)\}2 \leftarrow holds(f_1, T), ..., \ holds(f_m, T),$$
$$not \ holds(c, T), \ finally(l_1), ..., \ finally(l_n), \ time(T).$$

Traduction de executable(a,F)

Nous traduisons l'axiome exécutable comme suit en ASP :

$executable(a,T) \leftarrow holds(f_1,T),\ ...,\ holds(f_m,T),\ not\ obviate(a,T),\ time(T).$

Traduction de uncertain(a,{Fi})

Pour des ensembles $\{F_1,\ ...,\ F_i\}$ avec $\{f_{11},...,f_{1j}\} = F_1$ et $F_1 \subseteq F$, nous traduisons l'indéterminisme comme suit :

$1\{F_1(T),\ ...,\ F_i(T)\}1 \leftarrow occurs(a,T),\ time(T).$
$holds(f_{11},T)...,holds(f_{1j},T)\ \leftarrow F_1(T),\ time(T).$
...
$holds(f_{i1},T),...,holds(f_{ij},T)\ \leftarrow F_i(T),\ time(T).$

Traduction de control(l,F)

L'axiome *control* permet de vérifier le plan d'action et de détecter des modèles d'exécution. Le plan d'exécution est dépendant des actions dont l'effet est incertain et il est nécessaire d'éviter des exécutions indésirables comme des boucles infinies. Dans l'axiome ci-dessous, le prédicat *releaseGoal* permet d'annuler la poursuite d'un objectif l.

$releaseGoal(l) \leftarrow holds(f_1,T),\ ...,\ holds(f_m,T),\ time(T).$

Traduction de causes(a,l,F)

Cet axiome vérifie les conditions d'exécution et les effets d'une action. Notez que dès qu'une action est interdite d'exécution durant l'instant T, le prédicat *obviate* est vrai. Le prédicat *holds* permet de définir les effets de l'action a exécutée $occurs(a,T)$. Le prédicat *causes* permet de prédire l'effet de l'execution d'une action au temps T. Nous traduisons la cause des actions comme suit :

$executable(a,T) \leftarrow holds(f_1,T),\ ...,\ holds(f_m,T),\ not\ obviate(a,T),$
$time(T).$

$holds(l, T+1) \leftarrow occurs(a, T),\ action(a),\ time(T).$

$causes(a, l, T) \leftarrow executable(a, T),\ action(a),\ time(T).$

Traduction de prefer(a,F)

La traduction de l'axiome stipule que l'action s'exécute en préférence dès que les fluents F sont vérifiés. Dans le cas où $F = \varnothing$, l'action est exécutée en priorité en tout temps :

si $F \neq \varnothing$

$2\{occurs(a, T),\ existsPreferedAction(a, T)\}2 \leftarrow executable(a, T),\ causes(a, c),$
$not\ holds(c, T),\ holds(f1, T),\ ...,\ holds(fm, T),\ time(T).$

sinon si $F = \varnothing$

$2\{occurs(a, T),\ existsPreferedAction(a, T)\}2 \leftarrow executable(a, T),\ causes(a, c),$
$not\ holds(c, T),\ time(T).$

Il est à noter que le prédicat *existsPreferedAction* se retrouve dans l'expression *occurs* dans le cas de l'exécution séquentielle.

Traduction des conditions d'exécution

Il y existe deux modes d'exécution : séquentiel et parallèle. Le mode séquentiel, par défaut, stipule qu'à chaque étapes de temps T, une seule et unique action peut s'exécuter. Pour ce faire, nous introduisons dans la traduction ASP les contraintes suivantes :

$1\{possibleAction(A, T)\ :\ action(A)\}1\ \leftarrow time(T).$

$occurs(A, T) \leftarrow executable(A, T),\ possibleAction(A, T),\ action(A),$
$not\ existsPreferedAction(A, T), not\ occSubgoal(T),\ not\ goal(T),\ time(T).$

La fonction $\{possibleAction(A, T) : action(A)\}\ \leftarrow time(T)$ permet d'énumérer toutes les actions possibles pour un temps T donné. Par exemple dans le cas où $T = [1, 2],\ A \in\ \{a, b\}$ alors la fonction retourne les deux ensembles réponses suivants :

$S_1 = \{possibleAction(a, 1)\}$
$S_2 = \{possibleAction(b, 1)\}$
$S_3 = \{possibleAction(b, 1), possibleAction(a, 1)\}$
...
$S_{16} = \{possibleAction(b, 2), possibleAction(a, 2), possibleAction(b, 1), possibleAction(a, 1)\}$

Maintenant, la fonction $1\{possibleAction(A, T) : action(A)\}1 \leftarrow time(T)$ avec les valeurs numériques entourant la fonction précédente $min\{...\}max$ sont une extension de la logique non monotone et indiquent au programme P que le nombre d'axiomes $nbrAxiomes$ retournés à vrais respectent la contraintes $min \leqslant nbrAxiomes \leqslant max$. Cette contrainte indique qu'une seule action est sélectionnée durant chaque étape de temps, assurant ainsi la séquentialité des actions à être exécutées. La fonction retourne les ensembles réponses suivants :

$S_1 = \{possibleAction(a, 1), possibleAction(a, 2)\}$
$S_2 = \{possibleAction(b, 1), possibleAction(b, 2)\}$
$S_3 = \{possibleAction(b, 1), possibleAction(a, 2)\}$
$S_4 = \{possibleAction(b, 2), possibleAction(a, 1)\}$

Le mode parallèle *parallel* est une alternative au mode séquentiel. Il permet que plusieurs actions s'exécutent en parallèle. Le planificateur ASP doit toutefois s'assurer qu'il n'y a pas un accès concurrent aux données en écriture. Cette vérification se fait à l'aide du prédicat *concurrentAcces*. Les contraintes suivantes de parallélisme viennent remplacer les contraintes précédentes de séquentialité :

$occurs(A, T) \leftarrow action(A),\ time(T),\ executable(A, T),\ action(AA),\ concurrentAcces(A, AA),$
$not\ occurs(AA, T),\ not\ goal(T).$

$concurrentAcces(A, AA, T) \leftarrow action(A),\ action(AA),\ executable(A, T),\ executable(AA, T),$
$causes(A, C1, T),\ causes(AA, C1, T),\ time(T).$

$concurrentAcces(A, AA, T) \leftarrow action(A),\ action(AA),\ executable(A, T),\ executable(AA, T),$
$causes(A, C1, T),\ causes(AA, \neg C1, T),\ time(T).$

$concurrentAcces(A, AA, T) \leftarrow concurrentAcces(AA, A, T),\ action(A),\ action(AA).$
$\leftarrow action(A),\ concurrentAcces(A, AA, T),\ action(AA),\ occurs(A, T),\ occurs(AA, T),$

$time(T)$.

Lors de l'exécution parallèle, il est important d'empêcher l'accès concurrents à la même ressource. Pour ce faire, la fonction *concurrentAcces* identifie les paires d'actions qui ne doivent pas s'exécuter en parallèle. L'axiome *concurrentAcces* énumère les actions qui ont un effet sur le même fluent. Par exemple, les actions $move(a, b)$ et $move(a, table)$ ont pour effet consécutif $on(a, b)$ et $on(a, table)$. Ils font référence au même effet [2] et ils ne doivent pas s'exécuter en parallèle. De même, il est interdit d'exécuter des actions qui ont des effets opposés, comme par exemple $on(a, b)$ et $\neg on(a, b)$.
La règle suivante vérifie qu'un fluent et son opposé ne peuvent coexister au même temps t.

$\leftarrow holds(F, T),\ holds(\neg F, T),\ fluent(F),\ time(T).$

Traduction de l'inertie

Nous traduisons les lois d'inerties (voir Lifschitz, 1999) comme suit :

$holds(f, T+1) \leftarrow time(T),\ holds(f, T),\ not\ holds(\neg f, T+1).$
$holds(\neg f, T+1) \leftarrow time(T),\ holds(\neg f, T),\ not\ holds(f, T+1).$

Traduction des états initiaux

Les états initiaux respectent l'hypothèse du monde fermé (Closed World Assumption). Nous traduisons les états initiaux comme suit :

$holds(f, 0) \leftarrow initially(f).$
$holds(\neg f, 0) \leftarrow not\ initially(f).$

Traduction de l'objectif final

Afin de guider le processus de planifications vers l'objectif final, nous introduisons les règles suivantes. Elles intègrent l'axiome *releaseGoal*, qui permet d'annuler la poursuite/concrétisation

2. Dans le cas d'actions qui retournent un effet composé de plusieurs attributs, un traitement additionnel permet de détecter un accès concurrent.

d'un objectif :

$finallyGoal(l) \leftarrow finally(l),\ not\ releaseGoal(l).$
$negGoal(T) \leftarrow finallyGoal(l),\ not\ holds(c,T),\ time(T).$
$goal(T) \leftarrow time(T),\ not\ negGoal(T).$
$existPlan \leftarrow goal(T),\ time(T).$
$\leftarrow not\ existPlan.$

La contrainte d'intégrité *existPlan* fait en sorte de garder les plans qui mènent à l'objectif final. Tous les plans qui ne mènent pas à l'objectif final sont éliminés de l'espace de recherche.

6.2 Exemple d'ordonnancement

Nous présentons ci-dessous un exemple élémentaire d'ordonnancement (voir Figure 6.1), afin de mieux comprendre les instructions du nouveau langage d'action A^c_{k-ext}. Prenons le cas classique de l'empilage de blocs. Admettons qu'à partir d'une configuration initiale de blocs, on planifie une suite d'actions pour atteindre la configuration finale souhaitée.

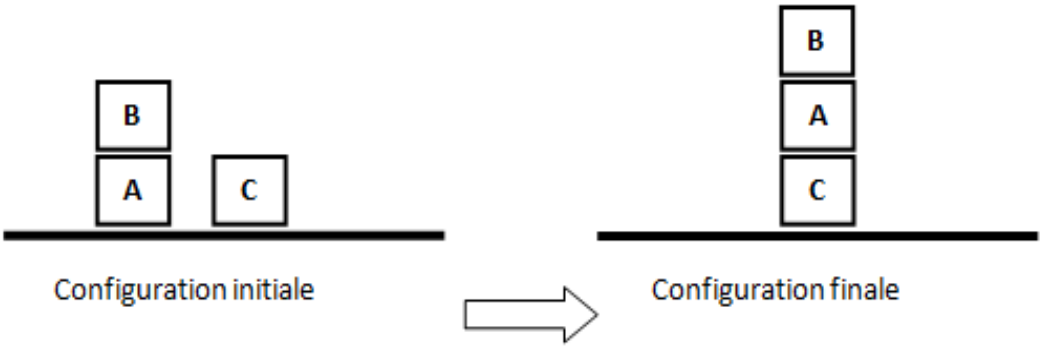


Figure 6.1 Exemple d'ordonnancement de blocs

Pour exprimer l'état initial des blocs et l'état final souhaité à l'aide du langage A^c_{k-ext}, nous définissons les axiomes suivant :

$initially(on(blocB, blocA)).$
$initially(on(blocA, table)).$
$initially(on(blocC, table)).$
$finally(on(blocC, table)).$

$finally(on(blocA, blocC))$.
$finally(on(blocB, blocA))$.

Les axiomes suivants informent sur l'action de déplacer un bloc et sur les conséquences de celle-ci :

$causes(moveOn(X,Y), on(X,Y), \{clear(X), clear(Y)\})$.
$causes(moveOn(X,Y), clear(X), \{clear(X), clear(Y)\})$.
$if(on(X,Y), \neg clear(Y))$.

Dès à présent, à l'aide du langage A^c_{k-ext} nous introduisons des connaissances de contrôle pour guider plus efficacement la procédure de planification. Nous supposons que la fonction *goodTower(Y)* veut dire que tous les blocs en dessous d'un bloc donné Y forment la solution finale :

$subGoal(move(X,Y), \{goodTower(Y), clear(X), clear(Y)\}, on(X,Y))$.
$subGoal(move(X, table), \{clear(X), not\ on(X, table)\}, on(X, table))$.
$obviate(move(X,Z), \{clear(X), goodTower(Y)\}, \{(X,Y)\})$.

Le premier sous-objectif indique qu'il faut déplacer un bloc X sur un bloc Y dans le cas où l'objectif final est d'avoir X sur Y et que tous les blocs en-dessous du bloc Y composent la solution finale. Le deuxième sous-objectif indique qu'il faut déplacer, de manière prioritaire, le bloc sur la table dans le cas où celui-ci fait partie de la solution finale. Finalement, les deux derniers axiomes *obviate* évitent de défaire une tour qui fait partie de la configuration finale.

Ces connaissances de contrôle optimisent le temps de traitement pour la détermination d'un plan d'exécution. En effet, nous avons réalisé une série de tests avec différents problèmes de planification de blocs, et il s'est avéré que le langage A^c_{k-ext} s'exécute plus rapidement que les autres langages d'actions.

Des tests ont été réalisés sur une machine IntelCore2Duo 2.0 GHz avec 4 GB RAM fonctionnant avec le solutionneur ASP Clingo 3.0.4. L'utilisation du solutionneur Clingo se base sur le fait qu'il est l'un des solutionneurs les plus performants de la compétition ASP de 2011. Nous résumons dans le tableau ci-dessous les résultats empiriques obtenus. Nous avons testé le langage A, le langage B, le langage C (voir Lifschitz et Turner, 1999), le langage A avec des connaissances de contrôle (voir Baral, 2003), le modèle A^c_k (voir Son *et al.*, 2006) et finalement notre langage d'actions A^c_{k-ext}. Les résultats sont présentés aux tableaux 6.2 et 6.3

Les données des tableaux sont triées en fonction du nombre d'étapes nécessaires pour

Tableau 6.2 Comparaison des langages d'actions sans connaissances de contrôle

Nom problème	A	B	C	Nbr étapes
Prob 5	0,78 s	0,58	0,187 s	7
Prob 7	0,421 s	0,87	1,2 s	10
Prob 14	ND	51 s	76 s	19
Prob 17	ND	ND	ND	23
Prob H1	ND	ND	ND	17
Prob H2	ND	ND	ND	23
Prob H3	ND	ND	ND	24
Prob H4	ND	ND	ND	26
ND : Non-Determiné				

Tableau 6.3 Comparaison de langages d'actions avec connaissances de contrôle

Nom problème	A	A_k^c	A_{k-ext}^c	Nbr étapes
Prob 5	0,56 s	0,14 s	0,094 s	7
Prob 7	1,81 s	0,312 s	0,197 s	10
Prob 14	ND	2,418 s	1,2 s	19
Prob 17	ND	5,97 s	1,5 s	23
Prob H1	ND	18,393 s	0,99 s	17
Prob H2	ND	552 s	3,0 s	23
Prob H3	ND	ND	3,13 s	24
Prob H4	ND	ND	28,5 s	26
ND : Non-Determiné				

solutionner le problème. Ces résultats permettent de conclure que le langage A_{k-ext}^c offre des performances supérieures aux autres langages d'actions. Nous remarquons que les différences de performances sont très importantes pour les problèmes complexes : Prob H1, Prob H2, Prob H3 et Prob H4.

