Université de Montréal

Analyse des traces d'exécution pour la vérification des protocoles d'interaction dans les systèmes multiagents

par

Nourchène Ben Ayed

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M. Sc.)
en informatique

Juin 2003

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Analyse des traces d'exécution pour la vérification des protocoles d'interaction dans les systèmes multiagents

présenté par :

Nourchène Ben Ayed

a été évalué par un jury composé des personnes suivantes :

Brigitte Jaumard
(présidente-rapporteure)

Houari Sahraoui
(directeur de maîtrise)

Arnaud Dury
(codirecteur de maîtrise)

Julie Vachon
(membre du jury)

Mémoire accepté le
17 juillet 2003

Sommaire

L'interaction est l'élément essentiel sur lequel reposent les sociétés d'agents pour communiquer et coopérer entre eux. Elle est la source de la dynamique des systèmes multiagents (SMAs). C'est pour cette raison que l'étude des interactions entre agents est devenue un thème essentiel pour s'assurer du bon fonctionnement de tels systèmes. Plusieurs voies de recherche sont actuellement explorées, à travers les méthodologies de modélisation des protocoles d'interactions (logique, réseau de pétri, AUML), la vérification formelle de protocoles, etc. Ce travail de recherche aborde la vérification des protocoles d'interaction en proposant une nouvelle approche basée sur l'utilisation du model-checking.

L'approche élaborée s'applique à tout SMA qui utilise les protocoles comme moyen d'interaction. Elle porte sur l'observation et l'analyse des traces d'exécution pour la vérification des protocoles d'interaction des SMAs dont le but de détecter les erreurs au niveau du protocole et par conséquent d'analyser le comportement dynamique de tout le système.

La première phase consiste à proposer un mécanisme pour l'observation et la collecte de traces dans les SMAs, permettant la prise en compte de la dépendance entre les différents événements invoqués lors des interactions. Ensuite, à modéliser la trace du SMA par un ensemble de processus de communication en SDL. Puis, pour comprendre et analyser le protocole d'interaction, nous modélisons les propriétés recherchées dans les protocoles d'interaction au moyen du langage GOAL. Enfin, nous avons utilisé le simulateur d'objectGEODE pour vérifier l'existence ou non de ces propriétés dans la trace d'exécution du système.

Mots clés : protocoles d'interaction, vérification formelle, model-checking, systèmes multiagents, observation.

Abstract

The multiagent society is based primarily on the concept of agent-based interaction, which is considered as the key element of the agent system dynamics. So far, this concept is still attracting the research community by conducting multiple researches, exploring different ways and using different methodologies. For instance, agent interaction protocol modeling methodologies (logic, petri network, AUML), protocol formal verification, etc. This research proposes a new approach, which is based on the use of the model checking in the interaction protocol verification.

This approach is applied on all the SMA that use protocols as a communication and cooperation tool among agents. It is based on observation and analysis of the SMA interaction protocol verification. The main objective of this verification is to detect the generated errors at the protocol level in order to analyze the behavior of the system dynamics.

The approach is composed of four different steps. The first step is about presenting a mechanism for the observation and collecting SMA traces in order to take into account the dependence among the different calling events during the interaction. The second step emphasizes the modeling of the SMA trace for multiple SDL communication processes. The following step includes the modeling the properties sought in the interaction protocols by using the GOAL language. In the last step, the objectGEODE simulator is used in order to verify the existence of properties in the system execution trace.

Keywords : Interaction protocols, formal verification, model checking, multiagent systems, observation.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

Acronyme	Description	Première utilisation
AUML	Agent Unified Modeling Language	25
BDD	Binary Decision Diagram	41
GOAL	Geode Observation Automata Language	32
IA	Intelligence Artificielle	10
IAD	Intelligence Artificielle Distribuée	10
IBAUTS	Intelligent Building AUTomation Systems	8
MC	Model Checking	4
MCS	Model Checking Symbolique	40
MCT	Model Checking Temporisé	40
MSC	Message Sequence Chart	73
SDL	Specification and Description Language	30
SMA	Système MultiAgents	1
TBL	Temporal Belief Logic	37
UML	Unified Modeling Language	25
XML	Extensible Markup Language	60

Remerciements

Une fois ce travail accompli, je ne pouvais m'empêcher de manifester ma reconnaissance à mes deux directeurs de recherche pour leurs appuis durant mon projet de recherche.

Je tiens à remercier vivement le professeur HOUARI SAHRAOUI, *mon directeur de recherche pour m'avoir offert l'opportunité de travailler au sein de l'équipe* GLIC *du* CRIM, *aussi pour ses directives et son appui financier.*

Je tiens à exprimer mes remerciements à ARNAUD DURY, *mon codirecteur de recherche pour ses conseils et ses directives si enrichissantes. Je lui exprime également ma gratitude pour son soutien moral dans des moments difficiles. Sans lui ce travail n'aurait pas été accompli.*

Je tiens aussi à exprimer mes remerciements à l'équipe ASD, *et spécialement à* HESHAM HALLAL *pour sa collaboration, son aide et surtout pour sa disponibilité.*

Finalement, je remercie toutes les personnes qui ont contribué directement ou indirectement à la réalisation de ce travail.

Dédicaces

Lis, au nom de ton Seigneur qui a créé, qui a créé l'homme d'une adhérence.
Lis! Ton Seigneur est le Très Noble, qui a enseigné par la plume, a enseigné à l'homme ce qu'il ne savait pas.
Les cinq premiers verset du Coran révélés au prophète Mohamed.

À celui qui m'a éclairé le chemin de la science....

Il me tient à coeur de remercier plus profondément mes très chers parents NOURI & NOURA *(N&N) qui ont fait de moi ce que je suis aujourd'hui et qui ont été les premiers à contribuer à ma formation. Je les remercie pour leurs sacrifices tout au long de mes études et surtout pour avoir semé en moi le goût pour la connaissance et le désir de l'acquérir. Que dieu, le tout puissant, leur donne santé et longue vie.*

J'adresse mes remerciements à mon mari GHAZI *pour son amour, sa confiance, sa patience, son soutien incessant et son encouragement durant la rédaction de ce mémoire et tout au long de mon cursus académique canadien. Merci de supporter mes changements d'humeur si fréquents et de m'exhorter à poursuivre mes études.*

Mes remerciements aussi à mes soeurs NOURHÈNE *et* NOUR *et à mon frère* ANOIR, *pour leurs coups de téléphone à 7h30 du matin – décalage horaire oblige –, leurs emails, leurs petits mots d'encouragement et surtout pour l'amour qu'ils portent pour leur aînée.*

Mes remerciements aussi à mes grands parents MOHAMED *et* SAIDA *qui n'ont pas arrêté de veiller sur moi durant toutes ces années, aussi pour l'amour et la tendresse qu'ils n'ont jamais cessé de me donner; à l'esprit de mes grands parents paternels, à ma belle famille spécialement mon beau père* ABDELWAHEB *et à ma belle mère* ZHOUR, *mes belles soeurs, mes gendres, mes tantes, mes oncles, mes cousins et cousines pour les liens très intimes que nous avons établis. Que tous les membres de ma famille trouvent ici la reconnaissance des soutiens qu'ils m'ont toujours apporté.*

Mes remerciements aussi à HICHAM SNOUSSSI *pour sa disponibilité, ses commentaires, ses directives et ses lectures de ce mémoire.*

J'aimerais remercier tous mes amis de l'équipe GLIC, *de l'université et mes amis intimes pour leurs encouragements, leurs conseils, leurs soutiens ou tout simplement pour leur présence :* AMINE, HICHEM, KADDOUR, RIDHA, ZEINEB, KHÉDIJA, SANA, RAHA, GUITTA, MONGIA, HAJER...

Chapitre 1

Introduction et Problématique

Les systèmes multiagents (SMAs) constituent une discipline qui a connu une très forte évolution durant la dernière décennie. Les recherches dans le domaine visaient à doter les SMAs de mécanismes de raisonnement et de comportement au niveau individuel d'un agent et des moyens de communication et d'interaction au niveau du SMA. En outre, ces systèmes commencent à être évolutifs et adaptatifs. De plus, ils sont influencés par des éléments externes comme les phénomènes physiques à contrôler.

Les preuves de concept de ce nouveau paradigme de développement de systèmes se sont succédées au travers de nombreux prototypes. Il y en a parmi ceux-ci qui sont devenus populaires jusqu'à faire partie des programmes des conférences, tels que les tournois de soccer pour robots. En même temps, il y a un constat d'intérêt manifesté et grandissant pour l'utilisation des SMAs dans le cadre d'applications industrielles diverses. Ceci dit, et afin de favoriser une telle application dans des systèmes réels, il faut offrir des garanties quant au bon fonctionnement ou à l'absence d'erreurs surtout s'il s'agit d'une utilisation dans des systèmes critiques.

Ce mémoire est donc l'aboutissement d'une recherche sur la vérification des interactions des SMAs, dont le but d'améliorer la robustesse, la fiabilité et la qualité de ces systèmes. Ce travail est motivé par les spécificités de tels systèmes telles que la répartition, le non-déterminisme, l'autonomie et la concurrence. Il a comme vocation de trouver une approche générique, s'appliquant à tout SMA supportant des protocoles d'interaction comme moyen de communication. Notre objectif est de promouvoir ainsi l'intégration et l'utilisation de tels systèmes dans des secteurs autres qu'académique.

1.1 Problématique

La vérification est une étape primordiale du processus de développement des logiciels. Le temps et l'effort mis pour l'élaboration de cette étape est assez important et varie selon la complexité des systèmes [Katoen, 2002] [Jennings, 1999]. Le but du processus de vérification est de prouver la conformité du système par rapport à sa spécification afin d'atteindre un degré élevé d'exactitude, de fiabilité fonctionnelle et par conséquent réduire les risques causant l'échec des systèmes développés. Notons comme exemples la panne du réseau téléphonique aux États Unis en 1989 ou la destruction de la fusée Ariane 5 en 1996. Ces échecs ont renforcé la nécessité de vérifier les logiciels critiques dont les défaillances peuvent avoir des conséquences désastreuses en termes de vies humaines ou de pertes financières. Cette notion n'est plus aujourd'hui limitée à certains secteurs névralgiques (aérospatial, nucléaire...) mais concerne aussi une multitude de secteurs d'activité (transports, énergie, télécommunications...). L'importance accrue de la vérification des systèmes informatiques, particulièrement les systèmes dont l'exécution est distribuée, est due au développement très rapide de l'utilisation de ces systèmes, de leur taille et de leur complexité.

Les systèmes formés par des composantes qui interagissent entre eux comme les systèmes de communication et les SMAs sont plus vulnérables aux erreurs. En effet, le nombre de ces derniers croît exponentiellement avec le nombre de ces composantes [Katoen, 2002]. De plus et à cause de la complexité, la concurrence, le non-déterminisme et le problème de l'incertitude de l'environnement durant l'exécution des SMAs, le maintien d'une haute qualité, la compréhension ainsi l'analyse de l'activité et du comportement de tels systèmes en utilisant les pratiques de tests existantes sont non suffisantes. D'une part, cette insuffisance est engendrée par le fait que les tests nous garantissent uniquement la présence d'erreurs mais non pas leur absence. Et d'autre part, l'utilisation de la technique du test ne suffit pas toute seule à couvrir toutes les propriétés recherchées dans un SMA telles que les propriétés de vivacité et de sûreté.

Nous nous proposons de traiter les problèmes de la vérification dans le contexte des SMAs, d'étudier les techniques de vérification adaptées à ces systèmes et d'identifier les différents niveaux d'abstraction des propriétés vérifiables.

1.1.1 Définition de la vérification

Concernant le sens de la vérification et comme le montre l'article de [Gonzalez and Barr, 2000], il existe une panoplie de définitions. Nous avons choisi comme fondement à notre recherche, la définition de [Wooldridge and Ciancarini, 2000] qui présentent la vérifi-

cation comme :

> *Un processus consistant à prouver que le système déjà développé est conforme aux spécifications originales, c'est-à-dire qu'il n'existe aucune erreur structurelle et aucune violation des contraintes sémantiques. Ce qui revient à répondre à la question suivante :* ***Am I building the system right?***

En cherchant une définition du processus de vérification, nous avons été confrontés à l'ambiguité qui règne sur les différentes étapes de ce processus. En effet, quelques chercheurs comme [Polat and Güvenir, 1991] voient la vérification comme une étape du processus de validation, alors que d'autres insistent sur le fait que ce sont deux processus différents. Notre avis rejoint celui de [Wooldridge and Ciancarini, 2000] et [Gonzalez and Barr, 2000] qui dissocient le processus de vérification du processus de validation et qui voient la validation comme :

> *Un processus consistant à s'assurer que les résultats d'un système sont équivalents à celle d'un expert humain avec les mêmes entrées. Ce qui revient à répondre à la question suivante :* ***Did I build the right system?***

1.1.2 Techniques formelles de vérification

Ils existent plusieurs techniques de vérification qui diffèrent selon les domaines d'application et les définitions. Les techniques d'analyse des SMAs rejoignent celles développées pour les systèmes distribués puisque l'exécution d'un SMA peut être considérée comme une exécution distribuée. La vérification de ces systèmes est fondée sur l'utilisation de méthodes formelles qui visent à offrir un cadre mathématique permettant de décrire de manière précise et stricte les systèmes et les programmes. De plus, cette vérification formelle permet de démontrer soit une absence du comportement indésirable du système, soit l'existence du comportement souhaitable. Parmi les techniques les plus utilisées, nous trouvons :

- **La génération de tests :** la technique de génération de séquences de tests consiste à prévoir un ensemble de scénarios – *à partir d'une spécification formelle du système et de la propriété à tester* – dans le but d'accroître notre confiance en la sûreté des systèmes et d'améliorer la compréhension du logiciel.
- **La méthode basée sur les preuves :** cette technique conduit la vérification directement sur la structure du programme source, en utilisant des démonstrateurs de théorèmes. Ils existent aussi des outils de démonstrateur de théorèmes (une liste assez complète de ces outils se trouve sur le site www.afm.sbu.ac.uk) qui

sont utilisés pour vérifier un modèle du système et non pas le code source du programme. Dans les deux cas, le principe de cette approche est fondé sur l'axiomatisation. En effet, elle consiste à axiomatiser un modèle ou un code du système, à exprimer les propriétés désirées dans une logique et à montrer, en construisant une preuve dans cette logique, que l'axiomatisation implique les propriétés.

- **Le model-checking :** la technique de model-checking[1] (MC) est une procédure automatique qui permet de vérifier qu'un modèle d'un système – *qui est soit un prototype ou une abstraction du système*– est conforme à sa spécification. Cette procédure de vérification se fait grâce à des algorithmes s'appliquant aux modèles extraits du système, ainsi que de la propriété.
  De point de vue plus formel, le model-checking se résume en : étant donné une formule φ d'un système S et un modèle M de S, déterminer si φ existe dans M, ce qui revient à vérifier si $M \models_S \varphi$ [Wooldridge and Ciancarini, 2000].

Il est à mentionner que ces différentes techniques de vérification sont complémentaires et l'utilisation de l'une n'exclut pas l'autre.

1.1.3 Classification des propriétés vérifiables dans un SMA

Les propriétés à vérifier dans un SMA appartiennent à différents niveaux d'abstraction. Ces niveaux sont soient :

- **Niveau interne :** nous pouvons vérifier des aspects internes aux agents comme les mécanismes de raisonnement et les stratégies adoptées par ces derniers.
- **Niveau externe :** nous pouvons vérifier des aspects externes des agents, c'est-à-dire les interactions entre agents comme les protocoles d'interaction, les mécanismes de coordination, etc.
- **Niveau global :** nous pouvons vérifier des propriétés globales du SMA en faisant une abstraction du système. Citons par exemple l'émergence d'une structure sociale.

Quelle que soit la catégorie de la propriété à vérifier, nous sommes amenés à définir des sémantiques claires et prouvables afin d'envisager des spécifications formelles de SMA dans le but de les vérifier.

[1] Littéralement «vérification de modèle»

1.1.4 Récapitulatif

Dans ce travail, nous nous sommes intéressés à vérifier les SMAs en nous intéressant à l'analyse de leurs protocoles d'interaction (aspects externes aux agents). Ce choix est dû à l'importance du rôle de l'interaction dans la dynamique de tels systèmes. D'ailleurs, l'interaction est considérée comme la composante essentielle sur laquelle reposent les sociétés d'agents. De plus, nous avons tenté de définir une approche de vérification aussi générique que possible, qui suppose que nous avons un accès limité aux connaissances et aux mécanismes de raisonnement des agents.

Nous avons aussi opté pour la technique de model-checking pour la vérification des protocoles d'interaction des SMAs. Les raisons qui nous ont amenés à choisir cette technique sont les suivantes :

- **Par rapport à la technique de démonstration automatique :** la technique du model-checking est plus simple et plus efficace que celle de la méthode basée sur la preuve. Jusqu'à maintenant et à notre connaissance, cette dernière est utilisée uniquement dans le secteur académique et n'a pas été utilisée pour des applications industrielles. De plus cette méthode de preuve connaît plusieurs difficultés :
  - Au début, cette méthode a été développée pour la vérification des programmes séquentiels. L'adaptation de cette dernière aux programmes présentant des aspects parallèles et distribués constitue une tâche assez complexe.
  - L'efficacité de cette technique dépend énormément des compétences de l'utilisateur et de son expertise dans les disciplines mathématiques. De plus, l'automatisation complète n'est pas envisageable puisque nous sommes amenés à établir une preuve pour chaque propriété recherchée dans le système.
- **Par rapport à la technique de génération de tests :** comme pour la technique du model-checking, la génération de tests se heurte au même problème, celui de l'explosion combinatoire. Néanmoins, la technique de génération de tests est confrontée aux problèmes qui concernent la façon de définir un scénario de test ainsi que la façon d'affirmer qu'une exécution de programme a réussi un test.

Le tableau suivant résume les principales différences entre ces techniques de vérification basées sur les méthodes formelles. Les distinctions mises en évidence dans ce tableau se concentrent sur quatre points : la notation utilisée, le type de fonctionnement, la nature des résultats et le domaine d'application de ces techniques.

	Vérificateur de modèles	Démonstrateur de théorèmes	Générateur de tests
Logique	FSM (Finite state machine), logique temporelle	logique de premier ordre et d'ordre supérieur	FSM ou une variante (extended FSM, communicating FSM)
Fonctionnement	automatique	semi-automatique	automatique
Résultats	confirmation de l'absence de la propriété ou contre-exemple (dans le cas de la présence de la propriété)	preuve (dans le cas de succès)	succès, échec et non concluant (les résultats obtenus sont cohérents avec la spécification mais sont différents des résultats attendus)
Application	modèle à état fini	non limité	modèle à état fini et système

TAB. 1.1: Différences entre les techniques de vérification formelle

1.2 Contributions

Les SMAs que nous voulons vérifier utilisent les protocoles comme moyen d'interaction, de coopération, de négociation ou de coordination. L'approche que nous avons développée pour vérifier ces SMAs est basée sur l'utilisation du model-checking. Elle porte sur l'analyse de l'exécution des protocoles d'interaction de ces systèmes dont le but est d'étudier le comportement des agents et de vérifier la conformité du système à sa spécification. C'est une approche par reconnaissance des anti-patterns d'interactions qui se trouveraient dans les modèles extraits de la trace d'exécution. Nous voulons dire par anti-pattern d'interaction toute instance indésirable du protocole d'interaction, impliquant plusieurs agents et présentant les échanges de messages entre agents, le contenu et les contraintes appliquées à ces messages. L'utilisation de ces anti-patterns a pour but de chercher et détecter les erreurs, ainsi que de prouver que les spécifications des systèmes sont bien implantées [Hilaire *et al.*, 2000].

Lors de l'élaboration de cette approche, nous avons été amenés à observer les interactions se déroulant lors de l'exécution du SMA afin de collecter la trace d'exécution. Il s'est avéré que la tâche d'observation des événements inhérents des interactions n'est pas simple. La notion de causalité est au centre de l'observation. En effet, La conformité de l'interprétation de l'ordre des événements à l'ordre réel n'est pas toujours garantie. Par la suite, nous avons cherché un formalisme permettant la modélisation de

cette trace d'exécution. Le choix d'un langage de spécification formel est primordial. Ce langage doit prendre en compte toutes les caractéristiques d'un SMA (répartition, communication, synchronisation...). Ensuite, nous avons identifié les propriétés (anti-patterns) recherchées. Cette identification est une question d'expertise liée au domaine d'application. Certes, il existe plusieurs niveaux d'abstractions des propriétés vérifiables (propriétés liées au comportement interne ou externe des agents et les propriétés globales) qui peuvent être considérées. Le langage utilisé pour la spécification des propriétés dépend du model-checking qui sera utilisé dans la phase d'analyse. Enfin, nous avons utilisé la technique du MC pour vérifier l'absence de ces anti-patterns dans les modèles extraits de la trace d'exécution du système. Une bonne connaissance des bases théoriques de l'outil est nécessaire pour utiliser au mieux toutes ces possibilités.

Il est à mentionner que cette vérification des protocoles d'interaction se fait de façon post-mortem. Ce choix est dû d'une part à la dynamicité[2] et à la répartition[3] des anti-patterns, d'autre part, la vérification post-mortem ne perturbe pas le comportement du système étudié. En effet, les seules perturbations de l'exécution sont celles occasionnées durant la phase de collecte et leur importance peut être limitée. D'ailleurs, la vérification post-mortem permet aussi de contourner le problème des traces partielles qui n'apporteraient pas de résultats immédiats lors de l'analyse et d'effectuer des traitements coûteux de l'information de trace sans perturber l'exécution.

1.3 Démarche pour traiter la problématique

À travers cette étude, nous essaierons de trouver des approches pour atteindre nos objectifs :

- **Observer l'exécution d'un SMA :** il s'agit de trouver une méthode pour observer le comportement externe des agents en l'appréhendant à travers leurs interactions. Cette méthode doit préserver l'ordre causal des événements associés aux interactions. De plus, elle doit proposer un mécanisme de collecte de ces événements observés et choisir un langage de représentation pour ces derniers.
- **Modélisation de la trace collectée et des propriétés recherchées :** le formalisme de spécification doit répondre à différents facteurs :
  - la facilité et la souplesse du langage choisi pour couvrir toutes les caractéristiques d'un SMA,

[2] nous parlons d'anti-pattern dynamique lorsque ces anti-patterns s'étalent sur un enchaînement d'événements, c'est à dire sur plusieurs pas de temps

[3] nous parlons d'anti-pattern répartis s'ils couvrent différents processus ou agents

  - la puissance du langage pour couvrir et exprimer la majorité des propriétés comportementales identifiées,
  - la disponibilité d'un outil supportant le ou les langages choisis.

- **Choix d'un outil de model-checking :** après le choix du langage de modélisation de la trace et de la formalisation des propriétés recherchées, nous devons trouver un outil permettant d'analyser les spécifications exprimées dans ces différents langages ou formalismes.

1.4 Projet IBAUTS

Ce travail s'inscrit dans le cadre du projet IBAUTS (Intelligent Building AUTomation Systems), impliquant le CRIM (Centre de Recherche Informatique de Montréal), l'Université de Montréal, CETC-Varennes (CANMET Energy Technology Center), PRECARN, ainsi que la compagnie industrielle Delta Controls. Le gouvernement Canadien, désireux de respecter le protocole de Kyoto en ce qui concerne la minimisation des émissions de gaz à effet de serre (GES), en particulier dans le domaine des bâtiments, a chargé PRECARN et DELTA CONTROLS d'élaborer un système intelligent en vue d'automatiser le contrôle et l'optimisation du fonctionnement des systèmes de chauffage, de ventilation et de climatisation des édifices. En plus de sa capacité à prendre des décisions qui minimiseront les besoins énergétiques, ce système doit assurer un niveau de satisfaction optimale aux occupants en ce qui a trait à la température ambiante. En outre, il est prévu que le système mette en oeuvre des techniques de détection et de diagnostic des anomalies, en temps opportun, afin de réduire les incidences négatives de ces dernières sur la consommation énergétique et sur la qualité de l'air ambiant.

L'objectif de ce projet est de proposer un prototype de contrôleur multiagent pour apporter une solution plus flexible, plus économique quant à l'automatisation intelligente des dispositifs de chauffage et en réduisant de 30% la consommation d'énergie au niveau des bâtiments. Sachant que les GES produits par le chauffage des bâtiments comptent pour une proportion de 15% des GES produits dans tout le Canada, ce projet permettrait de réduire de 5% les émissions totales de GES.

Notre travail apporte au projet IBAUTS un mécanisme permettant la détection des erreurs protocolaires et applicatives afin de garantir la cohérence du système développé et sa conformité à sa spécification.

1.5 Plan du mémoire

Nous commençons notre étude par une revue de la littérature sur les concepts des systèmes multiagents, des communications inter-agents et des interactions. Nous poursuivons par un troisième chapitre qui survole les différents formalismes et langages de spécification et de description des systèmes multiagents et des propriétés recherchées dans de tels systèmes. Ensuite, nous proposons dans le chapitre suivant une étude des différentes approches de vérification basées sur les modèles et une synthèse de ces différents travaux afin de tirer des conclusions pour s'en servir lors de l'élaboration de notre approche. Subséquemment, nous présentons notre approche pour la vérification de ces SMAs. Comme elle a nécessité un mécanisme d'observation pour l'instrumentation de la trace d'exécution des SMAs, nous étudions les mécanismes d'observation dans les systèmes distribués. Aussi, nous montrons la faisabilité de ces mécanismes pour l'observation des SMAs et nous présentons l'approche que nous avons élaborée pour l'instrumentation de la trace d'exécution des SMAs tout en préservant l'ordre des événements. Dans le même chapitre, nous donnons une classification des propriétés à vérifier et nous expliquons le mécanisme de fonctionnement de l'outil. Enfin, nous testons notre approche sur un système multiagent industriel (le SMA IBAUTS). Cette mise en application couvre les différentes étapes de collecte et de modélisation de la trace d'exécution, l'identification et la formalisation des anti-patterns et une analyse des résultats. Nous terminons cette étude par une conclusion et une série d'orientations pour les travaux à venir.

Chapitre 2

Introduction aux systèmes multiagents

La thématique de l'intelligence artificielle distribuée (IAD) est à l'intersection des domaines de l'intelligence artificielle (IA), du génie logiciel et des systèmes distribués. Le domaine de l'IAD est une extension du domaine de l'IA. Par opposition à l'IA qui traite les problèmes de la modélisation du comportement intelligent d'un seul agent ainsi que l'étude des comportements internes, des stratégies et des mécanismes de raisonnement de ces agents ; l'IAD soulève et traite les problèmes liés à la coopération, à la collaboration et à la négociation de plusieurs agents ainsi que les méthodes et les techniques qui permettent de construire une organisation de société d'agents.
Le domaine des systèmes multiagents est un sous axe de l'IAD qui vient pour traiter le problème de l'intelligence et la résolution des problèmes de façon distribuée. Les SMAs sont caractérisés en principe par la répartition, le parallélisme, la concurrence et le non-déterminisme. Tous ces aspects font de l'observation, de l'analyse et de la vérification des SMAs des tâches difficiles dont les objectifs sont complexes.

Nous aborderons dans ce chapitre les notions d'agents et de SMA. Puis nous détaillerons les différents aspects des SMAs comme les interactions, la communication, les protocoles de coopération, de négociation et de coordination. Enfin nous présenterons brièvement la problématique de la vérification des SMAs.

2.1 Notion d'agent

Le concept d'agent a suscité l'intérêt de plusieurs chercheurs durant plusieurs décennies. Il a émergé dans différents domaines comme les bases de connaissances, la robotique,

l'IA, les réseaux de communication notamment l'Internet, etc. Jusqu'à date, il n'existe pas une définition unique d'un agent, chacune est influencée par le domaine d'application. Selon [Gasser *et al.*, 1987] :

> *Un agent est défini comme un objet actif qui communique en utilisant des messages. Il a des buts à réaliser, possède de la connaissance, peut percevoir son environnement et effectue des actions.*

Néanmoins, [Ferber, 1995] a essayé de dégager une définition minimale et commune :

> *Un agent peut être défini comme une entité physique ou virtuelle capable d'agir sur elle même et son environnement, disposant d'une représentation partielle de cet environnement, pouvant communiquer avec d'autres agents et dont le comportement est la conséquence de ses observations, de sa connaissance et des interactions avec les autres agents.*

Selon ces définitions, nous pouvons situer les trois dimensions caractérisant un agent : ses buts, ses capacités à réaliser certaines tâches et enfin les ressources dont il dispose. Cependant, les capacités des agents varient selon leurs granularités et leurs buts changent selon leurs évolutions dans l'environnement et leurs interactions avec les autres agents. Mais, nous pouvons conclure que le degré d'implication de ces trois dimensions diffèrent d'un agent à un autre et dépend de la catégorie de l'agent. Les spécialistes des SMAs établissent deux catégories d'agents : cognitifs et réactifs. Cette distinction, établie entre ces deux écoles, se fonde sur le fait que les agents les plus compétents, ayant des raisonnements évolués, auront des interactions plus complexes et plus puissantes.

2.1.1 Agents réactifs

Un des pionniers de l'IA réactive est Brooks. Selon lui, le comportement intelligent devrait émerger de l'interaction entre divers comportements plus simples
[Jarras and Chaib-draa, 2002]. Par conséquent, il n'est pas nécessaire que chaque agent soit individuellement «*intelligent*» pour parvenir à un comportement global intelligent. L'organisation émergente de la fourmilière est un exemple de système réactif.
Un agent réactif est donc un agent très simple de faible granularité ne possédant pas de représentation explicite de son environnement, et dont le comportement est primitif et ne consiste en général qu'à répondre à la loi stimulus/réaction. Il ne dispose que d'un langage et d'un protocole de communication réduit.

2.1.2 Agents cognitifs

En revanche, les agents cognitifs sont considérés comme des agents individuellement «*intelligent*». Les systèmes ou agents cognitifs comprennent un ensemble de connaissances et d'aptitudes qui leur permettent de comprendre, de traiter des informations applicatives (dépendante du domaine d'application) ainsi que des informations relatives à la gestion des interactions avec les autres agents et d'acquérir des connaissances sur l'environnement externe (perception).

En fonction des connaissances et des croyances dont il dispose et des buts qu'il se fixe suite à une perception ou à une interaction avec l'environnement extérieur, l'agent produit un plan d'action. Pour cela, il décide du but à retenir et à satisfaire en premier, il planifie en fonction de ce but et il passe à l'exécution.

Le tableau 2.1 ci-dessous résume les différences entre les agents cognitifs et réactifs.

Système d'agents cognitifs	**Système d'agents réactifs**
Représentation explicite de l'environnement et des autres agents	Pas de représentation explicite
Peut tenir compte de son passé et dispose d'un but explicite	Pas de mémoire de son historique, ni de but explicite
Agents complexes (communication par envoi de messages)	Comportement de type Stimulus/réaction
Petit nombre d'agents de forte granularité	Grand nombre d'agents de faible granularité
Mode «*social*» d'organisation (planification...)	Mode «*biologique*» d'organisation
Les relations entre agents s'établissent en fonction des collaborations nécessaires à la résolution du problème	La structure du système émerge des comportements et non pas d'une volonté d'organisation

TAB. 2.1: Différences entre agents cognitifs et réactifs

2.2 Les systèmes multiagents

Que ceux-ci soient des agent réactifs ou cognitifs, nous pouvons dire que la compétence d'un agent pris individuellement est limitée. Cependant, en les regroupant dans des sociétés d'agents, nous pouvons acquérir plus de compétences et de connaissances ; ce que nous appelons systèmes multiagents. Un SMA est un système distribué composé par

un ensemble d'agents qui sont en général concurrents. Contrairement aux systèmes d'IA, qui simulent dans une certaine mesure les capacités du raisonnement humain, les SMAs sont conçus et implantés comme un ensemble d'agents interagissant, le plus souvent, selon des modes de coopération, de concurrence ou de coexistence pour résoudre un problème particulier. Ils se caractérisent par leur capacité à réaliser leurs buts à travers la communication ou la perception.
Plusieurs typologies des SMAs existent : on parle des SMAs ouverts lorsque les agents se déplacent librement ; des SMAs fermés lorsque l'ensemble des agents ne change pas ; des SMAs homogènes, si tous les agents sont construits sur le même modèle (exemple une colonie de la même espèce de fourmis) ; et des SMAs hétérogènes, si les agents ont des modèles différents et sont de granularité différente (exemple l'organisation hospitalière). Malgré ces quelques différences, un SMA, il est toujours constitué des éléments suivants [Ferber, 1995] :

- Un environnement (espace disposant ou non d'une métrique),
- Un ensemble d'objets situés (il est possible, à un moment donné, de leur associer une position dans l'environnement). Ils sont passifs et peuvent être perçus, créés, détruits et modifiés par les agents,
- Un ensemble d'agents capables de percevoir, produire, consommer, transformer et manipuler les objets de l'environnement
- Un ensemble de relations (communications) unissant les agents

Cette définition met en évidence le thème de base des SMAs, celui de l'action. Les SMAs se caractérisent, en effet, par une modélisation du problème à résoudre en termes d'environnement, et une modélisation des étapes de résolution en terme d'actions dans cet environnement [Dury, 2000].