Ces différences de performance sont possibles grâce à l'introduction de connaissances de contrôle comme les sous-objectifs et à l'interdiction de l'exécution de certaines actions. De plus, la manière de traduire les axiomes en ASP influence les performances du solutionneur. En effet, plus le nombre de fluents d'un programme P est important et plus le processus d'affectation de variables du $ground(P)$ est couteux en temps CPU. Il est important de représenter uniquement les fluents qui entrent en compte lors de l'exécution de l'actions. Par exemple, il est inutile de représenter l'information $(\neg on(blocA, BlocB))$ dans le cas où les axiomes n'utilisent pas cette information.

Cependant, il est à remarquer que les résultats de A_{k-ext}^c sont moins bons que les performances des planificateurs classiques à l'image du GraphPlan (voir Tableau 6.4). Il n'en

demeure pas moins que le langage ASP offre une plus grande souplesse dans la description des actions et est un langage de programmation à part entière.

Tableau 6.4 Comparaison des performances de planification

Nom problème	A^c_{k-ext}	GraphPlan
Prob 5	0,094 s	0,01 s
Prob 7	0,197 s	0,01 s
Prob 14	1,2 s	0,02 s
Prob 17	1,5 s	0,02 s
Prob H1	0,99 s	0,01 s
Prob H2	3,0 s	0,03 s
Prob H3	3,13 s	0,1 s
Prob H4	28,5 s	0,9 s

6.2.1 Exemple d'ordonnancement pour le domaine électrique

Appliquons dès à présent le langage A^c_{k-ext} pour un exemple plus concret. Admettons que nous ayons à simuler un réseau de distribution électrique. Supposons que le système de simulation se compose de 4 agents :

1. L'agent Aprem, spécialisé dans l'écoulement de puissance. Il calcule les valeurs électriques du réseau de distribution. Il est responsable de la détermination de la tension, du courant et de la puissance électrique pour les différents composants du réseau de distribution.
2. L'agent Régulation est responsable du maintien de la tension en sortie des bornes des transformateurs. Il est en mesure d'ajuster le changeur de prise du transformateur. En modifiant le rapport de transformation entre l'enroulement primaire et secondaire, il maintient la tension de sortie du transformateur dans un intervalle souhaité.
3. L'agent Réparation est spécialisé dans la simulation de la réparation d'une panne d'un réseau de distribution. Il simule une équipe d'intervention qui sillonne le réseau afin de localiser la panne et y remédier. Cette opération engendre des modifications dans la configuration du réseau afin d'isoler les sections défectueuses et de réalimenter les sous-sections saines du réseau.
4. L'agent Affichage permet d'afficher graphiquement l'état du réseau, le nombre de clients, les types des composants, la longueur et la nature des câbles, les durées d'interruption et certains indices électriques.

Le tableau suivant 6.5 est une description possible du modèle de coordination des agents du système Leopar++ :

Tableau 6.5 Exemple de modèle de coordination

Fluents :	
reguler.	(1)
afficher.	
valElectrique.	
panne.	
Actions :	
causes(aprem, valElectrique, {¬ valElectrique}).	(2)
causes(reparation, ¬panne. {panne}).	(3)
causes(reparation, ¬valElectrique, {panne}).	(4)
causes(reparation, ¬*reguler*, {panne}).	(5)
causes(affichage, afficher, {¬affichage}.	(6)
executable(regulation, {¬ reguler }).	(7)
uncertain (regulation, {{reguler}, {¬ reguler, ¬valElectrique}}).	(8)
Connaissances de contrôle :	
subGoal(aprem,{¬valElectrique}, reguler).	(9)
prefer(aprem).	(10)
parallel.	(11)
États initiaux :	
initially(panne).	(12)

La ligne (1) informe le système sur les fluents. Les lignes (2) (3) (4) (5) et (6) indiquent les conditions d'exécution des agents Aprem, Réparation et Affichage. Il est à noter que l'exécution de l'action réparation entraine l'apparition de trois fluents : ¬*reguler*, ¬*valElectrique* et ¬*panne*. Les lignes (7) et (8) introduisent le concept d'incertitude. En effet, l'exécution de l'action Régulation peut nécessiter plusieurs étapes avant ajustement de la tension en sortie. Il existe deux effets possibles pour l'action Régulation : «*reguler* » ou «¬*reguler*, ¬*valElectrique* ». Le premier effet indique que la tension aux bornes du transformateur est régulée. Le deuxième indique qu'il y a eu modification de la tension en sortie du transformateur nécessitant le recalcul des valeurs électriques du réseau. Il arrive souvent que pour atteindre la tension de sortie souhaitée, plusieurs interactions entre les agents Aprem et Régulation soient nécessaires pour permettre au changeur de prise de graduellement établir l'équilibre.

La ligne (9) indique que si l'objectif est de réguler le réseau, il est impératif de réaliser un écoulement de puissance en exécutant l'agent Aprem. Le sous-objectif est ignoré dès qu'il y a calcul des valeurs électriques. La ligne (10) stipule que l'action *Aprem* est prioritaire à toutes

les autres actions et doit s'exécuter dès que possible. La ligne (11) est optionnelle est permet d'activer les exécutions parallèles ou séquentielles. Il est à noter que l'exécution séquentielle représente la valeur par défaut. Finalement, la ligne (12) stipule qu'initialement le réseau est en panne.

Le tableau 6.6 présente les différents scénarios d'execution que peut réaliser le coordinateur en fonction des objectifs finaux définis par l'utilisateur.

Tableau 6.6 Exemple de scénarios d'execution

	Objectifs finaux	Plan possible	Hypotèses
Scénario 1	finally(valElectrique)	occurs(aprem,0)	
Scénario 2	finally(¬ panne)	occurs(reparation,0)	
Scénario 3	finally(valElectrique) finally(afficher)	occurs(aprem,0) occurs(affichage,1)	
Scénario 4	finally(valElectrique) finally(afficher) finally(reguler)	occurs(aprem,0) occurs(regulation,1) occurs(aprem,2) occurs(affichage,3)	Régulé à t=2
Scénario 5	finally(valElectrique) finally(afficher) finally(reguler) finally(¬panne)	occurs(aprem,0) occurs(regulation,1) occurs(reparation,2) occurs(aprem,3) occurs(regulation,4) occurs(affichage,5)	Régulé à t=2 Régulé à t=5

Nous constatons que les plans diffèrent selon les objectifs finaux. Le diagramme de transition suivant présente une vue de haut niveau **des différents plans de coordination possibles** (voir Figure 6.2). Ce graphique expose toutes les interactions possibles entre les agents ainsi que les interactions des scénarios du tableau 6.6. Nous remarquons qu'il y a une exécution cyclique entre les agents Aprem et Régulation. Ceci est dû au fait que les effets de l'action Régulation sont indéterminés et peuvent « $reguler$» ou « $\neg reguler$, $\neg valElectrique$». C'est dans ce genre de cas que l'axiome *control* a toute son utilité. Il permet d'éviter l'exécution des boucles infinies en annulant certains objectifs. Par exemple il peut arriver que l'agent Régulation ne soit pas en mesure de réguler le réseau après n itérations, avec n suffisamment grand. Dans ce cas, pour ne pas bloquer le système, l'axiome *control* permet de détecter ce modèle d'exécution et d'annuler la poursuite de l'objectif de régulation. Il permet le système de poursuivre normalement son exécution.

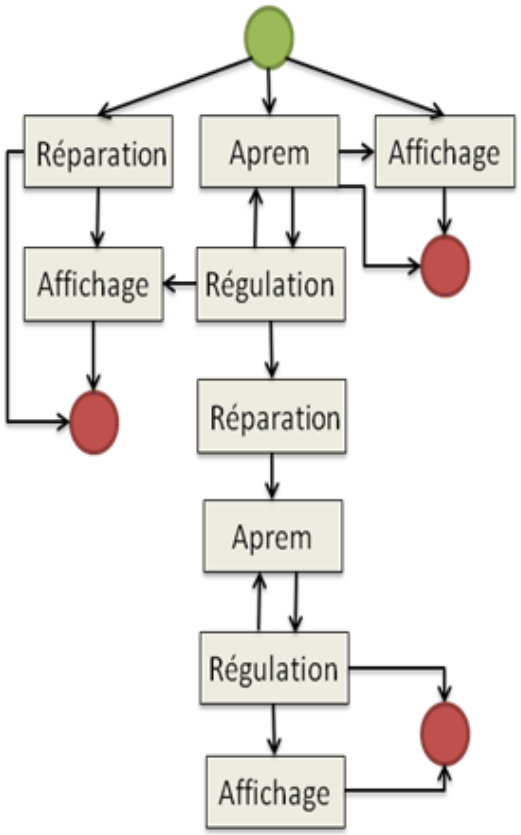


Figure 6.2 Diagramme de transition pour l'exemple électrique

6.3 Interaction coordinateur et agents de simulation

Pour les actions dont l'effet est indéterminé à l'avance, il est nécessaire d'établir une communication entre les agents du système et le module de coordination ASP. Pour ce faire, nous avons combiné le fichier de transition ASP, le solutionneur ASP Clingo et les agents du système Leopar++ par l'entremise du code suivant :

```
Axiomes := chargementAxiomes
t = 0;
(P, S):= resolutionModele(Axiomes)
While (exists occurs(a,t) in P){
     e := execute(a)
     t := t+1;
     if (existe f dans S tel que f est inconsistant avec (e)){
      Axioms := Axioms U {e};
```

```
        (P, S) := resolutionModele(Axioms);
        }
}
```

Posons que P et S sont des ensembles de prédicats de la forme *occurs*(a_i,t_i) et *holds*(f_i,t_i), respectivement, où $a_i \subseteq A$ et $f_i \subseteq F$. La fonction *chargementAxiomes* charge les axiomes qui décrivent le problème de planification et par la suite *resolutionModele* demande au solutionneur ASP Clingo de calculer le plan de coordination. Ce dernier sélectionne le plan optimal avec le moins d'étapes et retourne une séquence P d'actions et des fluents attendus S qui forment le plan final. Chaque action et fluent contient un temps t indiquant le moment de l'exécution de l'action et ou de l'apparition de l'état. Une action a de P représente un agent particulier à exécuter. La fonction *execute* retourne la valeur du fluent e au temps $t+1$ (que nous nommons fluent observé). Ce fluent n'est connu qu'après l'exécution effective de l'agent. Si le fluent observé e retourné est inconsistant avec un des fluents attendu de $f \in S$ au temps t, le code précédent met à jour le modèle en introduisant le fluent observé e. Par la suite, Clingo est ré-exécuté afin de déterminer le nouveau plan optimal.

Ce code précédent établi un lien de dépendance entre le planificateur ASP et les agents du système. Il représente un mécanisme nouveau car il est à notre connaissance le seul système qui ré-intègre des actions et faits nouveaux dans le modèle ASP. Le processus de planification est incrémental et est guidé par des actions et faits passés. De plus le système Leopar++ est unique en son genre car il est le seul système qui utilise les langages d'actions pour contrôler et coordonner l'execution des agents.

6.3.1 Langage A^c_{k-ext} et système Leopar++

Nous venons de définir un langage d'actions spécialement adapté aux systèmes multiagents utilisant un tableau noir comme moyen de communication entre les agents. Notre langage d'action est particulier car il est spécialement adapté aux architecture multiagents hybrides. Il utilise uniquement les fluents pour coordonner les agents et ne dispose pas de règles qui lient les actions. Ainsi, le langage A^c_{k-ext} respecte la vision des tableaux noirs qui interdit à un agent de référer à un autre agent.

Notre langage d'action introduit des connaissances de contrôle qui permettent de surpasser en terme de performance les langages d'actions de références. Ceci s'explique aussi par notre manière de traduire les axiomes, comme la non exécution et la parallélisation. En effet, notre traduction s'appuie sur les liens entre données et n'utilise pas les contraintes d'intégrité lors du processus de planification. Cette modification augmente grandement les performances du système.

Au final, le langage A^c_{k-ext} s'avère être un langage performant et facile d'utilisation pour permettre la description incrémentale des agents et de leurs effets. Il constitue un atout majeur pour le système Leopar++ qui rend accessible les systèmes multiagents aux non informaticiens.

6.3.2 Application du langage A^c_{k-ext}

Il est possible d'appliquer le langage A^c_{k-ext} dans des domaines autres que celui de la coordination des agents. Une première utilisation possible concerne le domaine de la planification et de l'orchestration. Il s'agit d'utiliser le langage d'action pour planifier une suite d'action afin d'atteindre des objectifs particuliers. Une autre application concerne le domaine de la vérification formelle. Il est possible de décrire les actions du modèle à l'aide de A^c_{k-ext} et par la suite de vérifier si une ou plusieurs séquences d'actions se produisent selon certaines conditions. Une autre application concerne le raisonnement à base de modèle. Comme notre langage utilise les actions pour raisonner, il est envisageable de raisonner sur les effets (i.e. Fluent). C'est à dire que notre langage permettrai de déterminer comment le modèle évoluerai dans le cas où on lui applique des actions particulières.