2.3 Interactions

Ce qui fait la différence entre un SMA et une collection d'agents indépendants est le fait que ces derniers interagissent pour aboutir à un but commun ou accomplir une tâche spécifique. Or selon [Ferber, 1995], pour un agent, l'interaction constitue à la fois la source de sa puissance et l'origine de ses problèmes. En effet, l'interaction est considérée comme :

- la source de sa puissance puisqu'elle dote l'agent d'une intelligence accrue qui lui permet de satisfaire l'objectif global du système,

- et l'origine de ses problèmes puisque :
  - il faut intégrer une quantité significative de connaissances au sein d'un agent pour qu'il puisse interagir avec les autres. De plus, il faut intégrer au sein des agents tout un mécanisme de communication,
  - et que ces interactions peuvent induire à des situations de conflit.

Les interactions sont généralement définies comme une mise en relation dynamique de deux ou de plusieurs agents par le biais d'un ensemble d'actions. Cette mise en relation dépend de la nature des agents (cognitif ou réactif) et se fait par une communication directe ou en agissant sur l'environnement ou via un autre agent. L'interaction est non seulement la conséquence d'actions effectuées par plusieurs agents en même temps, mais aussi l'élément nécessaire à la constitution de sociétés d'agents. D'ailleurs, elle est considérée comme une composante essentielle de la dynamique des SMAs. Elle offre aux agents la possibilité d'atteindre le but global du système et par conséquent dote tout le système d'un comportement intelligent quel que soit le degré de complexité des agents qui le composent.
Pour les systèmes réactifs, les agents ont une vue à court terme et ont chacun un but individuel, par conséquent ils entrent en conflit pour des ressources et parfois n'interagissent pas. Par contre, les agents cognitifs participent à la satisfaction du but global du système en coopérant avec les autres agents mais tout en poursuivant leurs objectifs individuels. Ils initient des interactions plus intentionnelles.

Quelle que soit la nature des agents, l'établissement des interactions se fait par un échange direct d'informations entre les agents, c'est-à-dire par communication, ou via l'environnement, ce que nous appelons perception. La différence entre la perception et la communication est distinguée par la manière à appréhender le changement du comportement des agents. En effet, la perception implique qu'un agent a effectué une action et la connaissance du changement de l'état de l'agent est perçue par le changement de l'environnement. Par contre, la communication implique un échange direct de messages entre agents et la connaissance du changement de comportement de l'agent se fait par le changement de son état.

Il est à mentionner que la communication est l'un des moyens les plus utilisés dans les SMAs pour établir une interaction entre les agents et que tout au long de ce chapitre, nous présenterons juste les interactions basées sur la communication. Dans ce qui suit nous décrivons un panorama des différentes classes d'interaction.

2.3.1 Classification des interactions

Afin de définir et classifier les différents types d'interaction, une connaissance des situations d'interaction s'avère nécessaire. [Ferber, 1995] définit les situations d'interaction comme suit :

> *On appellera situation d'interaction un ensemble de comportements résultant du regroupement d'agents qui doivent agir pour satisfaire leurs objectifs en tenant compte des contraintes provenant des ressources plus ou moins limitées dont ils disposent et de leurs compétences individuelles.*

En général, les situations d'interaction sont classées en tenant compte de ces trois critères:

- Les objectifs des agents. Lors de l'interaction des agents, ces derniers peuvent avoir des buts contradictoires. Cette distinction permet de donner une première classification qui est la suivante : les agents, dont les buts sont compatibles, sont dans une situation de coopération ou d'indifférence; dans le cas contraire, ils sont dans une situation d'antagonisme.
- Les relations que les agents ont avec les ressources. Les ressources sont les éléments environnementaux et matériels utiles à la réalisation d'une action. En général, ces ressources sont limitées et les différents agents peuvent avoir besoin des mêmes ressources, en même temps et au même endroit ce qui conduit à des conflits entre les agents. On parlera alors d'interactions perturbatrices. Ces situations peuvent être résolues par un mécanisme de coordination d'actions –*qui anticipe les conflits et fait en sorte qu'ils soient gérés avant qu'ils se manifestent* –, et un mécanisme de résolution de conflits comme les concessions, la loi du plus fort ou les techniques de négociation.
- Les compétences des agents pour satisfaire leurs buts se résument dans la possibilité qu'un agent a, pour réaliser individuellement une tâche ou son incapacité à l'accomplir. S'il n'est pas capable de l'accomplir, il y aura une nécessité d'avoir de l'aide d'autres agents.

Comme le montre le tableau 2.2 [Ferber, 1995], ces trois critères de classification des interactions permettent de faire une typologie des situations d'interaction. Cette classification s'intéresse à la vision externe des comportements des agents et permet de situer le comportement d'un agent et expliquer sa façon de percevoir l'environnement. De plus, le tableau montre l'effet des différentes caractéristiques et moyens mis à la disposition des agents (but, ressources et compétences) pour définir et classifier leurs interactions.

Buts	Ressources	Compétences	Situation	Catégorie
Compatibles	Suffisantes	Suffisantes	Indépendance	Indifférence
Compatibles	Suffisantes	Insuffisantes	Collaboration simple	Coopération
Compatibles	Insuffisantes	Suffisantes	Encombrement	Coopération
Compatibles	Insuffisantes	Insuffisantes	Collaboration coordonnée	Coopération
Incompatibles	Suffisantes	Suffisantes	Compétition individuelle pure	Antagonisme
Incompatibles	Suffisantes	Insuffisantes	Compétition collective pure	Antagonisme
Incompatibles	Insuffisantes	Suffisantes	Conflits individuels pour des ressources	Antagonisme
Incompatibles	Insuffisantes	Insuffisantes	Conflits collectifs pour des ressources	Antagonisme

TAB. 2.2: Classification de situations d'interactions

D'autres classifications existent pour les situations d'interactions afin de définir le comportement des agents. Parmi ces classifications, nous trouvons celle de [Martial, 1992], qui décrit les interactions en termes de relations entre les plans d'agents. Il les classe en deux catégories : les interactions positives et négatives. Les interactions positives sont celles qui sont bénéfiques au comportement des agents, elles peuvent induire des coopérations. Elles sont au nombre de trois : relations d'égalité (equality), de faveur (favor) et d'inclusion (subsumption).

- Relations d'égalité : deux plans d'agents contiennent la même action et le fait de l'exécuter par l'un ou l'autre ne modifie pas le résultat.
- Relation de faveur : l'accomplissement des actions d'un plan entraîne ou favorise l'accomplissement des actions d'un autre plan.
- Relation d'inclusion : l'accomplissement des actions d'un plan implique l'accomplissement de certaines actions d'un autre plan.

Cependant, les interactions négatives traduisent les conflits entre agents. En effet, ce sont l'ensemble de relations qui empêchent ou gênent l'accomplissement simultané de plusieurs actions. Elles sont au nombre de trois : relations d'exclusion, défavorables et d'incompatibilité.

- Relations d'exclusion : les plans contiennent des actions qui nécessitent l'accès à

une même ressource consommable.

- Relations défavorables : les plans contiennent des actions qui nécessitent l'accès à une même ressource non-consommable.
- Relations d'incompatibilité : le conflit ne réside pas dans l'utilisation des ressources mais plutôt dans le fait que deux actions exigent l'existence d'états exclusifs.

Ces deux approches de classification sont étudiées selon un point de vue de collectivité. Quelque soit cette classification, les agents ont besoin des moyens de communication pour interagir entre eux ou avec leur environnement. Dans ce qui suit, nous décrivons les techniques de communication utilisées lors des interactions ainsi que les protocoles d'interaction.

2.4 Topologie de communication

Fondamentalement, la communication est le transport d'information d'une entité à l'autre [Odell *et al.*, 2002]. Cependant, il y a une différence subtile entre transmission et communication de l'information. La transmission consiste à transférer un message à travers l'environnement sans pour autant qu'il y ait réception de ce message par ses destinataires. Par contre, la communication exige que la transmission d'un message vers un autre agent implique forcément une réaction et un changement d'état de l'agent récepteur. Par exemple, lorsqu'on visualise les spots publicitaires à la télévision, ça implique qu'on a perçu sa transmission. Cette perception peut nous pousser à acheter le produit ou tout simplement à ne pas réagir. Dans les deux cas, la communication s'est produite parce que l'acte de la sensation et de décision s'est réalisé et par conséquent un changement d'état du récepteur s'est produit (achat du produit ou pas d'achat).

La communication, dans les systèmes multiagents, est une forme d'interaction dans laquelle la relation dynamique entre les agents s'exprime par l'intermédiaire de médiateurs, qui une fois interprétés, vont produire des effets sur ces agents [Demazeau, 1995]. Le but de la communication entre agents est d'exploiter les interactions et d'assurer la coordination. Par ce fait, un agent peut persuader d'autres agents d'adopter ses buts et changer leur plan [Inverno and Luck, 2001].
Pour se faire comprendre, les agents utilisent soit un langage commun comme les langages de communication (ACL[1], FIPA-ACL[2]et KQML[3]) ou la théorie des actes du langage. Dans un acte de langage, on fait la distinction entre les aspects locutoires qui

[1]Agent Communication language
[2]Foundation for Intelligent Physical Agents-Agent Communication language
[3]Knowledge Query and Manipulation Language

portent sur les manières de formuler l'énoncé, illocutoires qui décrivent l'effet attendu sur l'auditeur et perlocutifs qui se rapportent aux effets réels des actes locutoires et illocutoires [Koning *et al.*, 1995].

Pour que la communication entre agents puisse avoir lieu, il faut définir une infrastructure. Plusieurs structures de communication existent. On trouve celles basées sur la communication par partage d'information (tableau noir) ou celles basées sur la communication par envoi de messages : point à point, multicast et broadcast; soit de façon synchrone ou asynchrone. Les différents systèmes développés utilisent l'un ou l'autre de ces paradigmes (envoi de message ou partage d'information) et parfois une combinaison des deux. Dans ce qui suit, nous allons décrire les différentes techniques de communication inter-agents permettant ainsi l'élaboration des interactions entre agents.

2.4.1 Communication par envoi de message

Les agents sont en liaison directe et envoient leurs messages directement au destinataire. La seule contrainte est la connaissance de l'agent destinataire. Ce paradigme implique que tous les agents disposent du même protocole qui leur permet d'échanger leurs informations. Ils disposent également d'une base de connaissance locale dans laquelle ils stockent leurs résultats partiels. Chaque agent possède une liste de tous les agents qu'il connaît dans le système et à qu'il peut envoyer des messages : c'est la liste de ses accointances. Il existe plusieurs formes de communication directe que nous allons décrire dans les sections qui suivent.

La communication point à point

La communication la plus basique est l'envoi direct d'un message d'un processus à un autre. Ce type de communication est avantageux dans le sens où les informations confidentielles sont toujours protégées puisque l'agent connaît à l'avance le destinataire du message envoyé. Par contre, cette forme de communication pose un certain problème lorsque le destinataire est non disponible.

La communication par diffusion ou par broadcast

Cette forme de communication est une généralisation de la communication point à point. D'ailleurs, elle est la plus utilisée dans les systèmes à plusieurs agents pour la raison suivante : les systèmes utilisant le mécanisme de diffusion comme moyen de transmission d'information, sont tolérants aux fautes puisqu'il est possible d'y changer les agents

qui ont échoué pour établir la communication, avec d'autres ayant un comportement semblable. Néanmoins, elle souffre de deux inconvénients majeurs. D'une part, la diffusion n'est pas très sécuritaire dans le cas où des agents espions se trouvent dans le système et modifient les messages échangés; d'autre part, la diffusion coûte très cher en termes de ressources réseau (surtout au niveau de l'Internet). Pour pallier à ce dernier problème, plusieurs techniques existent comme le multicast, les filtres, le maintien de liste de diffusion et la durée de vie des messages.

2.4.2 Communication par partage d'informations

Les agents ne sont pas en liaison directe mais communiquent entre eux en partageant leurs informations (solution que chaque agent offre pour la résolution d'un problème : solution partielle) via une structure commune et globale à tous les agents. Cette structure contient toutes les connaissances relatives à la résolution qui évoluent durant le processus d'exécution. Les agents communiquent en lisant ces différentes solutions partielles et en inférant de nouvelles qu'ils placeront dans cette zone commune. Chaque agent a un libre accès à toutes les informations. Ce principe de communication est désigné dans la litérature par le «*modèle de tableau noir*» ou «*blackboard*».
Un des avantages de cette forme de communication est que les agents sont anonymes et il n'est pas nécessaire de connaître quel agent a émis telle solution partielle dans le tableau noir. Il est donc facile d'ajouter ou de retirer des agents au système, pour cela, il suffit de les connecter ou les déconnecter au blackboard. Cependant, les agents ont libre accès aux données ce qui les amènent à traiter une grosse quantité d'informations inutiles d'où la nécessité de structurer les données partagées.

2.5 Protocoles d'interaction

Les conversations inter-agents sont considérées comme le moyen le plus commun par lequel les agents autonomes – *par autonome, nous voulons dire que l'agent est capable d'agir sans l'intervention d'un tiers et contrôle ses propres actions ainsi que son état interne* – coopèrent, se coordonnent et négocient [Reed *et al.*, 2002]. Ils sont souvent structurés selon des schémas typiques appelés protocoles d'interaction. Ces protocoles ne sont pas au même niveau conceptuel que les protocoles de communication. Tandis que les protocoles de communication se focalisent sur la transmission et le routage des messages, les protocoles d'interaction s'intéressent juste à l'échange des connaissances, à la description de la réaction des agents lors de la réception des messages et à la description des enchaînements conversationnels lors des communications entre les agents [Mazouzi, 2001].

Nous présentons dans les sections suivantes, un bref survol des protocoles d'interaction les plus utilisés dans les systèmes multiagents.

2.5.1 Protocoles de coordination

La coordination est le processus qui permet à un agent de raisonner sur ses actions locales et celles des autres afin d'aboutir à des comportements cohérents et satisfaire le but global du système [Jennings, 1996]. Un des objectifs de la coordination est qu'elle permet aux agents d'éviter le conflit d'accès aux ressources, les attentes infinies et l'interblocage. Les raisons principales qui amènent à implanter un protocole de coordination se concentrent sur les faits qu'il y a soit une dépendance entre les actions des agents, soit que les systèmes à élaborer ont des contraintes globales, ou tout simplement les agents n'ont pas assez de compétences, de ressources ou d'informations afin de résoudre seuls le problème tout entier [Jennings, 1996].

Le protocole contract-Net est l'exemple le plus connu des protocoles de coordination. Il a été développé par SMITH et DAVIS dans le but de définir un processus pour l'allocation des tâches [Davis and Smith, 1983]. Dans ce genre de protocole, les agents peuvent dynamiquement prendre deux rôles : contractant et manager, et chacun des agents est capable de communiquer avec ses pairs par l'envoi et la réception de messages. Ce protocole comporte trois étapes principales (Voir FIG.2.1) :

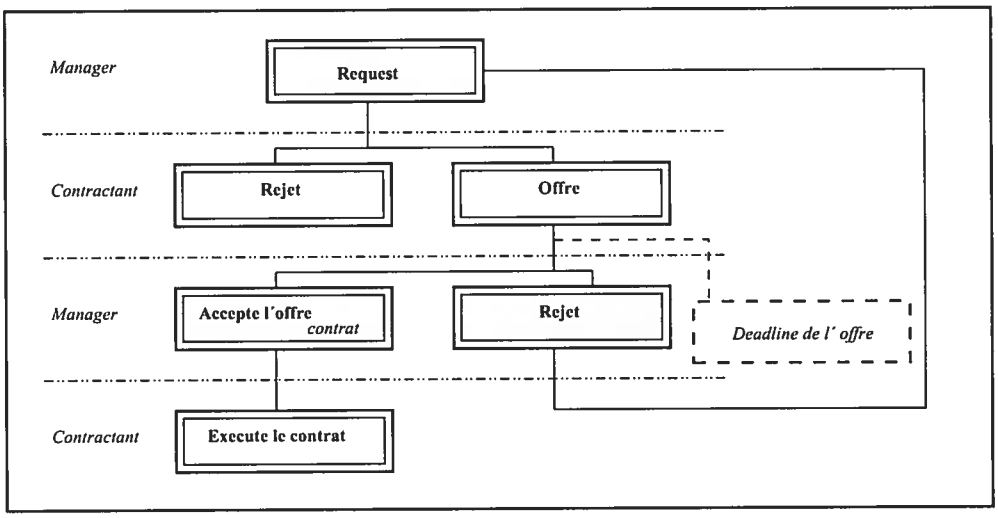


FIG. 2.1: Le protocole de contract-Net

- L'appel d'offre : un agent annonce à ses pairs la tâche qu'il veut exécuter. Durant

cette période, l'agent est considéré comme manager et les autres comme contractants. Si le manager ne connaît pas les capacités des agents pour l'exécution de cette tâche alors il doit faire un broadcast à tous les agents existants. Si par contre, le manager connaît les agents potentiels qui peuvent répondre à sa requête alors il peut limiter sa liste d'envoi.