CHAPITRE 7

SIMULATIONS ET RÉSULTATS

7.1 Introduction

Pour réaliser des études de fiabilité, nous avons mis en place une simulation de type Monte-Carlo pour l'étude des réseaux de distribution. Pour ce faire, nous avons développé des agents et défini un modèle de coordination afin de tester le comportement du réseau face à des pannes répétitives. Les résultats obtenus ont été comparés avec un autre système multiagents : Jade. Cette comparaison permet de positionner notre système par rapport à d'autres simulateurs en termes de performance et de facilité d'utilisation.

7.2 Étude de la fiabilité : agents

7.2.1 Agent Aprem

L'outil d'analyse paramétrique des réseaux électriques (Aprem) est un environnement de développement qui se veut simple et facile d'utilisation pour réaliser des écoulements de puissance. L'agent Aprem a été développé avec le souci d'être accessible à des non-informaticiens.

L'utilisateur définit dans un premier temps (1) un réseau électrique avec toutes ses propriétés électriques. Dans un deuxième temps, (2) l'utilisateur d'Aprem est en mesure de modifier la configuration du réseau. Ces modifications sont relatives aux variations de la charge ou de la production, à la modification de l'état des interrupteurs du réseau, à l'introduction ou à la suppression de composants, à la modification des paramètres des équipements et à la modification de la topologie du réseau. Une fois que ces deux étapes ont été réalisées, l'outil Aprem réalise des écoulements de puissances et retourne les valeurs électriques de chacun des composants.

L'outil Aprem est né dans les laboratoires de l'énergie électrique de l'École Polytechnique dans le but de mener, entre autre, des études de fiabilité des réseaux électriques. Il intègre un langage de haut niveau qui permet de décrire et de spécifier les scénarios d'évolution d'un réseau électrique. Il intègre un moteur de calcul de puissance permettant de déterminer et d'analyser les valeurs électriques du réseau. L'outil Aprem utilise la méthode de Newton-Raphson pour le calcul de l'écoulement de puissance. Cette méthode se démarque par sa rapidité à résoudre de grands systèmes (voir Tinney et Hart, 1967).

L'outil Aprem s'exécute sur l'environnement Matlab. Ce choix a été décidé par l'équipe

de recherche pour l'aisance qu'offre Matlab dans la manipulation matricielle et les calculs mathématiques. L'outil Aprem utilise la puissance de Matlab afin de représenter un réseau électrique. Il utilise une matrice nommée *matrice nodale augmentée*. Celle-ci contient tous les détails associés à la modélisation du réseau. Elle s'appuie sur les éléments suivants :

1. Les impédances,
2. Les sources de tension,
3. Les sources de courant,
4. Les transformateurs,
5. Les interrupteurs.

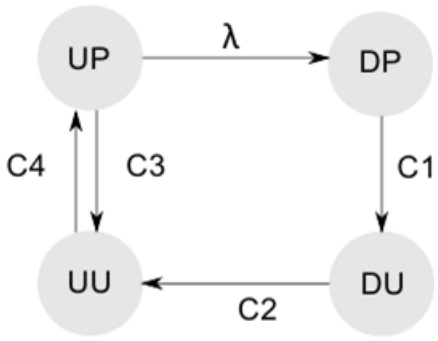
Pour résoudre les analyses de répartition de puissance, il faut ajouter des contraintes à la matrice nodale et ainsi obtenir un système non-linéaire. Les inconnues de la matrice sont évaluées par itération suivant la méthode de Newton-Raphson (voir Allemong *et al.*, 1993).

7.2.2 Agent Panne

L'agent Panne est responsable de la mise en panne d'un ou plusieurs composants du réseau de distribution. À chaque exécution, l'agent détermine les composants qui vont tomber en panne selon la probabilité de panne λ du composant. Un nombre réel aléatoire est généré $RND = [0, 1],\ RND \in \Re$ pour chaque composant, et dans le cas où $RND < \lambda$, le composant est sélectionné pour être mis en panne. Il passe du statut *UP* à celui de *DP* indiquant que le composant n'est plus alimenté (voir Figure 7.1). L'état *DP* évite à un composant d'être mis en panne dans le cas où il n'est pas alimenté.

7.2.3 Agent Protection

L'agent Protection est responsable de l'activation des mécanismes de protection d'un réseau de distribution. Quand une panne se produit, l'agent de protection active les mécanismes de protection en amont par l'ouverture automatique des disjoncteurs de poste. L'agent Protection reproduit le comportement réel des composants de sectionnement du réseau lorsque survient une panne. Une fois la protection activée, la contrainte *C1* (voir Figure 7.1) devient vraie, engendrant la transition de l'état du composant de *DP* à *DU*. Elle indique que le composant est dorénavant en panne et est non-alimenté. Parallèlement, les composants non-alimentés à cause de l'isolation en amont, changent de l'état *UP* à l'état *UU* indiquant qu'ils sont non-alimentés, mais toujours fonctionnels. La durée de l'état *UU* dépend du temps



λ: probabilité de panne
DP : Défectueux et alimenté
DU : Défectueux et non-alimenté
UU : fonctionnel et non-alimenté
UP : fonctionnel et alimenté
C1 : Vrai si composant non-alimenté
C2 : Vrai si composant est réparé
C3 : Vrai si composant est non-alimenté
C4 : Vrai si composant est réalimenté

Figure 7.1 Diagramme de transition des états des composants

mis par l'équipe d'intervention pour localiser et réparer la panne. Il est à noter que lorsque l'agent active les mécanismes de protection, la configuration du réseau change, ce qui implique l'exécution de l'agent responsable de la régulation de la tension au poste.

7.2.4 Agent Réparation

L'agent réparation est certainement l'agent le plus complexe du système. Il reproduit le comportement des équipes d'intervention pour isoler et réparer la panne. Il est basé sur un moteur de règles et est en mesure d'adopter différentes stratégies de localisation et de réparation de la panne. L'agent simule des équipes d'intervention qui patrouillent tout le long du réseau pour localiser la panne. L'équipe est capable d'identifier les sections en amont qui sont saines et y rétablir le courant. Cette opération est possible grâce à l'ouverture des interrupteurs à proximité de la panne.

La restauration en aval permet à travers des attaches de ré-alimenter une sous-section saine du réseau. Toutefois, cette opération nécessite de déterminer s'il n'existe pas de surcharge des câbles lors de la procédure de ré-alimentation. L'agent Aprem retourne les valeurs électriques du réseau et c'est à l'agent réparation de déterminer s'il est possible de réaliser une alimentation en aval sans risque de surcharge. Nous rappelons qu'à l'image des équipes d'intervention humaine, il est possible d'exécuter plusieurs agents Réparation en parallèle.

Une fois que l'opération de réparation a été effectuée, l'agent Réparation affecte à tous

les composants non-alimentés la durée de non-alimentation. Cette durée va servir en fin de processus Monte-Carlo pour le calcul d'indices de performances et va permettre d'identifier les sections du réseau les plus sensibles aux pannes.

7.2.5 Agent Calculateur_Indice

L'agent est exécuté à la fin de chaque simulation de rétablissement. Il collecte les durées de non-alimentation et calcule les indices de performances électriques suivants :

$$SAIDI = \sum (r_i \times N_i) / \sum N_t$$

$$CAIDI = \sum (r_i \times N_i) / \sum N_i$$

$$SAIFI = \sum (N_i) / \sum N_t = SAIDI/CAIDI$$

Avec :

i = le numéro de la charge.
r_i = temps de restauration en heures.
N_i = Nombre total de clients interrompus.
N_t = Nombre total de clients desservis.
$SAIDI$ = *System Average Interruption Duration Index*. Il représente la durée moyenne d'interruption en heures pour chaque client desservi.
$CAIDI$ = *Customer Average Interruption Duration Index*. Il représente la durée moyenne d'interruption en heures par client.
$SAIFI$ = *System Average Interruption Frequency Index*. Il représente le nombre moyen d'interruptions que le client connaitra.

7.2.6 Agent Régulation

L'agent Régulation est responsable du maintien de la tension en sortie du secondaire des transformateurs. Il est en mesure d'ajuster le changeur de prise du transformateur en modifiant le rapport de transformation entre l'enroulement primaire et secondaire. Il maintient la tension de sortie du transformateur dans un intervalle souhaité.

7.2.7 Étude de la fiabilité : module de coordination

Pour coordonner les agents, nous définissons le modèle de coordination A^c_{k-ext} du tableau 7.2.7 :

Tableau 7.1 Modèle de coordination pour l'étude de fiabilité

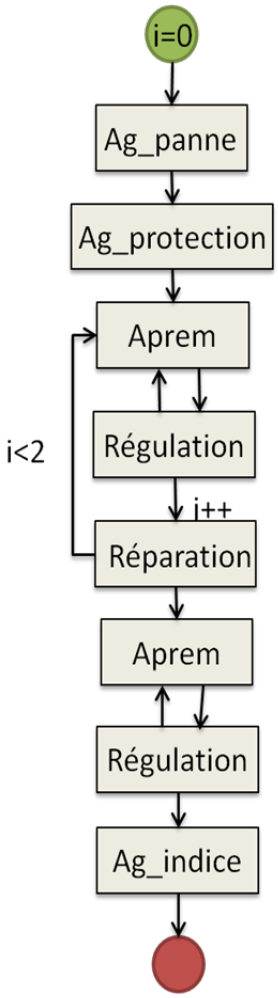
	Modèle A^c_{k-ext}
(1)	initially(¬ valElec). initially(protection).
(2) (3)	causes(ag_panne, {panne, ¬ protection}, ¬ panne). subgoal(ag_panne, ¬ panne, indice).
(4)	obviate(ag_panne,panne,{}).
(5)	causes(ag_protection,{ protection, ¬ elecVal, ¬ regulation}, {¬ protection, panne}).
(6)	prefer(ag_protection).
(7)	causes(aprem, elecVal, ¬ elecVal).
(8)	uncertain (ag_regulation, {{¬ regulation, ¬ elecVal} ;{regulation}}). executable(ag_regulation, {¬ regulation, elecVal}).
(9)	uncertain(ag_reparation,{{ panne, ¬ regulation,¬ elecVal} ;{¬ panne,¬ regulation,¬ elecVal }}). executable(ag_reparation,{panne,regulation,elecVal}).
(10)	causes(calcul_indice, indice, {¬ indice,¬ panne, regulation}).
(11)	finally(¬ panne). finally(indice).

La ligne (1) définit l'état initial du système qui représente un réseau protégé et dont nous ne connaissons pas les valeurs électriques. Nous rappelons que l'état initial utilise l'univers clos (Closed World Assumption), ce qui signifie que si un état initial n'est pas explicitement défini, alors il est automatiquement mis à Faux.
La ligne (2) informe sur les effets et les pré-conditions de l'agent Panne. Le sous-objectif de la ligne (3) indique que l'agent Panne doit s'exécuter quand l'un des objectifs finaux fait référence aux indices de performances. Cet axiome permet d'exécuter l'action Panne lors du premier pas de temps.
La ligne (4) annule le sous-objectif de la ligne (3) dès l'instant où une panne est générée. Cette instruction permet d'éviter des exécutions cycliques entre l'agent Réparation et l'agent Panne. Dans notre exemple, nous avons besoin que l'agent Panne s'exécute pour pouvoir mettre certains composant en pannes et réaliser des études de fiabilité.
La ligne (5) indique que l'agent Protection s'active lors de l'apparition d'une panne. Son exécution permet de protéger le réseau par l'ouverture du disjoncteur principal. Cette opération modifie les valeurs électriques et les charges alimentées du réseau.

La ligne (6) exprime que l'action de Protection est prioritaire et est exécutée dès que possible. Ceci permet de prioriser l'agent Protection sur l'agent Aprem, par exemple.
La ligne (7) exprime que l'exécution de l'agent Aprem réalise un écoulement de puissance et retourne les valeurs électriques.
La ligne (8) introduit le non-déterminisme pour l'agent Régulation. Cette expression permet de faire interagir les agents Aprem et Régulation. L'exécution de l'action Régulation retourne soit un réseau dont la régulation est satisfaite, soit un réseau qui n'est pas bien régulé et qui nécessite une modification du changeur de prise du transformateur principal.
La ligne (9) introduit le non-déterminisme pour l'agent responsable de la Réparation des pannes. Cette expression permet de faire interagir les agents Aprem, Régulation et Réparation. En effet, la réparation de la panne se fait en deux étapes. (1) Une première étape consiste en un rétablissement en amont et en aval des sections saines. Cette étape reconfigure la topologie du réseau sans toutefois réparer la panne. Nous remarquons que les valeurs électriques sont nécessaires pour que l'agent Réparation puisse s'exécuter, afin de déterminer les risques de surcharge lors de la procédure de rétablissement en aval. Une deuxième étape permet la réparation du composant défectueux et le rétablissement du réseau à sa configuration originale.
La ligne (10) indique le moment où l'agent responsable du calcul des indices de performances doit s'exécuter. Dans notre cas, il intervient à la fin de la procédure de rétablissement.
La ligne (11) informe sur les objectifs finaux à atteindre et oriente le processus de planification.

Le modèle de coordination du tableau 7.2.7 est représenté selon le diagramme de transition de la figure 7.2). Dans ce diagramme, nous introduisons une limite quant au nombre maximum d'exécutions de l'agent réparation ($i < 2$). Cette contrainte n'apparait pas dans le fichier de coordination, elle est déduite indirectement par le fait que l'agent Réparation nécessite au maximum deux étapes pour réaliser la réparation : la première étape est une restauration en amont et la deuxième étape est une restauration en aval, qui nécessite les valeurs électriques fournies par l'agent Aprem. Ce diagramme de transition ou workflow est exécuté en boucle pour simuler une période de temps continue. En effet, le système s'appuie sur les résultats de l'itération précédente pour réaliser la simulation courante.

Figure 7.2 Boucle principale du diagramme de transition pour l'étude de fiabilité



7.3 Étude de fiabilité

7.3.1 La méthode Monte-Carlo

Afin d'évaluer la fiabilité du réseau de distribution, nous utilisons la méthode Monte-Carlo. Cette approche permet de calculer, à l'aide d'une succession de simulations basées sur des paramètres aléatoires, les indices de fiabilité du réseau. L'intérêt de la méthode Monte-Carlo réside dans la possibilité de modéliser des phénomènes aléatoires dans des systèmes complexes, sans avoir à développer des fonctions analytiques, ce qui serait en général impossible. Cependant, cette approche nécessite de réaliser un nombre important de simulations pour permettre la convergence des données vers une moyenne statistique.

La méthode utilisée pour déterminer les indices de performances du réseau est la suivante : nous simulons le réseau électrique pendant une certaine période de temps (par exemple 100

ans), avec des charges fixes et nous générons des pannes aléatoires à différentes étapes de temps. Chaque panne enclenche la simulation de la procédure de rétablissement du courant du réseau tel que montré à la figure 7.2. Une durée de non-alimentation est affectée aux clients du réseau selon la localisation de la panne et la stratégie de restauration. Dès la réparation de la panne, l'agent responsable du calcul des indices de performance collecte les données et calcule de manière incrémentale le CAIDI, le SAIDI et le SAIFI du système. Cette opération est réitérée autant que nécessaire jusqu'à convergence de la moyenne statistique.

À titre d'exemple, le graphique suivant 7.3 expose les résultats obtenus lors de la simulation d'un réseau de distribution arbitraire composée de 20 éléments. Ce dernier a nécessité la simulation de plus de 300 pannes sur une période de 250 années avec des pas de temps de 60 min. Le choix du pas de temps de 60mn vise à tenir compte des cas où des pannes peuvent de produire consécutivements et influencer le processus de réparation. Toutefois, vu la rareté de cet événement, il est possible d'augmenter le pas de temps de 60 mn et ainsi permettre l'optimisation du traitement du simulateur.

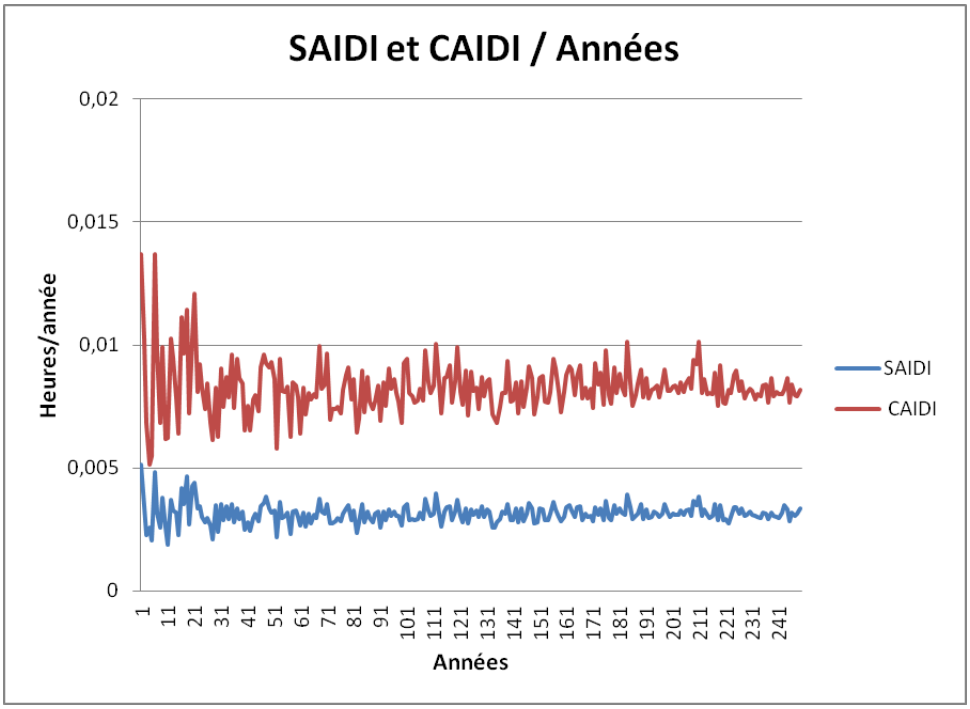


Figure 7.3 Variation du CAIDI et du SAIDI par années de simulation

Il en ressort que pour pouvoir simuler des réseaux de grande envergure à l'image de ceux d'Hydro-Québec, il faut disposer d'un système performant et robuste capable de réaliser des millions de simulations. En effet, plus le réseau à évaluer est grand, plus le calcul des indices de performance nécessite un grand nombre de simulations.

7.3.2 Étude de fiabilité d'un véritable réseau de distribution

Nous avons réalisé de véritables études de fiabilité avec l'outils Leopar++ sur des exemples réels de réseaux de distribution (voir Figure 7.4). Cette figure représente une portion du réseau de distribution à simuler où les lignes blanches sont des câbles triphasés. Les couleurs verte, bleue et rouge représentent des câbles monophasés. Le réseau simulé est composé de 477 composants et quatre départs de lignes. Il comporte 72 sectionneurs, 11 interrupteurs, 343 branches, 145 charges et dessert 17 000 clients. La tension de la ligne est de 24kV et est triphasée.

Nous avons simulé une période de 250 ans avec des étapes de temps de 60 min, ce qui revient à simuler plus de deux millions de fois le réseau. Les probabilités de pannes utilisées pour la simulation étaient de $\lambda = 0,035$ pannes par années pour les appareils de sectionnements et de $\lambda = 0,007$ pannes par années pour les câbles. Pour cette étude, nous avons fait varier le nombre d'équipes d'intervention et la durée de restauration du courant. Cette dernière représente le temps mis par l'équipe pour restaurer en aval les sections saines du réseau. Voici les résultats obtenus :

Figure 7.4 Exemple de réseau distribution d'Hydro-Québec

Scénario 1 :

Une équipe d'intervention et la durée de la restauration est $1h : 00mn$.
SAIDI = 1.890 heures par année
CAIDI = 10.007 heures par année
SAIFI = 0.18886
Nombre de pannes générées= 741
La simulation a duré 74.326 secondes

Scénario 2 :

Deux équipes d'intervention et la durée de la restauration est de $1h : 00mn$.
SAIDI = 1.786 heures par année
CAIDI = 10.003 heures par année
SAIFI = 0.18885
Nombre de pannes générées = 741 [1]
La simulation a duré 73.671 secondes

Scénario 3 :

Une équipe d'intervention et la durée de la restauration est de $0h : 06mn$.
SAIDI = 0.793 heure par année
CAIDI = 7.235 heures par année
SAIFI = 0.1096
Nombre de pannes générées = 741
La simulation a duré 77.737 secondes

Scénario 4 :

Deux équipes d'intervention et la restauration est de $0h : 06mn$.
SAIDI = 0.709 heure par année
CAIDI = 7.231 heures par année
SAIFI = 0.1091
Nombre de pannes générées= 741 [2]

1. Le nombre de pannes est identique car nous utilisons une graine aléatoire identique
2. Le nombre de pannes est identique car nous utilisons une graine aléatoire identique

La simulation a duré 77.825 secondes

À partir de ces résultats il est possible de conclure que la durée de la restauration influence significativement les performances du réseau de distribution et permet de réduire les durées de non-alimentation. Par l'intégration de nouveaux composants comme des interrupteurs télécommandés, il est possible de réduire les temps de restauration. La simulation du cas 3 démontre que les durées de non-alimentation diminuent d'environ 60 % quand la restauration en aval est quasi-instantanée (le SAIDI passe de 1,890 à 0,793).

Dans le cas présent, le nombre d'équipes d'intervention a un impact très limité dans la réduction des durées de non-alimentation. En effet, augmenter le nombre d'équipes permet de réduire de 6 % les durées de non-alimentation (le SAIDI passe de 1,890 à 1,786). Toutefois, il existe une relation entre la fréquence des pannes, le nombre des composants du réseau et le nombre d'équipes d'intervention. Les simulations sur de larges réseaux ont démontré que pour un grand réseau, avec des probabilités de pannes élevées et des délais de restaurations importants, il y a lieu de disposer d'un nombre d'équipes d'intervention élevé. Ceci s'explique par le fait que plusieurs pannes peuvent survenir en même temps, nécessitant l'intervention de deux ou plusieurs équipes simultanément. Le graphique 7.5 expose l'évolution du SAIDI d'un large réseaux composé de 1022 composants. Il apparait évident que quand le nombre de pannes augmente, le SAIDI change en fonction du nombre d'équipes d'intervention.

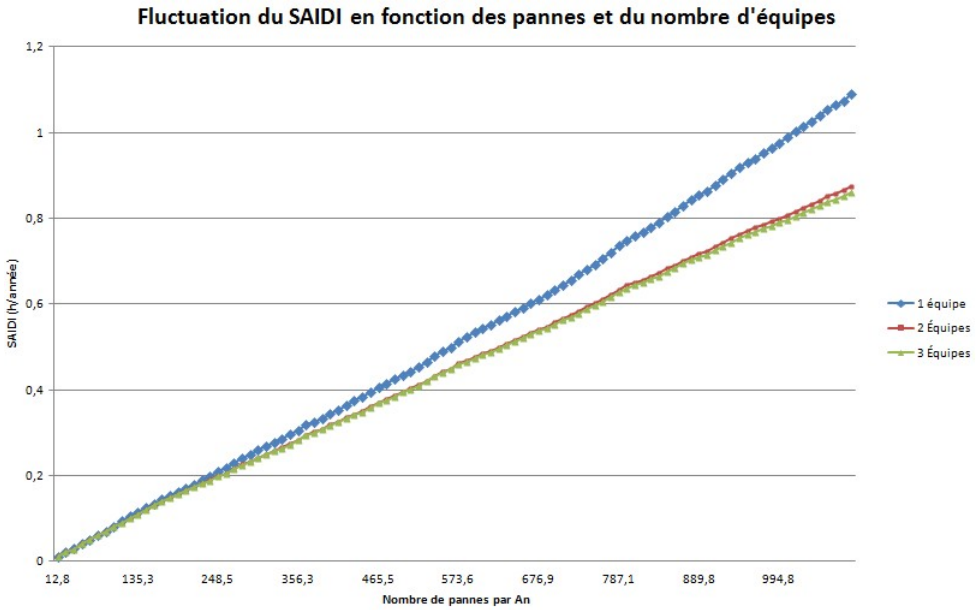


Figure 7.5 Fluctuation du SAIDI en fonction des pannes et du nombre d'équipes

7.4 Évaluation du système Leopar++

Le système Leopar++ est en mesure de réaliser des simulations de véritables réseaux de distribution. Toutefois, il est important de comparer notre architecture à celle de système multiagents.

Ainsi, nous avons développé une version multiagents standard de notre simulateur Leopar++. Pour ce faire, nous avons ré-implémenté les agents de Leopar++ dans l'environnement multiagents Jade. Par la suite, nous avons établi un mécanisme de communication inter-agents basé sur le langage FIPA et l'ontologie CIM. Finalement, nous avons remplacé la zone commune d'échange des données par des mémoire individuelles où chaque agent dispose de sa propre représentation du problème. Le tableau 7.2 expose les différences conceptuelles entre les deux simulateurs.

Tableau 7.2 Comparaison de Leopar++ et Jade

	Leopar++	**Jade**
Coordination Agents	Langage A^c_{k-ext}	FIPA-SL
Communication agents	Tableau noir	Communication asynchrone
Raisonnement agents	Système expert et code Java	Code java
Modèle de données	CIM v14	CIM v14
Nombre d'agents	5	5
Données de simulation	Distribuées	Distribuées

Pour résumer, il existe trois différences entre les deux systèmes. Premièrement, le mode de communication est plus simple sous Leopar++. Cela est dû à l'utilisation du tableau noir comme principal moyen de communication. De plus, le volume des données échangées entre les agents est moins important lorsqu'il y a utilisation du tableau noir. Dans le cas de Jade, il était important de s'assurer que tous les agents mettent à jour leurs données. Deuxièmement, le protocole de coordination est centralisé sous Leopar++ et il est distribué dans les agents sous Jade. Le fait que le mécanisme de coordination soit distribué rend la modification des objectifs globaux du système plus difficile. En effet, à l'aide de A^c_{k-ext}, il est beaucoup plus facile de spécifier des objectifs différents et d'obtenir des plans d'exécutions correspondants. En ce qui concerne la troisième différence, elle réside dans la manière de concevoir le mécanisme de coordination des agents du système. En effet, sous Jade, l'ajout, la modification ou la suppression d'un nouvel agent nécessite de connaitre les autres agents

du système, leur protocole de communication et leurs types d'interactions. Dans le cas de la version Leopar++, l'ajout d'un nouvel agent nécessite de connaitre les données sur lesquels agit l'agent. Cette propriété rend plus aisée l'introduction, la modification et la suppression de nouveaux agents du système.

À la figure 7.6, nous comparons les performances des deux systèmes en termes de temps d'exécution pour des simulations Monte-Carlo s'étendant sur une période allant de 1 an à 100 ans. Le réseau testé est composé de 1022 composants.

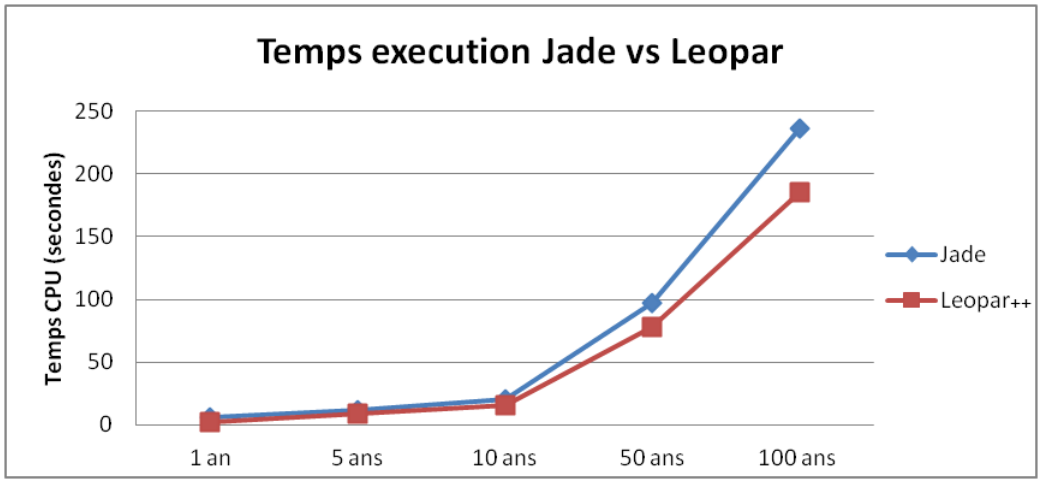


Figure 7.6 Comparatif des performances globales de Jade et Leopar++

La figure 7.7 compare les deux systèmes en termes de temps d'exécution pour des simulations avec un nombre d'agents croissant. Les simulations débutent avec un seul agent et atteignent un maximum de 45 agents. Le réseau testé est composé de 1022 composants.