- La soumission de proposition : ces contractants élaborent leurs offres et les proposent à l'agent manager.
- L'attribution du contrat : l'agent manager peut recevoir plusieurs offres. Il est alors amené à choisir la meilleure offre, sur la base de critères prédéfinis, et envoie le contrat au contractant choisi. À partir de ce moment, l'agent sélectionné doit s'engager envers le manager pour l'exécution du contrat et les autres agents n'interviennent plus jusqu'à la prochaine interaction, c'est-à-dire le prochain appel d'offres. Après l'exécution de la tâche, le contractant doit envoyer un rapport au manager.

2.5.2 Protocoles de coopération

La coopération est une forme de coordination entre des agents non antagonistes. Selon [Doran *et al.*, 1997], la coopération se produit quand les actions de chaque agent satisfaisont l'une ou les deux conditions suivantes :

- Les agents ont un but commun qui peut être implicite et qu'aucun d'eux ne peut réaliser seul. De plus, les actions de ces agents visent à réaliser ce but.
- Les agents exécutent les actions qui permettent d'aboutir non seulement à leurs propres buts, mais également aux buts des autres agents.

Dans cette définition, nous avons constaté que Doran s'est concentré uniquement sur les actions des agents et leurs buts, indépendamment de la façon dont ils surgissent. Le but du protocole de coopération est d'améliorer l'exécution collective des agents. Pour aboutir à une bonne coopération, chaque agent doit maintenir un modèle des autres agents, et planifier les futures interactions. SCHMIDT a classifié la coopération sous trois formes différentes [Schmidt, 1990] :

- La coopération confrontative : le résultat est obtenu par fusion. Chaque tâche est exécutée par plusieurs agents de spécialités différentes oeuvrant de manière concurrente sur le même ensemble de données.
- La coopération augmentative : le résultat est obtenu sous la forme d'un ensemble de solutions locales puisqu'une tâche est répartie sur une collection d'agents

similaires, oeuvrant de manière concurrente sur des sous-ensembles disjoints de données,

- La coopération intégrative : le résultat d'une tâche est obtenu au terme de son exécution puisqu'une tâche peut être décomposée en sous-tâches qui seront ensuite résolues par des agents de spécialités différentes et oeuvrant de manière coordonnée.

2.5.3 Protocoles de négociation

À l'inverse de la coopération, la négociation correspond à la coordination en univers compétitif ou entre agents concurrentiels. Selon [Jennings *et al.*, 1998], la négociation est un processus par lequel une décision de résolution des problèmes de coordination et de conflit est prise entre au moins deux agents. Les caractéristiques principales du protocole de négociation se manifestent par la présence d'une certaine forme de conflit qui doit être résolue d'une façon décentralisée, par les agents intéressés dans des conditions incomplètes. Le mécanisme d'une phase de négociation débute par une communication des positions des agents, qui pourraient causer un conflit entre ces derniers. Ensuite, l'établissement d'une entente entre les agents doit se faire par le biais de concessions. Un agent fait une concession quand il ne tient pas compte des contraintes les moins importantes et il y aura un accord entre les agents dès que toutes les contraintes sont satisfaites ; nous parlerons alors de négociation par compromis.

Le protocole de négociation essaiera d'aider les agents à atteindre leurs propres buts lorsqu'il y a une divergence ou tout simplement une différences entre ces buts. Pour arriver à assurer toutes ces fonctionnalités, un mécanisme de négociation doit être efficace, fiable – *pas de perte de messages afin d'aboutir a un accord*– et stable –*c'est-à-dire que l'agent ne doit pas changer fréquemment sa stratégie de résolution de problème*–.

2.5.4 Discussion

La notion d'interaction prend une place prépondérante dans l'aboutissement de la coopération, la collectivité et l'intelligence des SMAs, puisqu'elle englobe plusieurs actions de communication. Afin de standardiser ces interactions, des protocoles ont été définis (contract-net, FIPA-contract-net...). Malgré que ces protocoles limitent l'autonomie des agents parce qu'ils leur imposent des modèles de conversation, ils constituent un facteur essentiel dans l'amélioration de l'aspect conversationnel des agents.

2.6 Vérification des systèmes multiagents

La vérification est un processus important du cycle de développement de logiciels. Elle est une étape critique pour garantir la robustesse, la fiabilité, la sécurité et la qualité des systèmes. Les SMAs doivent disposer de méthodes et de techniques de vérification pour leur implication dans le secteur industriel. Néanmoins, les techniques classiques de vérification ne sont pas tout a fait adaptées à ces systèmes, puisqu'elles ne tiennent pas compte de la nature et la complexité des SMAs. En effet, pour avoir une bonne méthode de vérification des SMAs, cette dernière doit tenir compte des aspects tels que le non déterminisme, la concurrence et la répartition puisqu'une exécution multiagent est en principe non déterministe, concurrente et répartie.

Cette vérification a pour but d'expliquer les incohérences, de détecter et corriger, dans la mesure du possible, les erreurs protocolaires ou applicatives, qu'elles soient quantitatives ou qualitatives, et d'analyser le comportement global des SMAs. Ce processus de vérification, qu'il soit post-mortem ou on-line, nécessite un mécanisme d'observation permettant d'observer le comportement des agents afin de les analyser. Cependant, l'observation des SMAs est confrontée au problème de l'émergence. En effet, certaines propriétés du système apparaissent au niveau global (on parle de propriétés émergentes) et donc ne sont pas visibles au niveau des interactions. Pour résoudre ce problème, la communauté de recherche a eu recours à la simulation afin d'observer la dynamique du comportement.
L'observation du comportement d'un agent peut être abordée selon deux points de vue:

- Interne à l'agent. Ce qui revient à observer son mécanisme de raisonnement ainsi que ses capacités à prendre des décisions et à percevoir son environnement.
- Externe à l'agent. Ce qui revient à observer les actions prises par l'agent, ses activités et les communications établies entre eux.

Nous nous sommes basés sur l'étude du comportement externe des agents à travers les protocoles d'interaction dans le but d'analyser et vérifier les SMAs. Les deux raisons qui nous ont poussés à faire ce choix sont : la première est que le comportement externe des agents est fortement lié à leur comportement interne puisque tout comportement externe est la conséquence immédiate du raisonnemnet de l'agent ; la deuxième raison est que l'accès au comportement interne n'est pas toujours possible et que nous avons plus facilement accès aux informations relatives aux communications inter-agents.

2.7 Conclusion

Dans ce chapitre, nous avons survolé les concepts de base des systèmes multiagents et des interactions. Bien que les protocoles d'interaction permettent de contrôler les différentes formes existantes d'interaction entre les agents, il n'existe pas encore, à l'heure actuelle, des outils pour aider, simultanément, à la conception, à la vérification et à la validation de ces protocoles.
La conversation et l'échange d'informations entre les agents se fait grâce à la communication, c'est pour cette raison qu'elle constitue un concept fondamental du paradigme multiagents et le coeur de l'interaction. L'observation et la vérification du comportement des agents sont axées essentiellement sur les interactions. En effet, la notion d'interaction, offre un cadre de développement et d'analyse très intéressant vu qu'elle crée des possibilités d'échanges et d'influences entre les agents et qu'elle reflète à la fois le comportement individuel de l'agent et celui global du SMA .

Dans le chapitre suivant, nous nous intéresserons aux modèles et aux langages de spécification pour la modélisation des protocoles d'interaction et la formalisation des propriétés recherchées.

Chapitre 3

Spécifier pour vérifier

Plusieurs méthodes de vérifications formelles existent. Ces méthodes supposent qu'il doit être possible de spécifier complètement et sans aucune ambiguïté le comportement d'un système, que ce soit le comportement interne (mécanisme de raisonnement) ou le comportement externe (protocole d'interaction, coopération). Pour ces raisons, la spécification et la modélisation des comportements des SMAs par des approches formelles sont devenues un domaine de recherche actif. Cette spécification formelle permettra de modéliser les systèmes sans aucune ambiguïté et par conséquent de prouver certaines propriétés de ces derniers.

Ce chapitre traitera des différents formalismes de spécification des systèmes et des propriétés souhaitées dans ces systèmes.

3.1 Introduction

Les tentatives de spécification formelle des SMAs sont nombreuses, plus particulièrement pour les systèmes composés par des agents cognitifs qui suivent le schéma BDI (Belief Desire Intention) [Inverno *et al.*, 1997]. Ces méthodes de spécification doivent permettre la spécification des aspects dynamiques, de la concurrence et du non-déterminisme afin de garantir une parfaite modélisation des SMAs. Cependant, les méthodes de spécification des systèmes classiques se focalisent sur deux axes, l'axe fonctionnel (fonctions) et l'axe informationnel (données). D'où la nécessité d'avoir de nouvelles méthodes ou formalismes.
Les formalismes les plus souvent utilisés pour représenter l'aspect dynamique des SMAs sont les logiques temporelles, les langages formels et semi-formels comme UML (Unified Modeling Language), AUML (Agent Unified Modeling Language) ainsi que les réseaux

de pétri colorés.

Dans ce que suit, nous nous intéresserons aux modèles de spécification formelle permettant de couvrir ces trois axes (fonctionnel, informationnel et dynamique) et aux langages formels de description.

3.2 Modèles formels des SMAs

Les langages et les approches développées pour la formalisation et la description des modèles ont été à l'origine conçus pour les systèmes de télécommunication. Ces approches ont été adaptées et appliquées à l'étude des SMAs. Actuellement, il existe beaucoup de travaux qui ont proposé des modèles pour les SMAs. Ces modèles se composent de deux grandes familles, ceux qui reflètent le comportement interne des agents [Gasser *et al.*, 1987], [Martial, 1992] et ceux qui reflètent leur comportement externe [Mazouzi, 2001].

Le comportement interne décrit les capacités qu'un agent a pour raisonner, décider, percevoir son environnement et aboutir à ses objectifs. La modélisation d'un tel comportement revient alors à une représentation explicite des différents états dont lequel le système a évolué. Par contre, le comportement externe est décrit comme une séquence d'interactions entre les agents et entre le système et son environnement. Il permet de traduire aussi l'état interne des agents en offrant uniquement une vue partielle de son état. Ainsi, en modélisant le comportement externe des agents à l'aide des interactions, nous pourrons alors vérifier une panoplie plus large de propriétés par le fait que le comportement externe résulte du comportement interne des agents. De plus, ces propriétés peuvent être exprimées plus facilement par rapport à celles recherchées au niveau du comportement interne des agents. Les modèles qui décrivent le comportement des agents sont basés sur la description des états dans lesquels ils évoluent ou sur l'occurrence des événements inhérents aux messages échangés par les agents, ou une fusion des deux (modèles de transitions étiquetées).

3.2.1 Modèle d'événements

Un modèle d'événements permet de définir les types, les sources et les destinations des événements ainsi que les opérations qui leur sont appliquées. Avec ce genre de modèle, le comportement d'un agent est décrit par un ensemble de séquences d'événements simples et/ou complexes. Les événements simples sont les événements locaux et les complexes sont ceux qui font interagir plusieurs agents, comme les interactions. Ce modèle est

fondé sur deux principes simples :

- Le premier principe stipule qu'un événement est un changement d'état du système qui entraîne l'émission d'un message ou une action locale.
- Le second principe est le principe de réaction. L'émission d'un événement entraîne dans l'ensemble du système l'apparition et l'exécution d'une réaction.

Le modèle d'événement est ainsi basé sur trois types d'événements : les événements émis, les événements reçus et les événements locaux.

3.2.2 Les machines à états

Avec ce type de modélisation, le système sera représenté par son état et un ensemble d'opérations qui modifient et interviennent sur cet état. Ce dernier décrit un moment unique pendant la vie d'un agent. En général, un état est décrit par les tâches et les actions qu'un agent a pris à un instant t pour résoudre un problème en coopérant ou en négociant avec d'autres agents. Ceci dit, l'état d'un SMA est donné par l'ensemble des états de ses agents. Le modèle d'état peut être considéré comme le modèle le plus adéquat pour la modélisation du comportement interne des agents.

3.2.3 Les systèmes de transitions étiquetées

Les modèles de transitions étiquetées offrent une description comportementale du système. En effet, ils permettent de modéliser l'activité collaborative d'un système par ses événements et ses états. Notons comme exemple de modèle de transitions étiquetées, les automates à états finis ainsi que les réseaux de Pétri. La majorité des travaux de modélisation des SMAs est axée sur l'utilisation de ces deux types de modèles.

Dans le cadre de cette recherche, nous considérons uniquement les comportements externes des agents pour les raisons expliquées plus haut. Dans ce qui suit, nous présentons différents types de logiques qui sont utilisées dans le but de décrire les propriétés recherchées dans le modèle du système.

3.3 La logique comme formalisme de modélisation

Une des caractéristiques de la logique est qu'elle permet d'exprimer des propriétés portant sur les exécutions d'un système, c'est-à-dire les propriétés d'un état donné d'un

système. Cependant, il existe des logiques qui ont été définies pour spécifier non pas les propriétés recherchées dans un système, mais plutôt le système lui même. Dans cette section, nous allons détailler quelques unes de ces logiques.

3.3.1 La logique de Hoare

La logique de Hoare [Hoare, 1969] est utilisée pour annoter un programme afin de le vérifier. Son principe de base est d'enrichir le code source d'un programme, avec des assertions logiques afin de spécifier le comportement de ce programme. Ces assertions sont placées au début et à la fin du programme et sont appelées respectivement pré-condition et post-condition. Les pré-conditions sont vérifiées en observant l'état du programme avant l'exécution de celui-ci et les post-conditions sont vérifiées en observant l'état de la mémoire après l'exécution du programme. Nous pouvons dire que cette logique considère le programme comme une relation «état-initial, état-final», puisqu'elle exprime toujours les propriétés d'un état final, à partir des propriétés de l'état initial [Hoare, 1969]. Le triplet de $\{\phi\}P\{\psi\}$ exprime que dans l'état final, le système P vérifie la propriété exprimée par ψ si dans l'état initial, le système P vérifiait la propriété exprimée par ϕ. ψ et ϕ sont deux formules représentant les propriétés recherchées.

Cependant, la vérification des SMAs nécessite une prise en compte du comportement observable d'entités pour lesquels nous n'avons pas toujours accès au code source. De plus la majorité de ces systèmes tournent continuellement et donc n'ont pas d'états finaux. Ces deux inconvénients font que la logique de HOARE n'est pas adaptée pour modéliser les SMAs, donc il est nécessaire de disposer de formalisme capable d'énumérer tous les états du système d'où la logique temporelle.

3.3.2 La logique temporelle

La logique temporelle permet d'exprimer l'évolution de l'état d'un système et de voir la validité des formules qui évoluent dans le temps. Elle sert à énoncer formellement des propriétés portant sur les exécutions d'un système – ces propriétés dictent le comportement dynamique d'un système. Un exemple de ces propriétés est : *Dans un système de contrôle de chauffage, toute demande de modification de température doit finir par être satisfaite*–. Pour ce faire, elle dispose d'opérateurs pour exprimer le passé, le présent et le futur. Elle est bien adaptée pour exprimer des propriétés comme la terminaison, l'absence de blocage, la vivacité, l'équité et la sûreté, indépendamment de l'implantation du système [Jarras, 1995].

Les logiques temporelles se regroupent principalement en deux catégories :

- Les logiques temporelles arborescentes[1] (CTL). Les formules de la logique CTL ont deux composantes. Une composante temporelle qui permet de modéliser le temps comme «*E: éventuellement*», «*G: toujours*». Tandis que l'autre composante permet de spécifier la couverture de l'arbre d'exécution par des «*quantificateurs de chemin*» comme «*A: pour tous les chemins*», «*F: pour certains des chemins*». Les formules de CTL s'appliquent aux arbres qui correspond à toutes les évolutions possibles du système à un moment donné (Voir FIG. 3.1).
- Les logiques temporelles linéaires[2] (LTL). Cette logique s'intéresse à une exécution du système à la fois et ses formules représentent l'une des exécutions du système.

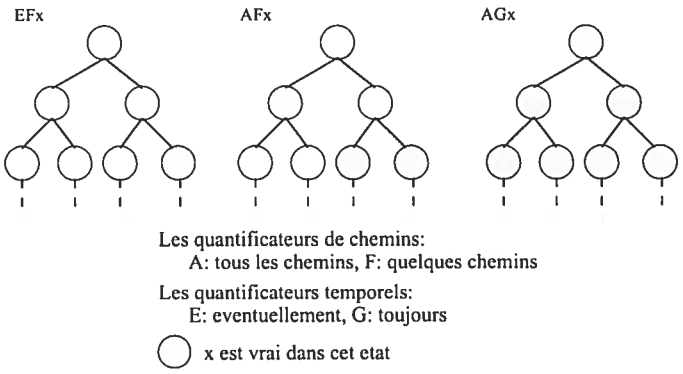


FIG. 3.1: Les quantificateurs de CTL

Ces deux types de logiques sont les plus utilisées dans les outils de model-checking (Voir §4.3). Le choix de l'une ou l'autre dépend de plusieurs facteurs. Si nous voulons faire une vérification exhaustive d'un système, alors une spécification en CTL est plus adaptée que celle en LTL. Si par contre, nous ne visons pas l'exhaustivité et nous voulons faire une vérification à la volée (on-line) alors LTL est considérée comme un choix judicieux [Berard *et al.*, 2001]. La logique temporelle a été utilisée dans l'approche axiomatique pour axiomatiser le comportement, spécifier et caractériser les propriétés comportementales à l'aide de formules en logique temporelle (Voir §4.2).