Nous en concluant que Leopar++ est aussi performant que le système développé en Jade. Pour des simulations s'étalant sur plusieurs années, Leopar++ surpasse légèrement Jade. Toutefois, la grande différence entre les deux systèmes se retrouve dans l'architecture :

1. Leopar++ offre une architecture accessible. La coordination entre les agents est centralisée, elle se fait à l'aide d'un langage de haut niveau et il devient aisé d'intégrer de nouveaux agents.
2. La programmation des agents est plus aisée sur Leopar++ que Jade et plus généralement les systèmes multiagents classiques. En effet, les agents de Leopar++ ne communiquent pas directement entre eux, mais via la modification du bassin central de données qu'est le tableau noir.

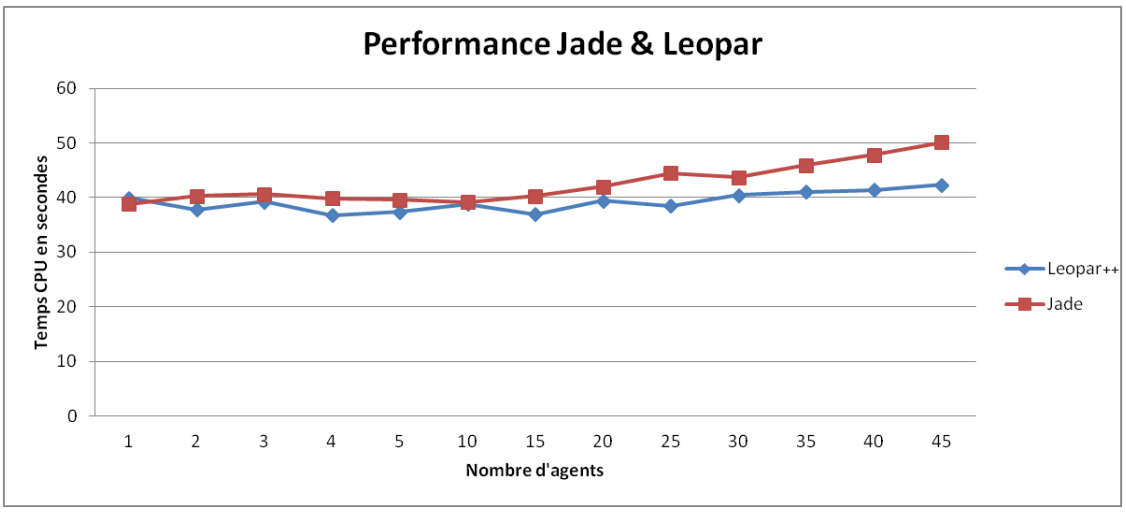


Figure 7.7 Comparatif des performances des agents de Jade et Leopar++

Pour conclure ce chapitre, nous avons comparé notre système Leopar++ à des systèmes de simulation électrique récents. Le tableau 7.3 présente un récapitulatif et une vue de haut niveau des architectures étudiées. Pour connaitre les détails de ces systèmes, nous invitons le lecteur à se référer à la partie annexe A.

Tableau 7.3 Comparaison de systèmes de simulation électrique

Simulateur	Domaine	Mode coordination	Environnement	Test
RMC	Micro-réseau	FIPA-ACL	Jade	Cas réel
CM	Micro-réseau	Multi-niveau	Jade	Cas réel
FACTS	Source d'énergie	Agents typés	Jade	Cas simple
IADSRP	Micro réseau	BDI	Jade	Cas simple
BSR	Isolation panne	FIPA-ACL	Zeus	Cas simple
COMMA	Transformateur	SMA multiple	–	–

Une première lecture de ce tableau fait ressortir que les systèmes de simulation électrique adoptent généralement l'architecture Jade. Conséquemment, les modèles de coordination sont complexes et nécessitent des connaissances avancées en systèmes multiagents, ce qui rend délicat toute extension future du système. Dans le cas de Leopar++, l'utilisation du langage

d'actions permet une coordination plus aisée. En effet, une comparaison qualitative du code de coordination de Jade et de Leopar++ révèle que ce dernier offre un mécanisme de coordination plus compacte et beaucoup plus compréhensible. Le tableau suivant 7.4 compare deux portions de codes identique, l'un développé sous Jade selon Fipa-ACL et l'autre avec les langages d'actions. Dans le cas présent, il s'agit d'un exemple simple où l'agent reçoit un message, le traite et par la suite répond à son interlocuteur.

Tableau 7.4 Code de coordination Leopar++ et Jade

Système	Code
Leopar++	causes(reparation, Reparer, ¬ Reparer).

Jade	msg = receive() ;// reception msg
	if (msg !=null){ //tester msg non vide
	if(msg.getPerformative() == ACLMessage.INFORM){
	if(msg.getConversationId().equals("reparation")){//déterminer type msg
	Network netState = startFailureRestoration() ; // procédure réparation
	reply = msg.createReply() ; // création réponse
	reply.addReceiver(msg.getSender()) ; // spécifier destinataire
	reply.setPerformative(ACLMessage.AGREE) ;//spécifier type msg
	reply.setContent(netState) ;// spécifier contenu message
	myAgent.send(reply) ; // envoie message expéditeur
	} } } // fin procédure réparation

7.5 Fonctionnement du simulateur Leopar++

Actuellement, le simulateur Leopar++ est opérationnel et fonctionne parfaitement. Il est en mesure de simuler différents aspects du réseau de distribution et permet de faire varier le nombre d'équipes d'intervention, le temps de restauration en amont et en aval, la stratégie de restauration, la fréquence de pannes des composants, etc.

Notre système offre une grande souplesse quant à l'ajout de nouveaux agents. En effet, le système est incrémental et permet d'intégrer de nouveaux agents sans avoir à modifier le reste du système. Il tire profit des ontologies et des langages d'actions pour offrir une grande souplesse d'utilisation et d'extension.

Nous avons développé une couche graphique au-dessus du simulateur Leopar++ pour permettre sa manipulation et sa configuration. Pour ce faire, la fenêtre de la Figure 8.1 permet de spécifier la base de données sur laquelle va se connecter D2RQ. Ces informations vont servir au *mappage* pour récupérer les données ontologiques à partir de différentes tables

relationnelles. À partir de l'ontologie extraite, l'utilisateur est en mesure d'identifier le poste et le numéro de la ligne électrique à simuler.

Dans la fenêtre de simulation principale (voir Figure 7.8), l'utilisateur introduit les données relatives à la simulation de pannes. Par exemple, il peut modifier le nombre d'équipes d'intervention, les probabilités de pannes, la stratégie de restauration, la durée de la simulation et les durées de restaurations. Ces informations vont être prises en compte par l'agent Panne.

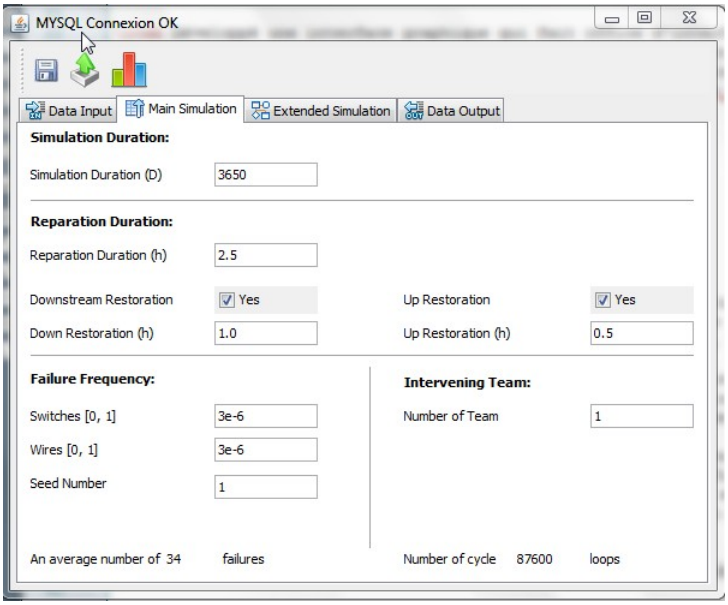


Figure 7.8 Fenêtre de simulation principale de Leopar++

Dans la fenêtre de simulation ASP, l'utilisateur peut spécifier l'état initial du système ainsi que les objectifs à atteindre par le coordinateur ASP (voir Figure 7.9). Cette fenêtre offre la possibilité à l'utilisateur de spécifier le fichier ASP en entrée ainsi que le solutionneur ASP à associer. À partir des états initiaux et les objectifs finaux, le solutionneur ASP est capable d'identifier les actions à exécuter pour l'atteinte de ses objectifs.

Dans la fenêtre résultats (voir Figure 7.10), Leopar++ affiche les indices de performances du réseau simulé. Il est aussi possible de visualiser l'évolution des indices de performances. Pour ce faire, la fenêtre *courbes et statistiques* permet de visualiser l'évolution des indices à travers les différentes étapes de la simulation (voir Figure 7.11). Ces information sont utiles pour déterminer quand se produit la convergence statistique des indices de performance.

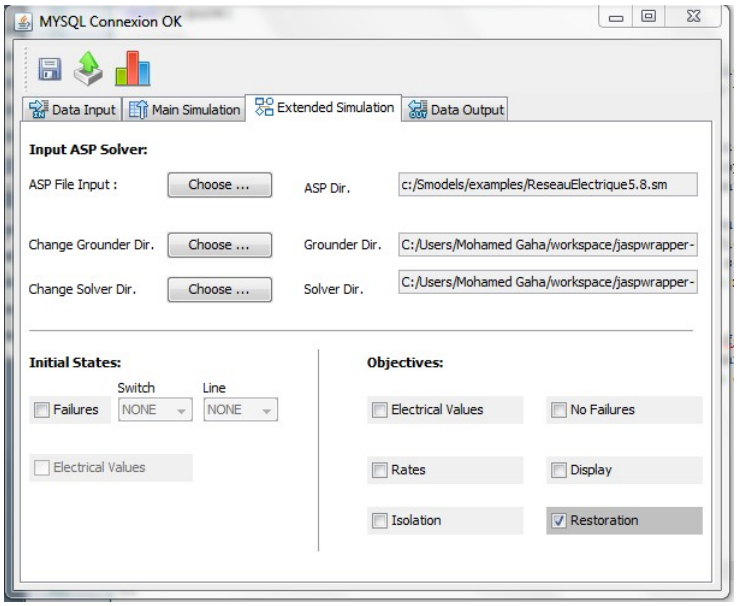


Figure 7.9 Fenêtre de simulation ASP de Leopar++

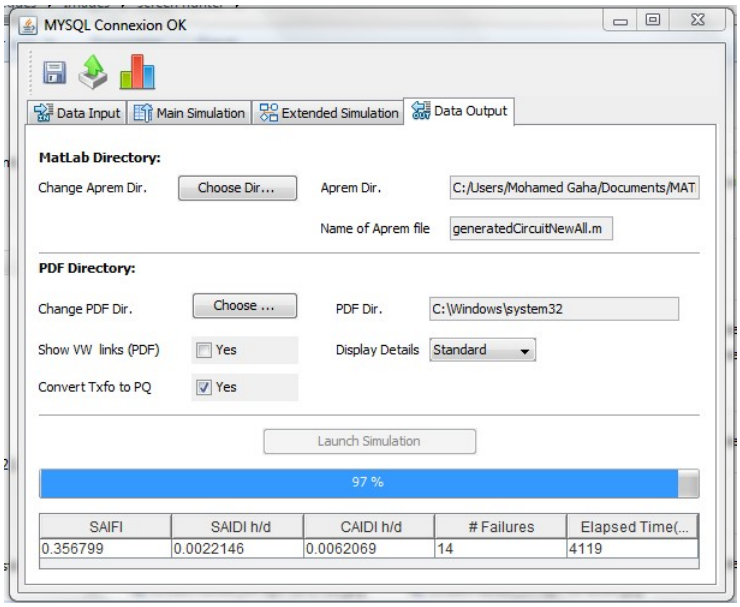


Figure 7.10 Fenêtre des résultats Leopar++

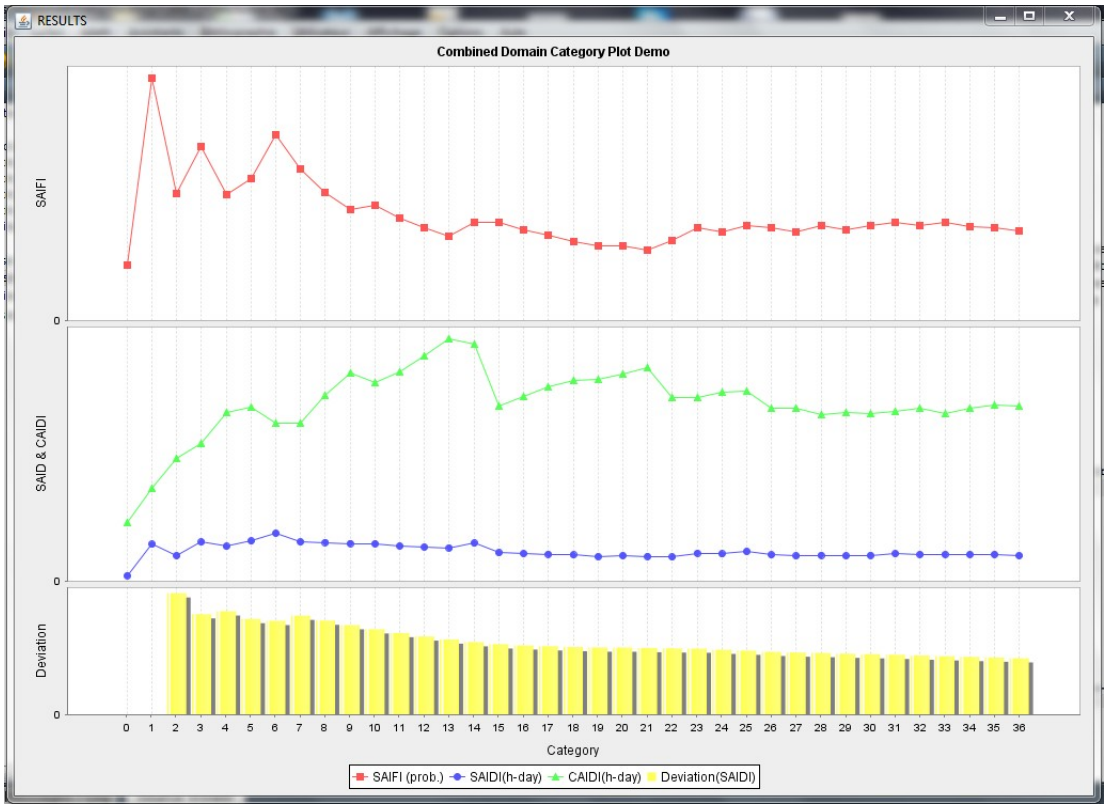


Figure 7.11 Fenêtre de l'évolution des courbes de Leopar++

CHAPITRE 8

DISCUSSION ET CONCLUSION

8.1 Synthèse des travaux

Tout au long de ce travail, nous avions comme principal objectif de développer un système multiagents performant et accessible à des non-experts en informatique. En effet, la contrainte d'accessibilité et de facilité d'utilisation fut au centre de nos préoccupations. Le choix de l'architecture, des formalismes et de la technologie ont été longuement étudiés pour offrir un simulateur qui répond à ces besoins.

Notre plus grand défi fut d'identifier et d'adopter des technologies pour développer un simulateur polyvalent, performant, capable de réaliser des études complexes et qui permet à des non-experts d'en modifier le fonctionnement. Pour développer Leopar++ nous avons :

1. Utilisé l'ontologie OWL2-RL et le raisonneur Pellet pour fédérer et vérifier automatiquement différentes sources de données. Cette étape a permis de collecter les données nécessaires au processus de simulation et d'en vérifier l'intégrité.
2. Défini un langage d'actions pour permettre la coordination des agents du système Leopar++. Le nouveau langage que nous proposons est spécialement adapté aux systèmes multiagents à base de tableau noir. Notre langage permet de coordonner efficacement les agents tout en offrant un formalisme accessible aux non-experts.
3. Combiné deux architectures collaboratives. Notre système multiagents à base de tableau noir tire avantages des deux systèmes. En effet, la centralisation des données permet de faciliter le développement et la coordination des agents et le module de coordination permet de guider et d'orienter les agents vers la résolution d'un problème donné.
4. Élaboré des études de fiabilité à l'aide de la méthode Monte-Carlo. Le système Leopar++ est en mesure de simuler plusieurs phénomènes électriques et mécaniques. Il permet d'évaluer la fiabilité du réseau de distribution selon différents indices de performance.
5. Développé une interface graphique qui fait office d'interface entre le système Leopar++ et l'utilisateur final. L'interface permet à l'utilisateur de sélectionner graphiquement l'état initial, les objectifs à atteindre et le réseau électrique sur lequel doit s'appliquer la simulation. Tout est fait de manière à faciliter la manipulation des différents paramètres de Leopar++ (voir Figure 8.1).

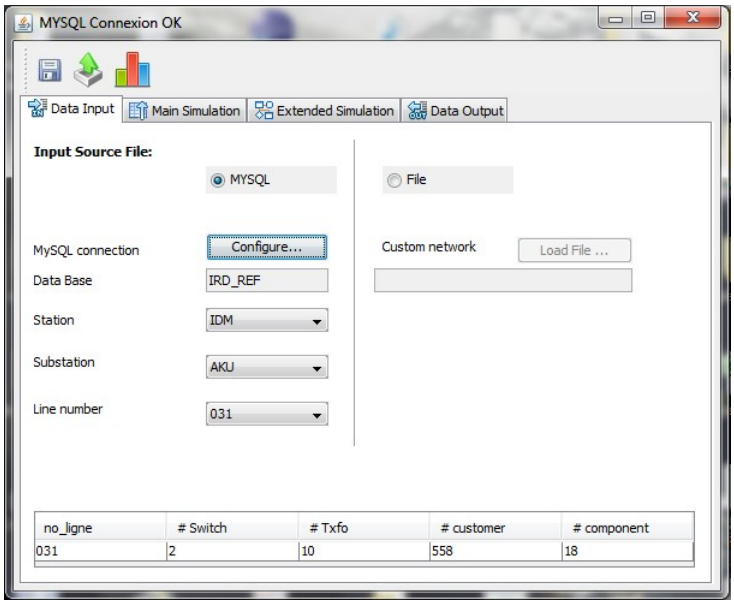


Figure 8.1 Fenêtre principale de Leopar++

8.2 Limitations de la solution proposée

Certaines des limites techniques de ce présent travail sont relativement facile à corriger et nécessitent un temps supplémentaire pour compléter et finaliser certains aspects liés à la programmation. D'autres limites nécessite la réalisation d'un travail de fond qui fait appel à une meilleure compréhension de certains aspects de l'architecture de Leopar++. Toutefois, au-delà des manquements techniques, le problème de l'utilisabilité reste une question ouverte.

1. À quel point l'architecture que nous proposons est facile d'utilisation pour des non spécialistes en système multi-agents ?
2. Dans quelle mesure notre nouvelle architecture facilite et accélère le temps de développement de systèmes multi-agents ?

Pour répondre à ces questions il faut quantifier les critères d'utilisabilté et d'accessibilité du système que nous proposons. C'est à partir de ces critères qualitatifs qu'il est possible d'amméliorer notre architecture et de la rendre viable.

8.3 Travaux futurs

Comme travaux futurs, nous citons les suivantes :

1. Enrichir le langage A^c_{k-ext} et y intégrer de nouvelles règles. Par exemple, il est possible

d'introduire des opérateurs temporels pour définir les sous-objectifs. Aussi, l'introduction de concepts plus élaborés pour exprimer des contraintes de parallélisme est tout aussi important.

2. Étudier les langages d'actions et les langages orientés agents. Une étude détaillées des deux types de langages permettrait de mieux comprendre les avantages et limites du langage A^c_{k-ext}. Le langage A^c_{k-ext} est à mi-chemin entre entre les langages d'actions, trop simples pour exprimer les besoins des agents, et les langages orientés agents, trop complexes pour des non-initiés.
3. Optimiser le processus de lecture des données de l'ontologie. Il serait préférable de charger uniquement les données nécessaires à la simulation. Il faut éviter de charger tout le réseau au complet comme c'est le cas actuellement pour le système Leopar++. En mettant en relation les agents et les données nécessaires à la simulation, il devient possible d'optimiser la phase de lecture des données de l'ontologie.
4. Réaliser une comparaison exhaustive entre les langages d'action et les langages de planification tels que ADL, PDDL et STRIPS (voir Kalisch et Knig, 2005). Les langages de planification offrent des fonctionnalités riches qui sont utiles au langage A^c_{k-ext}. Tirer profit de certaines fonctionnalités des langages de planification et les formaliser dans A^c_{k-ext} serait intéressant. Par exemple le fait de pouvoir spécifier des objectifs conjonctifs en langage ADL. Cette fonctionnalité peut être aisément traduite en langage A^c_{k-ext} et ASP.
5. Offrir à l'utilisateur un environnement dynamique pour introduire les paramètres de la simulation. Il faudrait qu'en fonction des agents à exécuter, l'interface de saisie de paramètre de Leopar++ change dynamiquement. Autrement dit, selon les objectifs, les paramètres de la simulation à introduire changent.

8.4 Conclusion

Le Smart-Grid apporte des bouleversements technologiques importants qui visent à moderniser le réseau pour le rendre plus *intelligent*. Le réseau se transforme d'un organe statique et figé en un organe dynamique et réactif. Collecter, analyser, simuler, fusionner, modéliser, vérifier et interpréter les données des nouveaux réseaux électriques deviennent un enjeu stratégique. Face à ces nouveaux défis, les sciences informatiques sont en mesure d'apporter des éléments de réponses pour accompagner cette transformation.

Or, aider les gestionnaires dans leur prise de décision nécessite de disposer de programmes informatique qui se doivent d'être **polyvalents**, **performants** et **faciles** d'utilisation. La polyvalence est indispensable, car le futur simulateur doit être capable de représenter des phé-

nomènes aléatoires, de modéliser des aspects électriques et de simuler des aspects temporels évolués. La simulation se doit d'être la plus réaliste possible afin de reproduire le plus adéquatement les phénomènes étudiés et performante, car la simulation de réseaux électriques nécessite de modéliser et de traiter des milliers de composants et de phénomènes dépendants, qui peuvent alourdir considérablement les temps de traitement. Finalement, comme les simulateurs sont destinés à être utilisés par des non-informaticiens, il y a un réel effort à réaliser afin de rendre ces systèmes accessibles et compréhensibles par des non-experts en informatique.

Ces caractéristiques de polyvalence, performance et accessibilité, font en sorte que le développement de simulateurs pour le domaine des réseaux électriques reste une opération complexe qui impose des choix technologiques particuliers. Le système Leopar++ s'inscrit dans cette logique et a cherché à satisfaire ces 3 spécificités.

Toute au long de ce travail, il m'a été permis d'explorer de nouvelles avenues et pistes pour apporter des solutions concrètes à des problématiques réelles. Ainsi, j'ai été amené à analyser le standard électrique CIM et à y réaliser des changements conceptuels pour le rendre beaucoup plus compacte. Ces modifications ont nécessité une compréhension poussée du standard CIM pour en maitriser les forces et faiblesse et y apporter les rectifications nécessaires. À l'aide de la version allégée du CIM il m'a été permis de créer une base de connaissance électrique consistante et distribuée.

Par ailleurs, mon travail m'a poussé à explorer des architectures nouvelles et de concevoir un nouveau type de simulateur à mis chemin entre les tableaux noirs et les systèmes multiagents classiques. Ainsi, j'ai pu démontrer qu'il était possible à l'aide des langages d'actions et des outils de représentation des connaissances de développer un simulateur à base de connaissances performant, accessible et fortement extensible.

Finalement, j'ai été en mesure d'analyser les langages orientés agents et de proposer une alternative viable pour coordonner les agents. J'ai exploré un nouveaux langage de coordination des agents, à savoir le langage d'actions A^c_{k-ext}. Ils représentent une alternative prometteuse dans la coordination des agents, car ils utilisent une approche intuitive qui se base sur les actions et leurs effets. J'ai pu démontrer qu'avec certaines extensions spécifiques, et un mécanisme de réinsertion des faits et actions, il devient possible de planifier et coordonner des agents. Cette expérimentation est une première dans le domaine des système multiagents à base de langages d'actions.

Pour conclure, au regard de l'évolution des réseaux électriques, il est fort probable que durant les prochaines années, il y aura une plus grande interaction entre les sciences informatiques et électriques. Cette interaction va permettre de développer des réseaux électriques performants, réactifs, capables de minimiser l'impact des pannes, de se reconfigurer automa-

tiquement, d'acheminer plus efficacement l'énergie, de faire face à une demande croissante d'énergie, etc.

Dans un monde où l'énergie représente la colonne vertébrale de notre société, il devient primordiale de disposer d'une énergie fiable et à portée de main. Grâce à des systèmes de simulation performants et fortement extensible il devient possible d'analyser, de modéliser et de concevoir des réseaux de distributions à forte valeur ajoutée.

RÉFÉRENCES

ALFERES, J. et PEREIRA, L. (1996). *Reasoning With Logic Programming*. No. ne 1111 Lecture Notes in Computer Science. Springer.

ALFERES, J. J., PEREIRA, L. M., PEREIRA, L. M., PRZYMUSINSKA, H. et PRZYMUSINSKI, T. C. (1999). Lups - a language for updating logic programs.

ALIBHAI, Z. et SC, B. A. (2003). What is contract net interaction protocol ? *Contract*.

ALLEMONG, J., BENNON, R. et SELENT, P. (1993). Multiphase power flow solutions using emtp and newtons method. *Power Systems, IEEE Transactions on*, 8, 1455 –1462.

ALOK CHATURVEDI, JAY GORE, S. F. (2006). Society of simulations : An architecture for integrating heterogeneous simulations. *Simulation Conference*.

ARISHA, K., OZCAN, F., ROSS, R., SUBRAHMANIAN, V., EITER, T. et KRAUS, S. (1999). Impact : a platform for collaborating agents. *Intelligent Systems and their Applications, IEEE*, 14, 64 –72.

BALDUCCINI, M. et GELFOND, M. (2003). Diagnostic reasoning with a-prolog. *Theory Pract. Log. Program.*, 3, 425–461.

BARAL, C. (2003). *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA.

BARRINGER, H., FISHER, M., GABBAY, D. M., GOUGH, G. et OWENS, R. (1995). Metatem : An introduction. *Formal Asp. Comput.*, 7, 533–549.

BELLIFEMINE, F. L., CAIRE, G. et GREENWOOD, D. (2007). *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. John Wiley & Sons.

BJORNSON, R. D., CARRIERO, N. et GELERNTER, D. (1997). From weaving threads to untangling the web : A view of coordination from linda's perspective. *COORDINATION*. 1–17.

BORDINI, R. H. et HEBNER, J. F. (2007). A java-based interpreter for an extended version of agentspeak. *Computer and Information Science*, 1.

BROOKS, R. A. (1990). The behavior language ; user's guide. 1227.

CARVER, N. et LESSER, V. (1991). Blackboard-based Sensor Interpretation using a Symbolic Model of the Sources of Uncertainty in Abductive Inferences. *In Proceedings of AAAI 1991 Workshop on Abduction. Also UMass Technical Report*.