3.4 Langages formels de spécification et de description

Nous présentons quelques langages de description permettant la modélisation des protocoles d'interactions des SMAs et des propriétés recherchées dans ces systèmes.

[1] Computational Tree Logic
[2] Linear Temporal logic

3.4.1 Langage Z comme langage de modélisation du SMA

Le langage Z est un langage de spécification formelle des systèmes, normalisé au standard ISO[3] basé sur les machines à états abstraits. Il trouve son fondement dans la théorie des ensembles et il est enrichi par des types et des notations, appelées schémas, pour structurer les spécifications. Chaque schéma est composé d'une partie signature qui définit ses variables et d'une partie prédicat définissant la relation entre les variables et les contraintes appliquées à ses variables au moyen d'assertions logiques.

Il est à noter que le langage Z n'est pas très adapté pour la modélisation et la vérification des protocoles d'interaction puisqu'il n'offre pas un modèle de communication et de synchronisation de processus. Malgré cette contrainte, l'article
[Inverno *et al.*, 1997] vient pour décrire l'utilisation du langage Z pour la spécification formelle des SMAs.

3.4.2 Langage SDL comme langage de modélisation du SMA

SDL (Specification Description language) est un langage de spécification[4] et de description[5] qui a été défini en 1972 par le CCITT[6] nommé présentement ITU[7]. Il est basé sur les modèles de transitions étiquetées, spécifiquement les machines à états finis. Il fait partie de la famille des techniques de description formelle[8], ainsi que les langages LOTOS[9] et ESTELLE[10]. SDL permet de spécifier et de décrire formellement et sans ambiguïté le comportement des systèmes englobant des activités parallèles et communicantes. Ce langage couvre aussi bien les procédures de contrôle que les applications temps réel. La force et la popularité de SDL sont dues à sa puissance quant à ses capacités d'expression, aux représentations graphique (SDL/GR) et textuelle (SDL/PR) équivalentes et surtout au fait qu'il existe des outils, comme ObjectGEODE et SDT de Telelogic, qui le supportent. En outre, l'utilisation du langage SDL nécessite l'utilisation de MSC (Message Sequence Charts) qui représente des séquences de messages entre les modèles décrits en SDL. La spécification de SDL contient les composantes suivantes [Hallal and Petrenko, 2001] :

Structure : la structure d'un système décrit en SDL est fondée sur la décomposition hiérarchique. Cette structure est composée de quatre hiérarchies : le système, le bloc,

[3]International Standard Organization
[4]exprime ce que le système doit faire
[5]exprimant comment le système le fait réellement
[6]Comité Consultatif International Téléphonique et Télégraphique
[7]International Telecommunication Union
[8]FDT, Formal Description Technique
[9]Language of Temporal Ordering Specification
[10]Extended State Transition Language

le processus et la procédure. L'entité la plus haute de la hiérarchie est système. Une instance du système contient un ensemble de blocs qui peuvent se décomposer en plusieurs processus. Chaque processus est représenté par une machine à état fini étendue. Ces états peuvent être simple ou complexe et donc peuvent contenir des sous états qui sont définis par les procédures.

Comportement : le comportement du système est décrit en termes de processus coopérants et communicants par échange de signaux. Le processus est représenté sous forme d'un automate à états finis communicant avec les autres de manière asynchrone (Voir FIG.3.2). Chaque processus possède une file d'attente de type First In First Out (FIFO) de taille infinie, dans laquelle les signaux sont stockés à leurs arrivées.

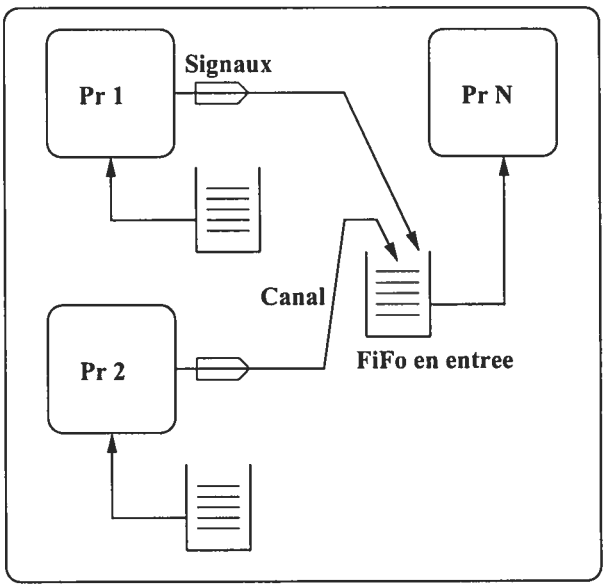


FIG. 3.2: Comportement d'un système SDL

Communication : la communication est asynchrone – *les messages sont imédiatement émis vers la file d'attente du récepteur* –. L'acheminement des informations se fait par l'utilisation des signaux et des canaux. Les signaux s'intercalent entre les processus ou entre les différents processus et l'environnement système du modèle. Par contre, les canaux s'intercalent entre les blocs ou entre les différents blocs et l'environnement système du modèle.

Données : la description des données est fondée sur les types abstraits algébriques, qui sont prédéfinis : BOOLEAN, CHARSTRING, INTEGER, TIME... ou sur des définitions de tableaux et de structures.

3.4.3 Langage GOAL comme langage de formalisation des propriétés

Comme nous avons mentionné dans la section 3.3.2, les propriétés sont formalisées en utilisant soit la logique CTL ou LTL. Cependant, ces logiques ne sont pas faciles à utiliser et nécessite une expertise pour les maîtriser. Ceci dit, il existe des langages d'un niveau d'abstraction plus élevé permettant la description des propriétés écrites en CTL ou LTL. Parmi ces langages, notons GOAL (Geode Observation Automata Language) qui est un langage propriétaire permettant la spécification des patterns ou des propriétés à vérifier dans un modèle SDL. Il spécifie les propriétés en processus SDL avec quelques différences syntaxiques. Ces processus s'appellent les *observers* et ils sont habituellement décrits en termes d'entités (objets, signaux) du système examiné. En outre, il permet la déclaration de deux types d'états : états de succès et états d'erreur. L'état de succès (état d'erreur) indique que le système respecte (viole) la propriété exprimée par un *observer*. Cependant, cette convention est informelle et l'utilisateur peut la définir à sa guise. L'annexe A offre plus de détails sur la structure du langage GOAL.

3.5 Choix et discussion

Les différents formalismes, cités auparavant, ont un grand pouvoir d'expression puisqu'ils offrent la possibilité de vérifier, de façon formelle, des propriétés des SMAs. En effet, ces formalismes ont été utilisés pour modéliser les SMAs ou les propriétés dynamiques et comportementales des systèmes à étudier. Avec cette diversité de langages formels, le choix d'un formalisme adéquat est fondamental. Dans notre travail, nous avons choisi SDL pour décrire les protocoles d'interaction des SMAs. Ceci nous a permis de vérifier et par conséquent d'analyser le comportement du SMA. Ce choix a été motivé par le fait que toute modélisation d'un SMA doit supporter les caractéristiques suivantes :

- Le non-déterminisme : il est engendré par le fait que les agents sont des entités autonomes et que deux exécutions peuvent donner deux comportements différents du SMA.
- Le parallélisme : En effet, deux agents peuvent être concurrents et donc exécuter la même tâche ou deux tâches différentes en même temps.
- La communication : c'est un moyen essentiel pour assurer l'interaction entre les agents ou entre les agents et leur environnement.
- La synchronisation : la synchronisation des actions entre les agents est un mécanisme indispensable pour assurer une bonne coopération entre les agents.

De plus, tout formalisme de spécification d'un SMA doit être suffisamment expressif pour modéliser non seulement le comportement d'un seul agent mais également la collaboration entre plusieurs agents y compris l'influence de l'environnement sur ces derniers.

3.6 Conclusion

Dans ce chapitre, nous avons abordé quelques unes des différentes méthodes formelles de modélisation des systèmes et de spécification des propriétés comportementales. Des modèles et des langages divers sont proposés pour fournir une modélisation aussi fidèle que possible des comportements des systèmes. Ces modèles seront par la suite utilisés pour la vérification formelle. Le chapitre suivant présente une étude sur les techniques de vérification ainsi qu'une synthèse de ces derniers. Nous choisissons la méthode la plus adéquate pour vérifier les propriétés recherchées.

Chapitre 4

Vérification formelle des SMAs

La nécessité de vérifier les systèmes, plus spécifiquement les logiciels industriels, ne cesse de s'accroître. Cette tâche est de plus en plus difficile à cause de la taille, de la complexité, du caractère distribué et des contraintes temporelles que l'application doit satisfaire. Les SMAs sont des systèmes distribués. Cette distribution est soit physique – *chaque agent peut être sur un site différent* – soit fonctionnelle – *chaque agent a une tâche (une partie du problème) qu'il se charge de résoudre*–, ou une combinaison des deux. De plus, ce sont en principe des systèmes concurrents et non déterministes. Ces aspects rendent la vérification des SMAs plus difficile.

Vérifier un SMA revient à répondre à la question suivante : «une exécution multi-agents possède-t-elle des propriétés?». La réponse à cette question n'est pas évidente. Dans ce chapitre, nous allons présenter un panorama des approches de vérification permettant de détecter les erreurs potentielles. Par la suite, nous montrerons comment elles ont fait la preuve de leur utilité au niveau des SMAs. Ensuite, nous présenterons une synthèse des travaux effectués dans ce domaine et nous décrivons brièvement notre approche de vérification qui sera détaillée dans le chapitre suivant.

4.1 Les approches de vérification

Dès qu'on développe un système, on est amené à s'assurer de sa conformité avec sa spécification originale, établie au début du processus de développement. Cette activité est connue sous le nom de vérification et il est particulièrement important lorsque on introduit des modifications ad-hoc au cur du processus de développement. Par exemple, les améliorations manuelles faites sans aucune preuve formelle peuvent créer une fausse transformation de la spécification à l'implantation. Il existe plusieurs tech-

niques statiques et dynamiques qui peuvent être utilisées pour la vérification et l'analyse des systèmes [Sommerville, 1996]. Les techniques statiques se basent sur l'analyse et la vérification, entre autres, des diagrammes de conception, du code source de programme. Ces techniques peuvent être appliquées à toutes les étapes du processus. Par contre, les techniques dynamiques ne peuvent être utilisées que si on a un prototype ou un programme exécutable (Voir FIG. 4.1).

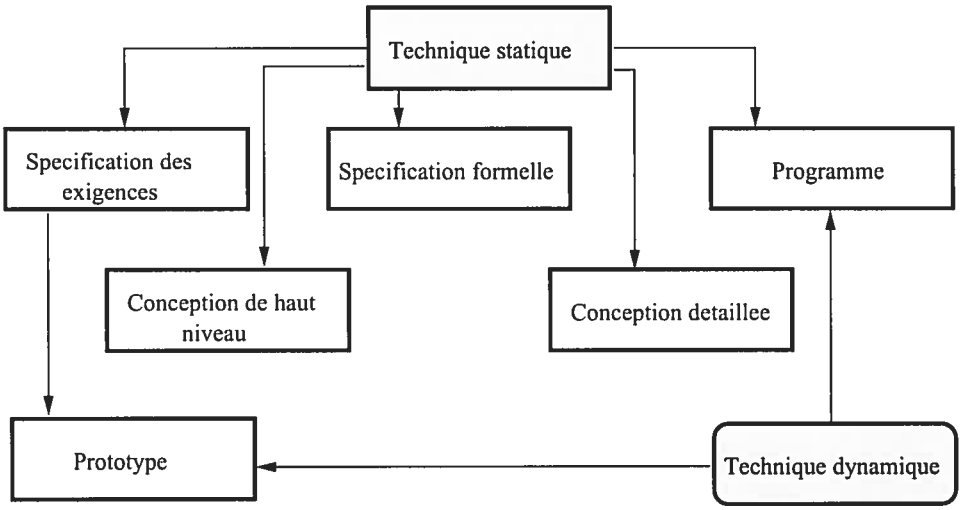


FIG. 4.1: Implication des techniques statique et dynamique dans le cycle de développement d'un logiciel

Les techniques statiques incluent les inspections du programme, l'analyse et la vérification formelle. [Wooldridge and Ciancarini, 2000] et [Katoen, 2002] ont divisé la vérification basée sur les notions mathématiques en deux larges classes : les approches axiomatiques et les approches sémantiques.

Les approches axiomatiques

Les approches axiomatiques ou ce qu'on appelle aussi vérification déductive ou par preuve sont basées sur le concept d'axiomatisation du code source du système. Cette approche permet, à partir d'un système et d'une propriété exprimés dans un même langage de spécification, de prouver que la propriété est vérifiée ou non par le système. Ceci est réalisé en utilisant des règles de déduction, comme on pourrait le faire pour démontrer un théorème de mathématiques. La vérification est considérée comme un théorème mathématique qui a la forme suivante : *spécification du système⇒propriété désirée.* Il existe beaucoup d'outils de vérification comme les démonstrateurs de théorèmes et les vérificateurs de preuves [Katoen, 2002]. Néanmoins, l'utilisation de ces derniers

nécessite que la description des systèmes soit présentée sous une forme de théorie mathématique ou une forme transformable. De plus, l'utilisateur doit donner des indications à l'outil (c'est-à-dire les points d'entrée).

Les approches sémantiques

Parmi ces approches, on trouve les techniques d'analyse de graphe, les exécutions symboliques et les techniques basées sur les modèles[1]. Les techniques basées sur les modèles sont fondées sur une représentation plus ou moins abstraite de l'ensemble des comportements possibles du système. Il s'avère que la modélisation précise des systèmes est une tâche critique et décisive pour la pertinence des résultats obtenus. En outre, elle mène souvent à la découverte d'imperfections, d'ambiguïtés et de contradiction dans les spécifications informelles du système. Ces modèles sont accompagnés par des algorithmes qui explorent systématiquement tous les états du modèle. Ceci fournit la base pour une gamme entière des techniques de vérification s'étendant d'une exploration exhaustive (le model-checking) à une expérimentation avec un ensemble restreint de scénario dans le modèle (la simulation), ou dans le système lui même (le test). Le model-checking est une procédure automatique qui permet de vérifier que le modèle d'un système vérifie une spécification. Cette procédure de vérification ne se fait pas «par déductions» comme dans le cadre de la vérification par preuve mais grâce à des algorithmes tirant profit des modèles utilisés pour le système et pour la propriété.

Dans les sections qui suivent, nous allons expliquer ces différentes approches de vérification et montrer la faisabilité de ces approches pour les SMAs.

4.2 Approches axiomatiques

L'approche axiomatique de vérification des programmes est considérée comme la première génération de méthodes de vérification. Un des fondateurs de cette approche est Hoare [Hoare, 1969]. Elle est basée sur une analyse du code source du système, ce qui implique qu'on doit disposer du programme. Elle s'appuie sur une traduction logique du système et des propriétés à vérifier. La vérification dans ce cadre s'exprime comme une recherche de preuve en logique. Elle fait appel à la logique temporelle pour caractériser les propriétés de comportement. Les différentes étapes de cette vérification sont les suivantes:

1. Les points de contrôle du programme doivent être assortis d'assertions logiques

[1] Model-based techniques

sur les variables du programme et sur leurs relations. Avoir une notation explicite pour la valeur de la variable x au point de contrôle P_i aide à mettre au point une spécification lisible.

2. Les assertions $A_1, A_2 \ldots, A_n$ sont associées aux points $P_1, P_2 \ldots, P_n$ d'une partie du code que l'on notera Q. A_1 (pré-condition) doit être une assertion des entrées de Q et A_n (post-condition) doit être une assertion des sorties de Q.
3. Pour vérifier l'exactitude du code entre les points P_i et P_{i+1}, le vérificateur doit s'assurer que les instructions qui séparent ces points impliquent une transformation de l'assertion A_i vers A_{i+1}.
4. Si on est sûr que A_1 mène toujours à A_2, A_2 à A_3 et ainsi de suite, jusqu'à ce que toutes les instructions seront parcourues, alors on peut conclure que l'assertion A_1 mène à A_n. Cette règle de déduction se note :

$$\frac{\{A_1\}P_1\{A_2\}\ldots\{A_i\}P_i\{A_{i+1}\}\ldots\{A_{n-1}\}P_{n-1}\{A_n\}}{\{A_1\}P_1;\ldots P_i;\ldots P_{n-1}\{A_n\}} \tag{4.1}$$

La partie du code Q est cependant partiellement correct. Pour montrer que le programme est totalement correct, il faut qu'il soit déterministe.