CARVER, N. et LESSER, V. (1992). The evolution of blackboard control architectures. Rapport technique.

CATTERSON, V., DAVIDSON, E. et MCARTHUR, S. (2005). Issues in integrating existing multi-agent systems for power engineering applications. *Intelligent Systems Application to Power Systems, 2005. Proceedings of the 13th International Conference on*. 6 pp.

CORKILL, D. (1991). Blackboard Systems. *AI Expert*, 6.

CORKILL, D. D. (2003). Collaborating software : Blackboard and multi-agent systems and the future. *Proceedings of the International Lisp Conference*. New York, New York.

DASTANI, M., RIEMSDIJK, B. V., DIGNUM, F. et JULES MEYER, J. (2003). A programming language for cognitive agents : Goal directed 3apl. Springer, 111–130.

DIMEAS, A. et HATZIARGYRIOU, N. (2005). A mas architecture for microgrids control. *Intelligent Systems Application to Power Systems, 2005. Proceedings of the 13th International Conference on*. 5 pp.

DIMEAS, A. et HATZIARGYRIOU, N. (2009). Control agents for real microgrids. *Intelligent System Applications to Power Systems, 2009. ISAP '09. 15th International Conference on*. 1 –5.

DONG, J., CHEN, S. et JENG, J.-J. (2005). Event-based blackboard architecture for multi-agent systems. *THE PROCEEDINGS OF THE IEEE INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY CODING AND COMPUTING (ITCC*. 379–384.

EITER, T., FABER, W., LEONE, N., PFEIFER, G. et POLLERES, A. (2004). A logic programming approach to knowledge-state planning : Semantics and complexity. *ACM Trans. Comput. Logic*, 5, 206–263.

EITER, T., SCHINDLAUER, R., IANNI, G. et POLLERES, A. (2006). Answer set programming for the semantic web. *TUTORIAL AT 3RD EUROPEAN SEMANTIC WEB CONFERENCE (ESWCe06*.

ENGELMORE, R. et TERRY, A. (1979). Structure and function of the crysalis system. *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 1*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, IJCAI'79, 250–256.

ERMAN, L. D. et LESSER, V. R. (1980). The hearsay-ii speech understanding system : Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12, 213–253.

FISHER, M. (1994). A survey of concurrent metatem - the language and its applications. *Proceedings of the First International Conference on Temporal Logic*. Springer-Verlag, London, UK, UK, ICTL '94, 480–505.

FISHER, M. (2005). Metatem : The story so far. R. H. Bordini, M. Dastani, J. Dix et A. E. Fallah-Seghrouchni, éditeurs, *PROMAS*. Springer, vol. 3862 de *Lecture Notes in Computer Science*, 3–22.

FUJIMOTO, R. (2001). Parallel and distributed simulation systems. *Winter Simulation Conference*, 2, 147–157.

FUJIMOTO, R. M. (1998). Time management in the high level architecture. *Simulation*, 71, 388–400.

GAGNON, M. (2012). *Logique descriptive et OWL*. Polytechnique Montreal.

GAHA, M., GAGNON, M. et SIROIS, F. (2011). A modern blackboard system. *Web Intelligence/IAT Workshops*. 163–166.

GELFOND, M. (2007). *Answer Sets*, Elsevier, chapitre I.

GELFOND, M. et LIFSCHITZ, V. (1993). Representing action and change by logic programs. *Journal of Logic Programming*, 17, 301–322.

GELFOND, M. et LIFSCHITZ, V. (1998). Action languages. *Electronic Transactions on AI*, 3.

HAGER, U., LEHNHOFF, S., REHTANZ, C. et WEDDE, H. (2009). Multi-agent system for coordinated control of facts devices. *Intelligent System Applications to Power Systems, 2009. ISAP '09. 15th International Conference on*. 1 –6.

HAMMAMI, S., MATHKOUR, H. et AL-MOSALLAM, E. A. (2009). A multi-agent architecture for adaptive e-learning systems using a blackboard agent. *Learning*, 184–188.

HEWETT, M. et HEWETT, R. (1993). A language and architecture for efficient blackboard systems. *Artificial Intelligence for Applications, 1993. Proceedings., Ninth Conference on*. 34 –40.

HINDRIKS, K. V., DE BOER, F. S., HOEK, W. V. D. et MEYER, J.-J. (1999). An operational semantics for the single agent core of agent0. Rapport technique.

HUNT, J. (2002). Blackboard architectures. Rapport technique, JayDee Technology Ltd.

KALISCH, M. et KNIG, S. (2005). Comparison of strips, adl and pddl. Rapport technique.

KALYANPUR, A., PASTOR, D. J., BATTLE, S. et PADGET, J. A. (2004). Automatic mapping of owl ontologies into java. F. Maurer et G. Ruhe, éditeurs, *SEKE*. 98–103.

KAO, H.-P., SU, E. et WANG, B. (2002). I2qfd : a blackboard-based multiagent system for supporting concurrent engineering projects. *International Journal of Production Research*, 40, 1235–1262.

KELLETT, A. et FISHER, M. (1997). Concurrent metatem as a coordination language. D. Garlan et D. L. Metayer, éditeurs, *COORDINATION*. Springer, vol. 1282 de *Lecture Notes in Computer Science*, 418–421.

LANGHEIT, C. (2009). Leopar. Presentation, IREQ.

LEITE, J. A., ALFERES, J. J., ALEX, J., LEITE, R., ALFERES, J. J. et PEREIRA, L. M. (2001). Minerva - a dynamic logic programming agent architecture.

LESPERANCE, Y., KELLEY, T. G., MYLOPOULOS, J. et YU, E. S. K. (1999). Modeling dynamic domains with congolog. *In Proceedings of the Eleventh Conference on Advanced Information Systems Engineering (CAiSEe99) (Lecture Notes in Computer Science*. Springer.

LIFSCHITZ, V. (1999). Answer set planning. *ICLP*. 23–37.

LIFSCHITZ, V. et TURNER, H. (1999). Representing transition systems by logic programs. *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, London, UK, UK, LPNMR '99, 92–106.

LO, Y., WANG, C. et LU, C. (2009). A multi-agent based service restoration in distribution network with distributed generations. *Intelligent System Applications to Power Systems, 2009. ISAP '09. 15th International Conference on*. 1 –5.

MAMOUD, Q. H. (2005). Getting started with javaspaces technology : Beyond conventional distributed programming paradigms. Article, Oracle.

MCMORRAN, A. W. (2007). An introduction to iec 61970-301 & 61968-11 : The common information model. Rapport technique Glasgow, UK, University of Strathclyde.

NII, H. P., FEIGENBAUM, E. A., ANTON, J. J. et ROCKMORE, A. J. (1982). Signal-to-symbol transformation : Hasp/siap case study. *AI Magazine*, 3, 23–35.

PAN, Y.-T. et TSAI, M.-S. (2009). Development a bdi-based intelligent agent architecture for distribution systems restoration planning. *Intelligent System Applications to Power Systems, 2009. ISAP '09. 15th International Conference on*. 1 –6.

PINTO, H. S. et MARTINS, J. P. (2004). Ontologies : How can they be built ? *Knowledge and Information Systems*, 6, 441–464.

PIPATTANASOMPORN, M., FEROZE, H. et RAHMAN, S. (2009). Multi-agent systems in a distributed smart grid : Design and implementation. *Power Systems Conference and Exposition, 2009. PSCE '09. IEEE/PES*. 1 –8.

QIANG, S. et LIU, L. (2010). The implement of blackboard-based multi-agent intelligent decision support system. *Proceedings of the 2010 Second International Conference on Computer Engineering and Applications - Volume 01*. IEEE Computer Society, Washington, DC, USA, ICCEA '10, 572–575.

RABIN, S. (2002). *AI Game Programming Wisdom*. Charles River Media, Inc., Rockland, MA, USA.

RAO, A. S. (1996). AgentSpeak(L) : BDI agents speak out in a logical computable language. W. Van de Velde et J. W. Perram, éditeurs, *Agents Breaking Away*, Springer, vol. 1038 de *LNCS*. 42–55. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 Janvier 1996, Proceedings.

RATHNAM, T. (2004). Using ontologies to support interoperability in federated simulation.

ROSS, R., COLLIER, R. W. et O'HARE, G. M. P. (2004). Af-apl : Bridging principles & practices in agent oriented languages.

SCHINDLAUER, R. (2008). *Naswer Set Programming for the Semantic Web*. Verlag Muller.

SEARLE, J. R. (1969). *Speech Acts : An Essay in the Philosophy of Language*. Cambridge University Press, Cambridge.

SHOHAM, Y. (1991). *AGENT0 : A simple agent language and its interpreter*, AAAI Press/MIT Press, vol. 2. 704–709.

SIMARD, G. (2008). Plan d'evolution du reseau de distribution. Presentation, IREQ.

SON, T. C., BARAL, C., TRAN, N. et MCILRAITH, S. (2006). Domain-dependent knowledge in answer set planning. *ACM Trans. Comput. Logic*, 7, 613–657.

STONE, P. et VELOSO, M. (1999). Team-partitioned, opaque-transition reinforcement learning. *Proceedings of the third annual conference on Autonomous Agents*. ACM, New York, NY, USA, AGENTS '99, 206–212.

SUBRAHMANIAN, V. S., BONATTI, P., DIX, J., EITER, T., KRAUS, S., OZCAN, F. et ROSS, R. (2000). Heterogeneous agent systems.

TANG, C. K. F. et TERNOVSKA, E. (2007). Model checking abstract state machines with answer set programming. *Fundam. Inf.*, 77, 105–141.

TINNEY, W. et HART, C. (1967). Power flow solution by newton's method. *Power Apparatus and Systems, IEEE Transactions on*, PAS-86, 1449 –1460.

TU, P. H., SON, T. C. et BARAL, C. (2007). Reasoning and planning with sensing actions, incomplete information, and static causal laws using answer set programming. *Theory Pract. Log. Program.*, 7, 377–450.

USCHOLD, M. et KING, M. (1995). Towards a methodology for building ontologies. *Proc. of IJCAI95's WS on Basic Ontological Issues in Knowledge Sharing*. Montreal, Canada.

VLASSIS, N. (2007). *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Morgan and Claypool Publishers, première édition.

WOOLDRIDGE, M. (2009). *An Introduction to MultiAgent Systems*. Wiley Publishing, seconde édition.

ZHU, T., LIU, G. et JIA, L. (2010). A cooperative making multi-agent model on railway daily dispatching plan based on blackboard. *Digital Manufacturing and Automation, International Conference on*, 1, 18–21.

ANNEXE A

Description des systèmes de simulation électriques

Le BSR SMA : Le système multiagents Based Service Restoration (BSR) est utilisé afin de surveiller, superviser, contrôler, détecter, isoler et restaurer une panne. Les agents du système sont capables de communiquer, de résoudre, de coordonner et de débattre avec les autres agents afin de prendre des décisions. Dès qu'un agent s'exécute, il exécute une procédure afin de restaurer une panne. Le système a été développé sous l'environnement JADE (Java Agent DEvelopment Framework), et les agents utilisent un langage de communication et de transmission de la connaissance FIPA-ACL. Les performances du système ont été comparées à un système monolithique (voir Lo *et al.*, 2009), et les résultats ont montré que le système multiagents est deux fois plus rapide qu'un système monolithique classique. Les auteurs ont conclu que le BSR a accéléré la détection de panne ainsi que l'isolation et la restauration du service.

Le RMC SMA : Dans le même ordre d'idée, le système multiagents du Real Microgrid Control (RMC), utilise des agents afin de simuler différents aspects des opérations du Microgrids. Chaque agent contrôle une ou plusieurs charges non critiques afin d'atteindre un objectif particulier comme minimiser la consommation électrique ou la gestion de la durée de vie de la batterie.

Tout comme le BSR, les interactions et par conséquent la coordination des agents sont directement encodés à l'aide du FIPA-ACL protocole. Les agents sont développés sous la plateforme JADE et un test réel a été réalisé pour 11 maisons durant trois jours. Dimeas (voir Dimeas et Hatziargyriou, 2005) a conclu que les systèmes multiagents permettent de contrôler les opérations du micro Grid et ont prouvé que se sont des outils qui limitent la complexité des opérations de contrôle.

Le IDAPS SMA : Le Intelligent Distributed Autonomous Power System (IDAPS) a été développé pour sécuriser un réseau de distribution lors d'une panne. Le réseau de distribution a la particularité d'intégrer plusieurs producteurs d'énergies à des charges PQ. Les agents du système fonctionnent de manière collaborative afin de détecter les pannes et de les isoler de manière autonome. L'architecture de IDAPS, contrairement au BSR et RMS, offre une meilleure structuration des agents. Le système est composé de quatre types d'agents : des agents de contrôle qui supervisent le voltage et la fréquence de la tension, des agents res-

ponsables de la sauvegarde des données électriques, des agents qui affichent les propriétés du système à l'utilisateur et finalement des agents qui sauvegardent les données du système dans une base de données. Le système qui utilise l'environnement ZEUS et le langage FIPA-ACL pour coordonner les agents a été testé sur un réseau de distribution simplifié composé par un nombre réduit de composants électriques. Il a été en mesure de réagir adéquatement en déconnectant et en stabilisant le réseau durant une panne (voir Pipattanasomporn *et al.*, 2009).

Le CM SMA : Tout comme le IDAPS, le système multiagents pour le Contrôle de Microgrid (CM) identifie distinctement le rôle de chaque agent. Le système qui a pour objectif de contrôler et d'opérer un réseau Microgrid, est bâti sur une technique de niveaux multiples d'apprentissage issu du domaine de la robotique (voir Stone et Veloso, 1999). Cette méthode permet d'organiser et d'organiser les interactions dans un système complexe. Le système CM est composé de trois niveaux, où chaque niveau renferme un ensemble d'agents regroupés par comportement et interactions. Le plus haut niveau renferme les agents au comportement complexe et vice-versa. Ce système est basé sur la plateforme JADE et utilise le protocole de communication FIPA-ACL. Le système CM a été développé afin d'offrir un système extensible et standardisé (voir Dimeas et Hatziargyriou, 2009).