Peu de travaux de vérification des SMAs, utilisant l'approche axiomatique, ont été effectués au sein de la communauté des systèmes orientés agents. L'un de ces travaux est la thèse de [Wooldridge, 1992], il a proposé une approche axiomatique pour la vérification des SMAs, l'idée était d'utiliser une logique temporelle qui décrit les croyances[2] (TBL) pour axiomatiser les propriétés de deux langages de programmation multiagents. Avec une telle axiomatisation, une théorie de programmes représentant les propriétés du système a été systématiquement dérivée.

La TBL est une logique temporelle linéaire enrichie par un ensemble de liaisons modales unaires de croyances, avec une sémantique donnée en termes de modèles de croyance. Elle a été utilisée pour deux raisons. D'abord, une composante temporelle a été requise pour capturer le comportement continu du SMA. Par la suite, une composante de croyance a été utilisée parce que les agents à vérifier sont des systèmes symboliques, c'est-à-dire que chaque agent inclut une représentation de son environnement et un comportement désiré. La composante de croyance permet de capturer les représentations symboliques de chaque agent.

Les deux langages de programmation multiagents qui étaient axiomatisés dans la TBL étaient AGENT0 de Shoham [Shoham, 1994] et le Concurrent METATEM de Fisher [Fisher, 1995]. La méthodologie utilisée est la suivante :

[2] À défaut de trouver une traduction exacte de Temporal Belief Logic, nous utilisons, tout au long du rapport, le mot en anglais

1. Un modèle abstrait est développé pour les agents. Ce modèle capture le fait que les agents sont des systèmes de raisonnement symbolique, capable de communiquer. Il donne une vue sur la façon dont les agents pourraient changer d'état et réagir.
2. La trace d'exécution d'un tel système est utilisée comme base sémantique pour TBL. Cette logique permet d'exprimer les propriétés des agents modélisées à l'étape 1.
3. TBL est utilisé pour axiomatiser les propriétés du langage de programmation multiagents. Cette axiomatisation a été utilisée pour développer une théorie de programme du SMA.
4. La théorie de preuve de TBL a été utilisée pour vérifier les propriétés du système.

Cette approche repose sur le fait que les agents sont suffisamment simples et que leurs propriétés sont facilement axiomatisées dans la logique. Elle a fonctionné pour AGENT0 et le Concurrent METATEM vu que ces langages ont une sémantique simple, étroitement liée aux systèmes basés sur les règles. Pour des agents plus complexes, une axiomatisation n'est pas aussi évidente, surtout que la prise en compte de la sémantique de l'exécution concurrente des agents n'est pas une tâche facile. Un autre inconvénient de cette approche est le fait que la vérification axiomatique se réduit à un problème de preuve, donc elle est ainsi limitée par les difficultés de cette démarche. Déjà l'élaboration d'une preuve est assez difficile, même dans la logique classique. De plus, la complexité théorique de la preuve pour plusieurs de ces logiques est une tâche assez ardue ; l'addition des liaisons temporelles et modales à une logique rend le problème considérablement plus dur à traiter [Wooldridge, 1992].

4.3 Approches sémantiques

Le MC est une méthode récente qui s'applique à une large classe de systèmes : tous ceux qui peuvent être modélisés par un automate fini ou une variante de cette représentation. C'est un exemple de procédé permettant de vérifier formellement, de manière automatique et exhaustive, que le modèle à états fini d'un système vérifie certaines propriétés, exprimées en général par des formules de logique temporelle [Katoen, 2002].

Du point de vue mathématique, le MC se traduit par : étant donné une spécification φ d'un certain système et un programme π qui implante φ, alors pour déterminer si ou non π implante correctement φ, on procède comme suit (Voir FIG. 4.2) :

- À partir de π, on génère un modèle M_π qui correspond à π, ce modèle décrit comment le système se comporte,

- Puis on détermine si $M_\pi \models \varphi$, si la spécification de la formule φ est valide dans M_π; le programme π satisfait la spécification φ dans le cas où la réponse est positive.

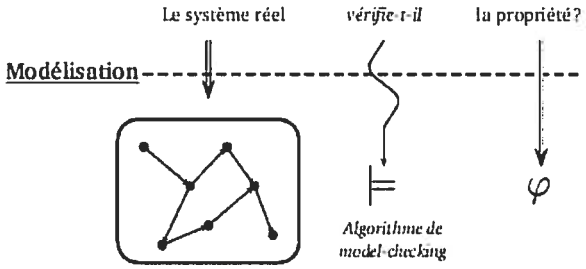


FIG. 4.2: Principe général du model-checking

4.3.1 Processus de vérification par model-checking

La technique de MC s'organise selon trois étapes comme l'illustre la figure 4.3. Ces trois étapes couvrent la modélisation du système, la formalisation des propriétés recherchées et la vérification.

La phase de modélisation du programme ou du système

Cette étape permet de construire une modélisation formelle du système, sous la forme d'un automate ou plus généralement, sous la forme d'un réseau de plusieurs automates synchronisés, reproduisant aussi fidèlement que possible le comportement du système réel. Pour cela, on doit utiliser un langage de spécification (nous avons fait la revue de quelques langages, pour plus de détails se référer au §3.4). Cette phase est très pertinente et souvent cruciale pour la qualité des résultats obtenus lors de la phase de vérification.

La phase de spécification des propriétés

Cette phase permettra d'énoncer formellement les propriétés à vérifier. Son objectif est de définir ce que le système devrait faire et/ou ce qu'il ne devrait pas faire. En général, on utilise un langage de spécification de propriétés comme la logique temporelle ou une variante de cette logique. Cette logique permettra d'exprimer ces propriétés sous forme de formule logique du genre : Il est possible d'avoir la propriété φ dans un état futur ou on aura forcément φ dans un état futur, etc.

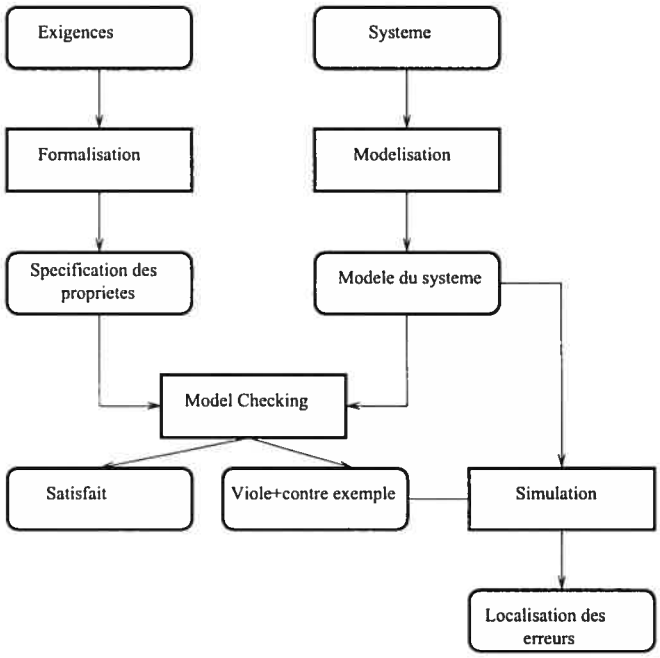


FIG. 4.3: Une vue schématique de l'approche du model-checking

La phase de vérification

Pour vérifier si le modèle du système satisfait un certain nombre de propriétés, il faut disposer d'un algorithme permettant d'analyser le modèle et d'examiner l'existence ou non de ces propriétés. Cet algorithme est exécuté par un *model-checker* [Berard *et al.*, 2001]. Ce dernier explore tous les états possibles du système. S'ils satisfont la propriété alors le *model-checker* donne un diagnostic positif. Dans le cas contraire, si l'état rencontré viole la propriété, alors le *model-checker* fournit un contre-exemple qui indique comment le modèle a pu atteindre cet état indésirable. Ce contre-exemple décrit un chemin d'exécution qui relie l'état initial à l'état qui viole la propriété à vérifier. Ensuite et grâce à un simulateur, l'utilisateur peut rejouer le scénario de violation. De cette façon, il peut obtenir les informations de mise au point[3], et par conséquent adapter le modèle ou reformuler la propriété.

Il existe deux variantes de MC : le model checking symbolique (MCS) et celui temporisé (MCT). Le MCS s'applique à toute méthode de MC qui représente de façon symbolique (non explicite) les états et les transitions d'un automate. Par contre, le

[3]debugging

MCT s'applique aux automates temporisés qui sont des automates étiquetés par des informations quantitatives sur l'écoulement de temps. Ces deux types de MC peuvent s'appliquer aux SMAs et le choix de l'un ou l'autre dépend des caractéristiques du système à vérifier. En effet, si nous devons vérifier des systèmes assez grands alors nous devons choisir le MCS pour contourner le problème de l'explosion combinatoire du nombre d'états. Et si les propriétés recherchées dans le système exigent un ordonnancement temporel tenant compte d'informations quantitatives sur le délai qui sépare les actions, alors nous sommes amenés à modéliser le système par des automates temporisés et donc utiliser le MCT. Nous détaillerons uniquement l'algorithme de MCS puisqu'il a été utilisé pour la vérification d'un SMA producteur-consommateur et plus spécifiquement le protocole d'interaction de ce SMA.

4.3.2 Model-checking symbolique

Le principal obstacle rencontré par les algorithmes de model-checking est le problème de l'explosion combinatoire du nombre d'états[4]. Plutôt que de représenter de façon explicite l'ensemble des états et des transitions du modèle, les chercheurs ont tenté de les représenter de façon symbolique. On parle alors de MCS. Ce dernier permet de représenter un grand nombre d'états de façon concise et de les manipuler comme s'il s'agissait d'un seul état [Benerecetti and Cimatti, 2001]. Ces ensembles d'états sont représentés par le diagramme binaire de décision[5](BDD).

Le travail présenté par [Wen and Mizoguchi, 1999] à propos de l'utilisation de model checking symbolique pour l'analyse et la vérification des protocoles d'interaction des SMAs, est un exemple concret de l'utilisation des différentes méthodes de MC dans la vérification des SMAs. Ils ont utilisé le model checker symbolique SMV[6] pour analyser et vérifier un simple protocole d'interaction d'un SMA producteur-consommateur qui est spécifié en utilisant le Concurent METATEM comme le montre la figure 4.4. Le protocole est décrit comme suit, le producteur est modélisé par un agent **ap** et les deux consommateurs sont modélisés par des agents **ac1** et **ac2**. Le comportement des agents est défini par un ensemble de règles. Par exemple **ap** peut donner une seule ressource à un seul consommateur à un instant t.

[4]En anglais state explosion problem, l'explosion combinatoire du nombre d'états se manifeste lorsque le graphe représentant le modèle croit rapidement en fonction du nombre d'événements et de variables

[5]Les BDDs sont une structure de données utilisée pour la représentation symbolique d'ensemble d'états

[6]SMV est un système qui a été développé par le groupe de model checking de CMU à l'Université de Carnegie-Mellon. Il est considéré comme le premier outil qui a pu, grâce à une représentation interne du graphe de transition par la structure de données BDD, vérifier de façon exhaustive que des systèmes de taille très élevée satisfaient leurs spécifications CTL [URL1,].

Légende :

1. $\circ$: indique qu'on est à l'étape courante;
2. $\diamond$:éventuellement;
3. $\forall$: toujours;
4. Les arguments entre () sont les messages entrants, et ceux entre [] sont les messages sortants;

ap(ask1,ask2)[give1,give2] :

1. $\circ$ ask1 $\Rightarrow \diamond$ give1;
2. $\circ$ ask2 $\Rightarrow \diamond$ give2;
3. start $\Rightarrow \forall$!(give1$\wedge$give2);

Si les messages entrants à l'agent ap sont ask1 ou ask2, alors trois réponses sont possibles : la première (respectivement, la deuxième) est que dans l'état courant l'agent ap reçoit le message ask1 (respectivement, ask2) de l'agent ac1 (respectivement, ac2), alors éventuellement l'agent ap doit répondre par give1 (respectivement give2). La troisième réponse se déclenche lorsqu'on est au début du protocole d'interaction.

ac1(give1)[ask1] :

1. start $\Rightarrow$ ask1;
2. $\circ$ ask1 $\Rightarrow$ ask1;

Si le message entrant à l'agent ac est give1, alors deux réponses sont possibles : la première réponse se déclenche lorsqu'on est au début du protocole d'interaction, la seule action que l'agent ac peut faire est d'envoyer un message ask1.

ac2(ask1,give2)[ask2] :

1. $\circ$(ask1$\wedge$!ask2) $\Rightarrow$ ask2;

FIG. 4.4: Le protocole d'interaction du SMA producteur-consommateur

Le processus de cette vérification se compose de la façon suivante :
Modélisation du protocole d'interaction : le modèle du système a été extrait à partir du code source du système. Ce modèle est illustré par un graphe de transition d'état. Dans ce graphe de transition, chaque agent est représenté par une machine d'état concurrente (Voir FIG. 4.5).
Traduction du graphe de transition d'état en un programme SMV : les spécifications dans SMV sont des formules de logique temporelle basées sur CTL. L'architecture globale du protocole est décrite par le module *main* et chacune des machines à états est transformée en processus :

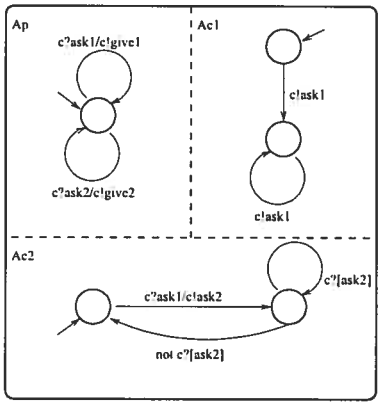


FIG. 4.5: Le graphe de transition d'état du protocole producteur-consommateur

```
MODULE main -- Ap
VAR
Ap: process provider(Ac1.mes,Ac2.mes)
Ac1: process consume1(Ap.mes)
Ac2: process consume2(Ac1.mes,Ap.mes)
-------------
MODULE provide(Ac1mess_in,Ac2mess_in)
VAR
....
ASSIGN
...
-------------
MODULE consume1(pcmess_in)
VAR
mes:{ask1};
ASSIGN
init(mes):=ask1;
next(mes):=ask1;
-------------
MODULE consume2(ac1mess_in,pcmess_in)
VAR
...
ASSIGN
...
```

Le bloc **ASSIGN** définit l'ensemble des règles de transition. Les méthodes **init** et **next** définissent respectivement la valeur initiale et suivante d'une variable.

Spécification des propriétés : la description des propriétés se fait par le biais de formules CTL. Une des propriétés qui a été vérifiée dans ce protocole est l'interblocage – *les agents ac1 et ac2 demandent infiniment une ressource à l'agent ap* –. La formule CTL de cette propriété est la suivante :

```
SPEC AG ( AF (ac1.mes = ask1))
SPEC AG ( AF (ac2.mes = ask2))
```

Utilisation du model-checker de SMV : après la spécification du système et des propriétés, on exécute le model-checker de SMV afin de détecter la présence de ces propriétés dans le modèle. Selon l'article [Wen and Mizoguchi, 1999], toutes les propriétés ont été satisfaites.

4.4 Synthèse et choix de la technique de vérification

4.4.1 Synthèse

Le tour d'horizon que nous venons de faire sur les approches de vérification des SMAs, nous a permis de recenser les techniques existantes et d'avoir une meilleure idée de ces dernières : leurs principes de fonctionnement, leurs avantages et inconvénients et leurs champs d'application. Certes et à notre connaissance, il n'existe pas d'approches propres à la vérification des systèmes orientés agents, mais il commence à y avoir quelques ébauches d'idées pour inclure des méthodes, destinées au début pour la vérification des systèmes classiques ou distribués, dans les plates-formes multiagents, aidant ainsi le programmeur à s'assurer de la correction du système développé durant tout son cycle de vie.

Comme mentionné auparavant, deux catégories d'approches existent pour la vérification des systèmes multiagents : axiomatique ou sémantique. L'approche axiomatique permet de traiter et vérifier les systèmes qui sont déjà développés, impliquant ainsi une limitation de son champ d'application à la phase finale du cycle de développement du système. Ceci augmentera significativement le coût de la vérification. De plus, cette approche s'applique uniquement aux systèmes écrits dans un langage procédural et elle est difficilement applicable aux systèmes orientés-objets [Sommerville, 1996]. Par contre, la technique de model-checking est une technique qui peut être utilisée dans les différentes phases de développement du système. En effet, elle permet de vérifier soit l'exactitude de la spécification, soit la conformité du système à sa spécification. Ceci confère un champs d'applications plus large à cette méthode.

Cependant, la technique de model-checking se heurte aux deux problèmes suivants: le premier est qu'elle se limite à la vérification des systèmes dont la taille est finie et le deuxième est celui de l'explosion combinatoire de l'espace d'états. Ce dernier a été résolu par la représentation symbolique de l'ensemble d'états et des transitions du modèle. En revanche, la limite de la taille du système à vérifier reste toujours une contrainte. Il est en effet impossible de faire une énumération exhaustive d'un nombre d'états infini.

Néanmoins, le model-checking est une technique qui est entièrement automatisable,

plus complète et plus simple que celle de la vérification par méthode axiomatique. En effet, l'axiomatisation des SMAs, ayant des aspects de concurrence et de non déterminisme, n'est pas évidente. De plus, nous sommes contraints d'élaborer une preuve pour chaque propriété recherchée dans le système. Toutefois, le MC possède l'avantage de permettre une détection rapide et économique des erreurs de conception dans des systèmes complexes.

Le tableau suivant est un résumé des similitudes et des différences entre les deux approches de vérification.

	Approche axiomatique	**Approche sémantique**
Langage du système	Système développé avec un langage procédural	Pas de limite
Utilisation	Fin du cycle de développement	Au cours du cycle de développement (au début, en vérifiant la spécification et à la fin, en vérifiant la conformité du système à sa spécification)
Application	Code source du système	Modèle du système extrait de la trace d'exécution, du code source du système ou simplement de la spécification
Fonctionnement	Semi-automatique	Automatique
Facilité	Difficile à utiliser, exige une grande connaissance des notions mathématiques et une compétence lors de l'élaboration des preuves	Plus facile à utiliser

TAB. 4.1: Synthèse des approches de vérification

4.4.2 Choix de la technique

Le model-checking demeure une solution attrayante permettant la vérification des systèmes temps réels, distribués, parallèles, multiagents, etc. Néanmoins, le MC a été utilisé comme une technique de vérification statique et a été appliqué à des modèles extraits de la spécification des systèmes ou du code source du système et non pas à des systèmes réels. D'une part, les modèles extraits de la spécification ne permettent pas de représenter tout ce qui peut se passer dans la réalité. En effet, les systèmes réels sont influencés par un environnement non contrôlé et parfois non-déterministe. C'est

ainsi que les modèles de ces systèmes ne reflètent qu'une partie du comportement de cet environnement. D'autre part, l'extraction des modèles à partir du code source est coûteuse puisqu'elle nécessite une compréhension et une maîtrise du code.

Afin de tirer profit de la puissance de MC, contourner les limites citées au paragraphe précédent et capter le comportement effectif du système et de l'influence de l'environnement, une exécution du système s'imposait pour extraire un modèle d'exécution reflétant la totalité du comportement du système ; et par conséquent analyser le modèle d'exécution réel au lieu du modèle de spécification.

L'approche élaborée est donc basée sur l'utilisation du MC. Ce choix implique une vérification «post-mortem». Cette vérification permet de ne pas dégrader les performances du système et ne pas perturber son comportement. En outre, la vérification «on-line» peut être confrontée au problème de trace partielle puisque, nous sommes amenés à déterminer le temps opportun pour la vérification d'une propriété. Parfois la propriété recherchée peut être vérifiée pour la trace collectée à l'instant t et non pas à l'instant $t+1$ ou inversement. En outre, la vérification «post-mortem» nous permet d'effectuer des traitements coûteux des données de la trace sans perturber l'exécution. Les seules perturbations de l'exécution sont celles occasionnées durant la phase de collecte et leur nuisance peut être limitée en choisissant la meilleure architecture.

4.5 Conclusion

Dans ce chapitre, nous avons présenté les principales méthodes qui ont été utilisées pour la vérification des systèmes multiagents. Nous avons fait une analyse de ces approches et une synthèse générale sur les procédés utilisés par ces derniers. Puis, nous avons tiré des conclusions afin de s'en servir dans l'élaboration de notre approche de vérification.

Nous proposons dans notre choix de solution une approche basée sur l'utilisation de la technique de model-checking. Nous décrivons, dans le chapitre qui suit, cette approche ainsi que ses différentes phases. Enfin, nous présentons les différents problèmes rencontrés lors de l'élaboration de cette dernière, les solutions apportées et notre contribution.

Chapitre 5

Description de l'approche de vérification

Vérifier le comportement global d'un SMA ne revient pas à vérifier le comportement individuel de chacun de ses agents mais plutôt analyser le comportement résultant de la collaboration et la coopération de tous les agents. Comme mentionné dans le deuxième chapitre, nous avons choisi de vérifier le comportement externe des agents en étudiant leurs interactions. Basée sur l'utilisation de la technique du model-checking, l'approche que nous avons définie a nécessité un mécanisme d'observation des exécutions multiagents pour extraire le modèle du système. Ce choix a été motivé par le fait que ces traces permettent de capter une vue aussi fidèle que possible du comportement des agents et de leurs interactions avec l'environnement.

Dans ce chapitre, nous présenterons notre approche de vérification ainsi les différentes phases qui la composent. Puis nous décrirons les paradigmes et les problèmes rencontrés lors de l'élaboration de ces phases. Notamment, nous détaillerons les difficultés inhérentes à l'observation d'une exécution distribuée et les différentes techniques pour déterminer l'ordre dans lequel se produisent les événements de cette exécution. Nous nous intéresserons aussi aux différentes manières d'observer, de capturer et d'enregistrer les informations recueillies dans des journaux d'événements. Puis, nous développerons notre démarche d'observation des interactions du SMA et de collecte de traces qui permettra la construction du modèle d'exécution. Par la suite, nous définirons le processus de modélisation de la trace d'exécution. Subséquemment, nous donnerons une classification des propriétés dynamiques qui permettent d'assurer un meilleur comportement du système et nous décrirons les formalismes dans lesquels ces propriétés seront réécrites. Enfin, nous expliquerons le mécanisme de fonctionnement de l'outil.

5.1 Présentation de notre approche

Le principe général de cette approche est fondé sur la détection et la reconnaissance des anti-patterns d'interaction dans le modèle extrait de la trace d'exécution du système. Cette trace a été obtenue après une observation et un filtrage des événements inhérents aux interactions entre agents. Le choix de l'architecture d'observation est primordiale pour garantir une vue globale du système.
Nous désignons par anti-pattern d'interaction toute instance indésirable de protocole d'interaction conduisant à un comportement erroné du système et représentant des échanges de messages entre agents ainsi que le contenu de ces messages. Nous avons choisi de définir des anti-patterns au lieu de patterns d'interactions pour deux raisons majeures : la première est due au fait qu'il est plus facile de vérifier qu'un système présente un comportement erroné qu'un bon comportement. Par exemple, il est plus facile de diagnostiquer une maladie chez un patient que de garantir qu'il est en parfaite santé. La deuxième porte sur le nombre de patterns qui sera considérablement plus important que celui des anti-patterns.
Grâce à cette approche de vérification, nous avons pu s'assurer du bon déroulement et fonctionnement des protocoles d'interaction et par conséquent de comprendre les comportements, les stratégies adoptées par les agents et leurs mécanismes de raisonnement.

Fondée sur l'utilisation du MC, notre approche est composée des cinq phases qui seront détaillées dans les sections qui suivent (Voir FIG. 5.1) :

- La première étape consiste à proposer un mécanisme pour l'observation et la collecte de traces d'un SMA, permettant la prise en compte de la précédence entre les différents événements invoqués lors des interactions.
- La deuxième étape consiste à modéliser la trace du SMA par un ensemble de processus de communication en SDL.
- La troisième et la quatrième étape vise à identifier et à formaliser les anti-patterns d'interactions au moyen du langage GOAL.
- La cinquième étape est l'utilisation du simulateur de l'outil du MC pour faire une simulation exhaustive des modèles SDL avec une indication des anti-patterns recherchés dans la trace. Ce simulateur construira le graphe d'état global du système qui couvrira toutes les exécutions possibles du système en générant l'entrelacement[1] des exécutions respectives de chacune des composantes de système (Voir FIG. 5.2 et 5.3).

[1] interleaving

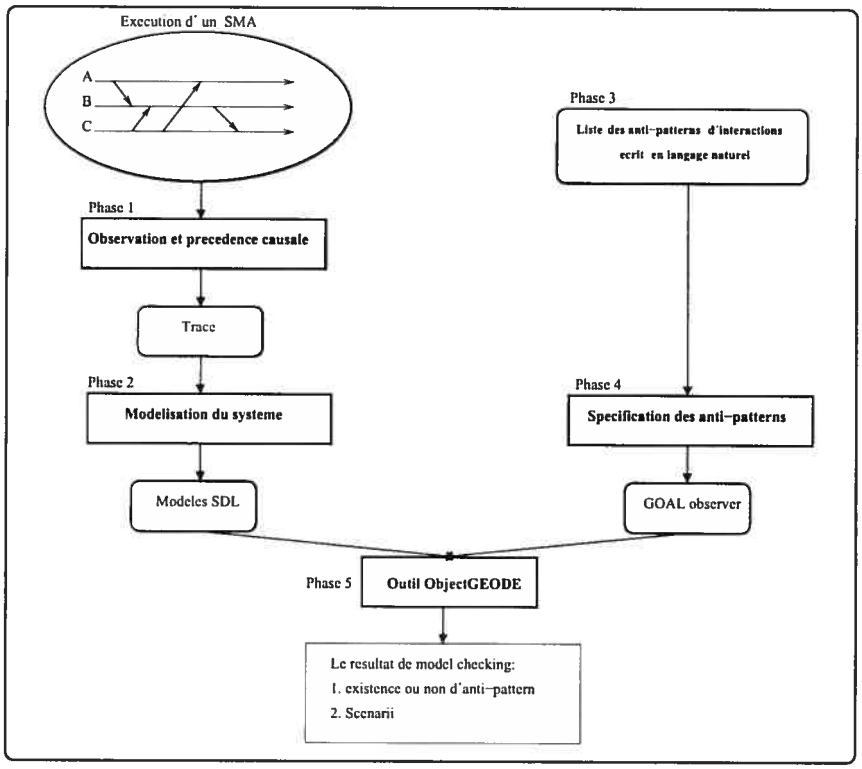


FIG. 5.1: Les différentes phases de notre approche

L'outil utilisé pour la vérification des exécutions des SMAs par le biais des interactions, est objectGEODE, qui a été mis à notre disposition, par l'équipe d'analyse des systèmes distribués (ASD) du CRIM. Cet outil nous a permis de représenter la trace d'exécution sous forme de modèle SDL. Puis, de représenter les propriétés par un *observer* en GOAL. Ensuite, nous avons vérifié l'existence ou non de ces propriétés dans le graphe d'état global construit à partir de ces modèles SDL. Nous avons pu utiliser cet outil, destiné au début à l'analyse de systèmes distribués asynchrones, vu que l'exécution d'un SMA peut être considérée comme une exécution distribuée et l'agent peut être représenté par un ou plusieurs processus. L'analyse du modèle du système est donc faite en utilisant le simulateur et le model-checker de cet outil.

Cette approche a été appliquée au SMA du projet IBAUTS – *qui est un contrôleur multiagents non-déterministe renfermant des éléments non modélisables comme l'évolution de la température à l'extérieur de la pièce, le nombre d'individus occupant la pièce à chaque instant, etc* (Voir §6.2) – afin de vérifier l'absence des anti-patterns.

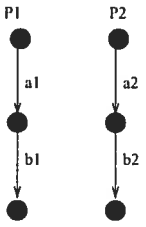


FIG. 5.2: Deux processus s'exécutant en parallèle

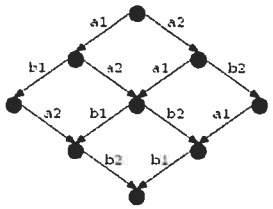


FIG. 5.3: Treillis d'entrelacement des actions de P1 et P2

5.2 Première phase : observation d'une exécution multiagents

La génération et l'analyse de la trace d'exécution de n'importe quel système nécessite un mécanisme d'observation des événements pertinents[2]. En général, cette observation se fait par le biais d'un processus automatisé qui récolte et filtre les informations et les messages d'exécution. Les premiers travaux d'observation étaient centrés sur l'étude et l'analyse des exécutions séquentielles qui sont considérées comme un processus assez simple par rapport à l'observation des systèmes actuels. De part l'aspect réparti et parallèle de tels systèmes, l'observation est devenue un axe de recherche assez développé englobant ainsi la collecte et l'analyse des traces.

Dans ce qui suit nous allons décrire les difficultés liées à l'observation d'une exécution répartie et parallèle ainsi que les situations d'incohérence que peuvent présenter de telles exécutions. À la fin de cette section, nous présentons notre architecture d'observation.

5.2.1 Paradigmes d'observation

Ce qui rend important le processus d'observation au niveau de notre approche est le fait qu'une mauvaise observation peut fausser l'étape d'analyse et de compréhension d'un problème. En effet, une bonne observation, et par conséquent une bonne compréhension du comportement de l'exécution, facilite énormément la découverte des erreurs et par la suite leur correction. La qualité des résultats de ce processus est de plus en plus indispensable pour la vérification post-mortem.
La pertinence de cette étape d'observation par rapport au processus de vérification est similaire à celle de la pertinence d'une décision lors de sa prise. En effet, une décision dépend de la quantité et de la qualité des informations disponibles au moment de la

[2]Nous voulons dire par événement, toute occurrence d'envoi ou de réception d'un message par un agent. Par contre, le message est un ensemble de données échangées entre les différents agents.

prise de cette décision.

Définition de l'observation

Le processus d'observation consiste essentiellement à représenter et/ou à sauvegarder les informations sur les composants, les processus et les données du système à vérifier ainsi que leurs évolutions et leurs interactions. Un des buts de ce processus est la collecte de l'historique des messages échangés entre les agents à des fins d'analyse. Cette collecte est souvent effectuée par un module dédié appelé moniteur, que nous appellerons dans ce mémoire «observateur», capable d'identifier les entités participantes à une exécution et de capter, aussi fidèlement que possible, l'état réel du système observé.

On distingue deux catégories d'observateurs : les observateurs physiques (les sondes matérielles, les capteurs, etc) et les observateurs logiques (les programmes espions, etc). Nous nous intéressons uniquement aux observateurs logiques qui peuvent être soit :

- Répartis : chaque composante dont nous voulons sonder le comportement, détient un module d'observation local qui garde une trace locale à chaque fois qu'elle exécute une action ou une interaction observable. Cet observateur achemine ensuite ces traces à un module de jounalisation central qui permet de construire la trace globale du système (à partir des différentes traces locales). La construction de la trace globale peut se faire on-line ou off-line.
- Centralisés : on crée une nouvelle composante, différente de celles observées, disposant d'un module d'observation global qui sera notifié de l'occurrence de certains événements pertinents.

Difficultés de l'observation d'une exécution répartie

Plusieurs problèmes et difficultés surgissent lors de l'observation des systèmes dont l'exécution est répartie et parallèle. D'abord, l'observation est confrontée au problème de la variation du comportement de l'exécution. En effet, le comportement d'un programme parallèle est souvent non-déterministe. Il dépend des conditions particulières d'exécution (ordonnancement des processus, charge des médias de communication, le temps de transit, etc.), qui peuvent changer d'une exécution à une autre et modifier ainsi l'ordre des opérations de communication et de synchronisation. Ceci dit, nous ne pouvons pas garantir que le comportement d'une deuxième exécution du système va donner le même comportement et les mêmes résultats que la première. Cette incertitude fait apparaître un facteur aléatoire dans l'obtention du comportement erroné de

l'exécution. De plus, ce phénomène de variation du comportement rend plus difficile la correction des erreurs parce que la localisation de ces dernières sera d'autant plus difficile. En outre, nous ne pouvons pas nier que l'observation est une forme d'intrusion et que ceci risque de perturber et modifier l'exécution du système. Plus spécifiquement, elle risque de ralentir les vitesses d'exécution des processus et par conséquent l'ordre des événements. Une autre difficulté dont il faut tenir compte est le fait que le contrôle des programmes répartis n'est pas immédiat à cause des décalages entre la production d'une information et sa prise en compte. Enfin, de part la nature répartie des systèmes, l'absence d'un état global dans l'exécution de ces systèmes est inévitable. Ceci engendre un problème lors de la conservation de l'ordre des événements.

Toutes ces difficultés font en sorte qu'il est important de disposer de méthodes et de techniques qui permettent d'observer les phénomènes dynamiques engendrés par l'exécution de tels systèmes, tout en tenant compte de ces aspects contraignants. Dans la section qui suit, nous décrivons l'impact de ces difficultés sur le processus d'observation en détaillant quelques situations d'incohérence.