Le FACTS SMA : De récents systèmes tels que le Coordinated Control of a Flexible Alternative Current Transmission Systems (FACTS) visent à contrôler l'impact des sources d'énergie renouvelable dans un réseau de distribution. Le système utilise des agents distribués pour contrôler le fonctionnement du réseau. L'idée est de répartir la complexité du module de contrôle en sous-modules élémentaires afin d'améliorer les performances du système. Le système a recours à deux types d'agents : l'agent *non-controllable* soumet, à travers la topologie du réseau, des messages relatifs aux charges, aux impédances et à l'état électrique des composants. L'agent *controllable* embarqué dans chaque Power Flow Controllers (PFC) récupérer analyse et évalue les messages transmis aux gestionnaires de flux électrique. Le système est basé sur une approche innovante de contrôle distribué où les agents soumettent des messages à leurs voisins à travers la configuration du réseau. L'influence de chaque agent contrôlable est limitée à une zone spécifique correspondante au PFC. Le système décrit n'a pas de structure d'organisation hiérarchique ou de module de supervision, ce sont les agents qui traitent localement les informations qui sont transmises à travers le réseau (voir Hager *et al.*, 2009). Une topologie simplifiée avec cinq bus trois PFC et une charge a été testée avec succès.

Le ADSRP SMA : L'Architecture pour le Distribution Systems Restoration Planning (ADSRP) a pour objectif d'isoler automatiquement une panne et de reconfigurer le réseau électrique pour réalimenter les clients. Ce système multiagents de restauration est basé sur l'architecture BDI (croyance – désir– intention) qui apporte une meilleure compréhension des agents et de leurs interactions. En effet, les agents sont capables de réaliser des rétablissements électriques dépendamment de leurs connaissances, de leurs objectifs, de leurs intentions et de leurs croyances. En plus, un ensemble de règles permettent de guider le système durant le rétablissement de la panne. Tout comme FACTS, la communication entre les agents se fait grâce à la transmission d'informations de proche en proche à travers la structure électrique du système de distribution. Le système a été développé en JADE et un test grandeur nature sur quatre sources, 11 charges, trois disjoncteurs et sept interrupteurs a été réalisé avec succès (voir Pan et Tsai, 2009).

Les systèmes IADSRP et FACTS sont innovants et explorent de nouvelles voies, toutefois elles présentent l'inconvénient d'offrir des systèmes non performants. En effet, simuler un large réseau de distribution implique la création de nombreux agents ainsi que la transmission d'un nombre élevé de messages redondants à travers le réseau résultant en une diminution de la performance du système en entier.

Le COMMAS : Le Condition Monitoring Multiagents Systems (COMMAS) explore la possibilité de fusionner deux Systems multiagents afin d'atteindre une simulation de plus, haut niveau. Le COMSMA est spécialisé dans l'étude de la décharge partielle des transformateurs électriques. Il a été enrichi par un autre système multiagents : Protection Engineering Diagnostic Agents (PEDA) qui a pour objectif d'automatiser la détection de panne en s'appuyant sur un SCADA et un enregistreur numérique de panne (Digital Fault Recorder DFR). L'intégration du PEDA et du COMSMA offre aux ingénieurs un système nouveau qui combine la supervision des données issues du SCÀDÀ et du DFR. Ce nouveau système utilise une ontologie et le langage FIPA-SL pour transmettre les messages entre les deux systèmes multiagents. L'auteur a été en mesure de fusionner deux SMA et ainsi offrir un système d'aide à la décision de haut niveau aux ingénieurs (voir Catterson *et al.*, 2005).

ANNEXE B

Description détaillée des langages orientés agents

MINERVA : Ce langage utilise la programmation logique de commande LUPS (voir Alferes *et al.*, 1999) qui spécifie de manière déclarative un programme logique afin de décrire les caractéristiques de l'agent. Le langage LUPS offre la possibilité d'insérer ou de retirer des règles logiques dans la base de règles de l'agent. Sous Minerva il existe les agents spécialisés dans la détection des changements des données de l'environnement, la communication entre les agents, la mise à jour des données de manières cycliques, la planification par abduction pour déterminer les actions à entreprendre, le gestionnaire d'objectif responsable de traiter les conflits d'objectifs (voir Leite *et al.*, 2001).

3APL : C'est un langage qui permet l'implémentation d'agents avec états mentaux. Les agents, dits cognitifs, disposent de Croyances, Intentions et Objectifs et sont en mesure de générer et de modifier leurs plans afin de satisfaire les objectifs pour lesquels ils sont créés. Les agents disposent d'un ensemble de règles de premier ordre (semblable à Prolog) pour permettre la prise de décision. Le langage 3APL est muni d'une plateforme de développement permettant le support de l'exécution du système multiagents (voir Dastani *et al.*, 2003).

IMPACT : Le (Interactive Maryland Platform for Agents Collaborating Together) est une plateforme pour la création distribuée de systèmes multiagents. Les agents sont composés de deux parties : un ensemble de données destinées à être manipulées par l'agent et un ensemble de fonctions qui permettent de manipuler les données. Il existe différents types de composants : un mécanisme décrivant les services offerts par l'agent, une boite de messages permettant la communication, l'état actuel de l'agent, les actions réalisables par l'agent, un langage de requête permettant de raisonner sur les données, un mécanisme permettant d'exécuter des actions concurrences, un mécanisme de contrôle de contraintes d'intégrités et finalement le programme agent qui informe sur les objectifs de l'agent (voir Subrahmanian *et al.*, 2000) (voir Arisha *et al.*, 1999).

Jason : C'est une implémentation Java du langage AgentSpeak(L). Le langage est basé sur le langage abstrait AgentSpeak(L) (voir Rao, 1996) qui permet de décrire un agent avec des états mentaux BDI. Le langage AgentSpeak(L) est bâti sur la programmation logique des prédicats de premier ordre pour décrire les états mentaux de l'agent. Le mode de raisonnement

de AgentSpeak(L) est défini comme étant un raisonnement réactif. La plateforme Jason tire sa force de sa capacité à interpréter le langage AgentSpeak(L) (voir Bordini et Hebner, 2007) .

Agent0 : C'est un langage qui permet de représenter des agents avec états mentaux. Le langage Agent0 est en mesure de décrire les actions de l'agent, ses croyances et ses intentions. Le langage de programmation Agent0 intègre la possibilité de décrire les croyances des agents, les actions (simple ou conditionnelles), la communication (à travers un sous ensemble de primitives FIPA) et l'engagement pour l'exécution d'une action dépendamment d'une étape de temps (voir Hindriks *et al.*, 1999) (voir Shoham, 1991).

Concurent MetateM : C'est un langage de programmation qui se base sur de la logique modale temporelle. Le langage offre la possibilité de décrire des comportements réactifs, des comportements déductifs de haut niveau et de décrire des processus d'exécution concurrentiels pour des systèmes distribués. Le langage Concurrent MetateM offre un large éventail d'opérateur (au nombre de 11) pour décrire les événements temporels. Ce type de langage est utile pour les applications qui nécessitent de représenter les contraintes temporelles exigeantes (voir Barringer *et al.*, 1995) (voir Fisher, 1994) (voir Kellett et Fisher, 1997).

ConGolog : L'environnement ConGolog est un langage de haut niveau qui permet de décrire le comportement des agents informatiques et robotiques. Le langage ConGolog supporte le raisonnement logique pour inférer les actions des agents qui sont spécifiés selon une approche déclarative selon le langage du domaine Golog. Ce dernier bien qu'issu du langage d'actions de Gelfond et Lifschitz's A (voir Gelfond et Lifschitz, 1998) la différence entre ces langages est que le Golog est un langage de premier ordre tandis que le langage d'actions est un langage propositionnel (voir Lesperance *et al.*, 1999).

AF-APL : Le (AgentFactory Agent Programming Language :)est un langage permet de représenter des agents avec des états mentaux basés sur l'approche BDI. Le langage AF-APL garanti une exécution asynchrone des agents ainsi que la possibilité de l'agent de raisonner sur ses propres actions (voir Ross *et al.*, 2004).

ANNEXE C

L'architecture HLA

Introduction à l'architecture HLA : L'idée de développer des fédérations de simulateurs n'est pas nouvelle et remonte au début des années quatre-vingt-dix. Un effort et un enthousiasme soutenus par les chercheurs ont permis la mise en place d'un standard qu'est la HLA. Ce dernier est une architecture qui décrit la manière dont s'organisent les différents composants d'une fédération de simulateurs. Cette architecture vise à permettre aux développeurs de suivre une démarche spécifique afin de faciliter le développement d'une fédération. Elle offre, en outre, un cadre où les composants, leurs rôles ainsi que les types de messages échangés entre les simulateurs sont identifiés et standardisés (IEEE2000).

Selon Rathnam (voir Rathnam, 2004), la structure HLA se divise en deux parties : une première partie se compose des simulateurs à fédérer et la deuxième partie se compose d'une interface nommée Run Time Infrastructure (RTI). Cette dernière offre des outils et des facilités permettant l'interaction entre les simulateurs fédérés (voir Fujimoto, 2001). La RTI peut se décomposer en trois éléments principaux (i) des règles (ii) des Object Model Template (OMT) et (iii) d'une interface. Ces entités représentent les représentations formelles permettant l'interaction entre les simulateurs fédérés (voir Figure C.1).

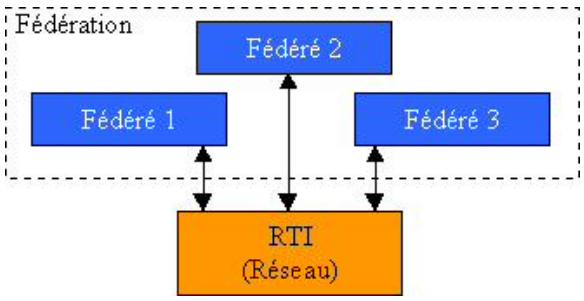


Figure C.1 Vue de haut niveau de HLA

Les règles décrivent les différents éléments d'une fédération et la manière dont elles doivent interagir. Elles permettent de gérer la communication entre les fédérés émetteurs et récepteurs grâce à l'OMT. Les OMT définissent la structure et la représentation de toutes les informations partagées entre les fédérés. L'OMT est défini comme format standard pour décrire la fédération et les fédérés. Finalement, l'interface HLA permet d'établir la communication entre les fédérés et la RTI.

Il existe deux types d'objets OMT à savoir les (Simulation Object Modèle) SOM et le (Federation Object Modèle) FOM. Le FOM est au nombre d'un seul par fédération. Il décrit dans un format standardisé la nature des données communes échangées et partagées entre les fédérés. Il énumère les interactions possibles entre les fédérés à l'aide de plusieurs tables qui décrivent entre autres les champs communs que se partagent les fédérés. Les SOM sont au nombre d'un (1) par simulateur fédéré. Ils spécifient les capacités intrinsèques de la simulation et décrivent les informations qu'un simulateur fédéré peut offrir ou recevoir d'un autre simulateur. Ainsi, un fédéré se verra doté d'un SOM qui décrit les publications et l'identification des messages issus d'autres simulateurs.

En ce qui concerne la gestion temporelle des simulateurs, chaque fédéré représente en interne via le SOM la manière dont il représente le temps. Il existe deux manières de représenter le temps : continue ou discrète. Dans un système de gestion de temps continu, le temps de simulation avance de manière incrémentale et par étape. Une gestion discrète du temps se base uniquement sur les évènements. Le temps de la simulation avance d'un événement à un autre (voir Figure C.2).

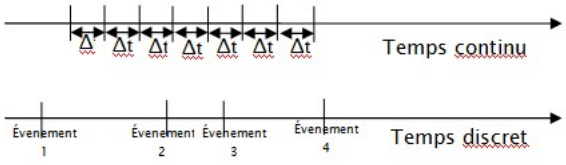


Figure C.2 Exemple de blocage de synchronisation

Dans les architectures HLA, un gestionnaire de temps s'occupe du contrôle de l'avancement du temps dans chaque fédéré de manière à ce que les événements arrivent selon une logique de cause à effet. Le gestionnaire de temps utilise des structures d'organisations temporelles (voir Fujimoto, 1998) : (i) l'ordre de réception des messages et (ii) le timestamp ou étampe temporelle . La première structure qui se présente sous la forme de liste FIFO est utilisée dans le cas d'évènements ponctuels qui dépendent uniquement du temps (ex : à minuit les portes se ferment). Pour l'étampe temporelle, elle est utilisée dans le cas où il faut synchroniser les différents messages reçus de la part des fédérés distribués ou non sur le réseau. Pour éviter que certains fédérés connaissent des dysfonctionnements dus à l'arrivée de message en retard, tous les simulateurs fédérés doivent attendre et s'assurer que tous les messages aient été transmis pour pouvoir avancer au prochain temps δT.

Le rôle de l'étampe temporelle est de vérifier que les événements arrivent en ordre aux simulateurs fédérés. Ainsi, à chaque étape, le RTI reçoit un ensemble de messages TimeStamp Order (TSO). Ces messages sont ordonnés par ordre croissant selon l'étampe temporelle et

délivrés par la suite aux fédérés. Toutefois, ces messages ne sont pas transmis directement. Pour éviter tout risque de réception de message TSO non ordonné, le RTI doit attendre un certain temps nommé Lower Bound Time Stamp (LBTS) avant de trier et de transmettre les TSO et ainsi s'assurer qu'aucun fédéré ne reçoit de message avec une étampe temporelle ultérieure à son temps interne de simulation. Pour résumer, le LBTS est calculé pour chaque fédéré. Il retourne un temps T' représentant l'avance temporelle minimum que chaque fédéré peut réaliser sans avoir le risque de recevoir un $TS < T + T'$. Le calcul du LBTS se fait par le RTI en se basant sur le comportement du réseau ainsi que du lookahead. Il est déterminé pour chaque fédéré. C'est une information transmise par les fédérations de simulateurs qui indiquent qu'aucun message TSO ne sera envoyé dans un temps inférieur à une valeur positive spécifique. Cette information permet de déterminer quand aura lieu la transmission du prochain message TSO de la part d'un simulateur. Cette information est primordiale dans la fédération, sans quoi le processus de simulation connait un temps de latence trop important.