Situations d'incohérence lors de l'observation d'une exécution répartie

L'observation d'un système distribué tel qu'un SMA, pose le problème de l'absence d'un référentiel de temps absolu permettant d'établir un ordre total entre les événements et de garantir l'exactitude de la précédence et de la dépendance causale entre ces événements. Ce problème est d'autant plus important lors de l'observation d'un système distribué asynchrone –*le terme asynchrone signifie que le système ne dispose d'aucune horloge globale, aucune hypothèse n'est faite concernant les vitesses d'exécution relatives des différents sites, ni concernant les délais de délivrance des messages par le réseau, et l'émission et la réception d'un message sont deux actions distinctes*–.

De plus, lors de l'observation et la collecte de trace, on ne peut pas observer tous les messages pour plusieurs raisons. D'une part, les messages échangés entre agents ne sont pas toujours pertinents pour la phase de vérification. Et d'autre part, si nous sauvegardons tous ces messages alors la taille de la trace deviendrait trop volumineuse, et cela compliquerait la phase de vérification. D'où la nécessité de filtrer les événements observables. Cependant, le filtrage de ces événements observables rend de plus en plus difficile la détermination de l'ordre dans lequel se produisent les événements d'une exécution répartie.

Dans ce qui suit, nous exposons quelques situations d'incohérence qui peuvent surgir lors de l'observation d'une exécution répartie. Cette partie qui expliaue en détail ces différentes situations est extraite de la thèse de [Mazouzi, 2001].

Pour comprendre les différentes figures qui suivent, nous allons décrire les composants de ces dernières (exemple figures 5.4 et 5.5 : les lignes verticales représentent les axes temporels associés aux sites ou aux processus, les flèches continus représentent les messages échangés entre les agents et les flèches discontinus représentent la copie du message qui sera acheminée à l'agent observateur. Les événements sont représentés par des points.

Plusieurs vues différentes de la même exécution : la perception de l'ordre des événements d'une exécution particulière par deux ou plusieurs observateurs peut être différente sans pour autant que ces interprétations soient fausses. On parle ainsi de deux vues différentes de la même exécution. En effet, la figure 5.4 illustre une situation où deux observateurs O et R sont notifiés suite à l'occurrence des événements b et c. Les délais de communication associés aux messages font que l'observateur O reçoit la notification de l'événement b avant celle de c, alors que R observe un ordre différent.

Perception incohérente des ordres causaux : la perception de l'ordre des événements par un seul observateur peut être erroné. Cette mauvaise perception affecte la dépendance causale des événements. En général, ce problème est dû aux contraintes temporelles lors de la propagation des messages. Par exemple, lorsque l'événement d'émission d'un message est perçu chez l'observateur après celui de sa réception. La figure 5.5 illustre cet exemple, l'observateur R perçoit l'occurrence de l'événement c (réception du message) avant celui de a (envoi du message), alors que c'est le contraire qui s'est produit.

Ordonnancement selon le temps physique : un des problèmes posés par une exécution répartie est l'absence d'une horloge globale. Parfois et pour y remédier, l'observateur linéarise les événements, c'est-à-dire qu'il dispose les événements dans le temps réel pour les ordonner ; mais rien ne peut certifier que ce temps permet une bonne observation (Voir FIG. 5.6).

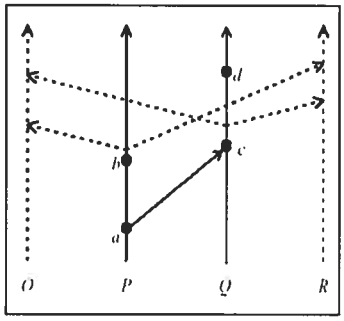


FIG. 5.4: Plusieurs vues différentes de la même exécution

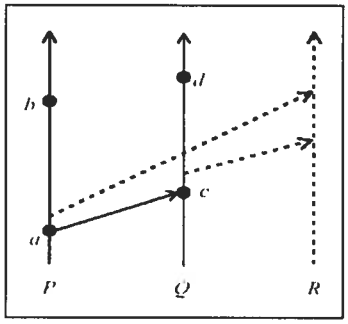


FIG. 5.5: Perception incohérente des ordres causaux

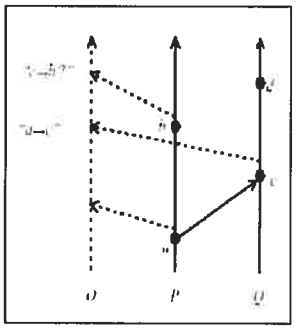


FIG. 5.6: Ordre défini arbitrairement par l'observateur

Récapitulatif

Il est clair que l'observation d'une exécution d'un système distribué asynchrone ne préserve pas toujours l'ordre réel inhérent à l'exécution des événements. En effet, et par opposition aux systèmes séquentiels, nous ne pouvons pas avoir un ordre total entre deux événements venant de deux processus distribués puisque nous avons de la difficulté à maintenir une notion de temps absolu cohérente. De plus, les événements concurrents ne sont pas nécessairement dépendant et nous ne pouvons donc pas établir un ordre entre eux. La relation définie entre les événements est un ordre partiel.

Dans ce qui va suivre, nous allons définir la notion de causalité et présenter des mécanismes permettant ainsi de préserver l'ordre des événements.

5.2.2 Ordre partiel et technique d'estampillage

Définition de l'ordre partiel

L'ordre partiel est établi par la relation «happened before», nommée aussi précédence causale et notée par $\rightarrow$. Il a été défini par Lamport [Lamport, 1978]. Cette notion a été, par la suite, à la base de beaucoup de travaux. Le principe de la précédence causale est que :

- Les événements locaux d'un processus peuvent être ordonnés en se basant sur l'ordre de leur exécution ou selon le temps absolu.
- La date d'émission d'un message sur le site émetteur précède toujours sa réception sur le site récepteur du point de vue de l'observateur.

La précédence causale ne permet pas l'ordonnancement des événements concurrents. Il faut noter la différence entre la notion de précédence causale et celle de dépendance causale. En effet, la précédence nous informe seulement sur l'ordre des événements, par contre la dépendance est une relation de cause à effet. Si on a une dépendance entre deux événements, on aura forcément la relation de précédence entre eux. L'inverse n'est pas toujours vrai.

Les techniques utilisées pour définir l'ordre de la causalité des événements trouvent leurs fondements dans le calcul des estampilles (timestamp) des événements. Ces estampilles sont obtenues en ajoutant de l'information aux messages de l'application (mécanisme de piggybacking), plus précisement, en datant les messages avec une notion de temps logique. Il existe plusieurs solutions dans la littérature qui permettent de définir un référentiel de temps absolu. Parmi ces solutions, nous trouvons l'horloge logique [Lamport, 1978], vectorielle [Fidge, 1988] [Mattern, 1989] et matricielle [Wuu and Bernstein, 1984].
L'horloge logique ou linéaire de Lamport assure la précédence causale et les horloges vectorielle et matricielle garantissent la précédence ainsi que la dépendance causale. Les horloges matricielles sont les plus efficaces mais aussi les plus coûteuses à implanter. Par exemple, pour un système distribué comprenant n sites, l'algorithme de calcul de l'horloge matricielle impose de munir chaque site et chaque message échangé d'une estampille représentant une matrice de $n \times n$ entiers. Dans tous les cas, l'évolution de l'horloge doit respecter l'ordre de causalité de l'exécution, puisque en définissant une relation d'ordre sur les estampilles, nous définissons une relation sur l'ensemble des événements observés.

Technique d'estampillage : horloge linéaire

Dans l'article où il défini la relation d'ordre partiel, Lamport a présenté également le concept de l'horloge logique permettant de dater les événements dans le but de les ordonner. Il s'agit d'un algorithme d'estampillage défini comme suit :

- Un compteur entier L_i, initialisé à 0, est maintenu sur chaque site ou processus P_i.
- À l'occurrence d'un événement local e sur un processus P_i, la valeur de L_i est incrémenté de 1.
- Si e est l'émission d'un message m, celui-ci est estampillé par la date de son émission sur P_i (L_i).
- Dès que le processus P_j reçoit ce message, l'estampille L_j associée à l'événement sera égale à : $L_j = \max(L_i, L_j) + 1$

Cet algorithme est illustré par la figure 5.7 qui représente un diagramme «espace-temps». Cette dernière montre correctement comment la technique d'estampillage de Lamport préserve la précédence causale. Prenons l'exemple des événements $e_{1,1}$ et $e_{2,3}$: $e_{1,1} \rightarrow e_{2,3}$ et nous avons bien $L(e_{1,1}) < L(e_{2,3})$; par contre, la relation de dépendance causale n'est pas assurée. En effet, malgré que $L(e_{1,1}) < L(e_{2,2})$, cela n'implique pas que $e_{1,1} \rightarrow e_{2,2}$.

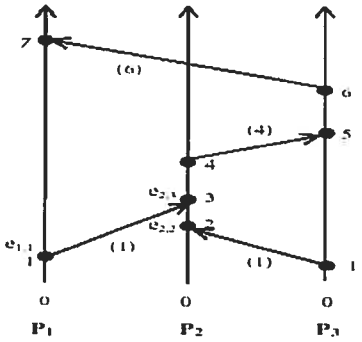


FIG. 5.7: Illustration de l'algorithme d'estampillage de Lamport

Cet algorithme d'estampillage, même s'il ne permet pas de définir exactement la relation de causalité d'une exécution et qu'il est considéré comme une approximation des dépendances, est intéressant puisqu'il est facile à mettre en oeuvre. En outre, cet algorithme minimise le fait qu'on influence la vitesse relative d'exécution des processus, étant donné qu'il demande peu de calculs et les messages sont estampillés par des entiers.

5.2.3 Journalisation des événements

Les problèmes de la détermination de l'ordre des événements et de l'appariement (matching) des événements d'envoi et de réception sont donc cruciaux pour garantir une bonne analyse. En effet, lorsque nous observons ou nous élaborons un faux ordre dans la trace d'exécution d'un système, cela peut alors affecter considérablement le résultat de l'analyse et de la vérification de ces traces. Comme nous l'avons vu précédemment, ce problème peut être résolu en datant les messages avec une notion de temps logique.
Dans cette section, nous allons décrire les architectures de journalisation qui existent.

Architectures pour la journalisation

Le processus de l'analyse et de l'appariement des événements peut être «on-line» ou «off-line». Dans le cas où l'analyse se fera de façon «off-line», les événements doivent

être enregistrés dans un journal qui doit contenir suffisamment d'informations pour permettre la localisation des erreurs potentielles. Deux architectures de journalisation peuvent être considérées : la journalisation distribuée ou la journalisation centralisée.

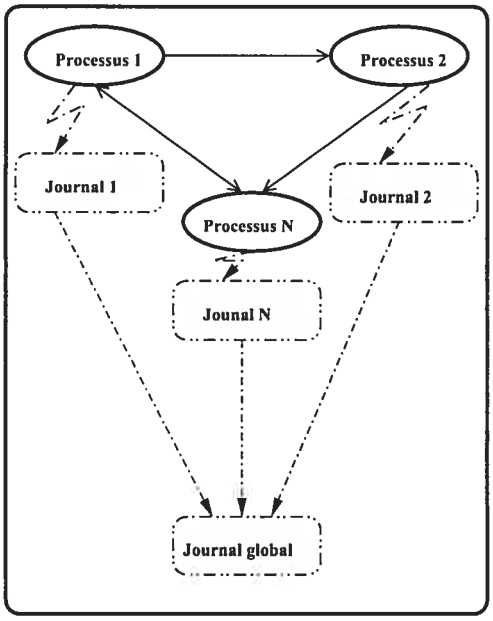


FIG. 5.8: Jounalisation distribuée

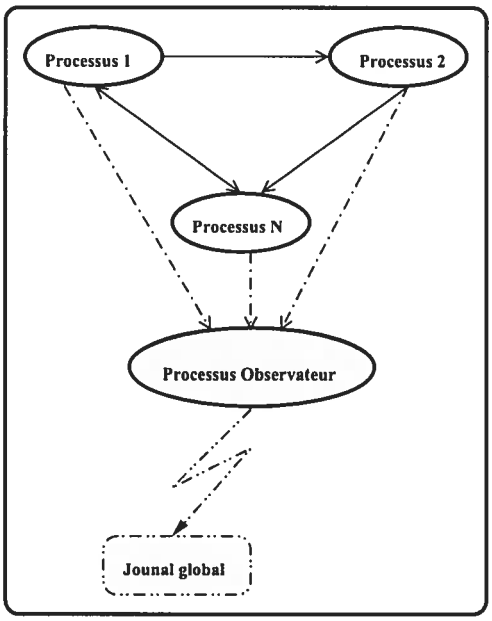


FIG. 5.9: Jounalisation centralisée

La solution distribuée (Voir FIG.5.8) implique que chaque agent ou processus conserve son propre journal. Dans ce cas, on aura autant de journaux locaux que de processus. Certes, cette solution diminue le risque de congestion au niveau du processus de conservation et minimise le phénomène de goulot d'étranglement sur celui-ci puisqu'elle ne nécessite pas le transport des événements vers le site de conservation. Cette solution est intéressante dans le cas où le nombre de processus ou des sites observés est important, ou si le nombre de messages échangés est grand. Cependant l'instrumentation de la trace se fera à deux niveaux : au niveau local où chaque agent ou processus récupère sa propre trace ; et au niveau global où il y a une récupération des différentes traces locales et un appariement entre les événements d'envoi et de réception de message.

La solution centralisée (Voir FIG.5.9) consiste à avoir une seule composante qui est notifiée par l'occurrence des événements observables et qui a comme mission de sauvegarder ces événements dans une trace globale. Cette solution peut provoquer une congestion au niveau du processus de conversation. Cependant l'appariement et l'ordonnancement des événements se fait au fur et à mesure de l'enregistrement. De plus, elle permet de minimiser la perturbation des comportements des composantes observées.

Nature des événements observables

Quelle que soit l'architecture choisie pour la journalisation des événements, le journal doit contenir assez d'informations pour assurer une bonne vérification. C'est pourquoi il a tendance a être volumineux. Cependant, une importante taille du journal peut présenter des inconvénients :

- Nous ne pouvons pas nier que l'observation, la collecte et la journalisation des informations de trace est une forme d'intrusion. Par conséquent, plus on journalise d'informations, plus on risque de perturber l'exécution du programme.
- Les informations sont enregistrées dans le but d'être analysées. Cette analyse sera d'autant plus longue que le journal sera volumineux. En outre, l'utilisation de la technique du MC pourrait engendrer une explosion du nombre d'états lors de la construction du graphe global à partir de la trace à analyser.

Quelques travaux ont essayé de trouver des solutions pour réduire le volume des informations conservées dans l'historique. Par exemple, [Garcia-Molina *et al.*, 1984] proposent de ne conserver que les informations importantes, sous la forme d'une trace ordonnée d'événements de haut niveau. Ces événements sont les actions macroscopiques faisant intervenir le système (la création et la destruction d'un processus, l'envoi et la réception de messages, l'allocation et la libération d'une ressource, etc). Par contre, les actions microscopiques ou internes au programme telles que le changement de valeur d'une variable, ne sont pas journalisées.

Pour minimiser le volume des informations à conserver, nous avons par ailleurs choisi de ne conserver que les informations strictement nécessaires à l'analyse du comportement des agents lors de leurs interactions. Ces informations contiennent toutes les données nécessaires pour détecter tous les anti-patterns que nous avons identifiés.
Les événements sont ceux permettant de connaître la création des agents et ceux émanant des interactions observées en termes d'émission et de réception de messages. Nous donnons avant la fin de ce chapitre la liste exacte des événements conservés dans l'historique d'une exécution d'un SMA. Dans ce qui suit, nous allons détailler notre méthode d'observation.

5.2.4 Méthode d'observation des interactions dans les SMAs

Avant d'exposer la méthode d'observation que nous avons élaborée pour la collecte et la génération de la trace d'exécution, nous présentons tout d'abord les conditions que

doit remplir, à notre avis, la phase d'observation des protocoles d'interactions dans les SMAs.

Conditions d'observation

What you get is what happened : l'observation de l'état de l'exécution d'un SMA doit refléter l'état réel du système, de la manière la plus fidèle possible. Comme nous avons choisi d'implanter nos observateurs sous forme d'agents (Voir §5.2.4), cela signifie en particulier que la liaison entre l'agent ou les agents responsables de l'observation et les autres agents doit être fiable c'est à dire que l(es) agent(s) observateur(s) ne perdent pas les messages et qu'ils doivent avoir une vue cohérente de l'exécution.

Filtrage : il est très important que l'agent observateur ne sauvegarde pas toutes les informations et que le programmeur puisse choisir les événements qu'il juge pertinents pour le processus de vérification. De plus, nous devons aussi enregistré juste les informations nécessaires qui nous permettent de détecter les anti-patterns. Ce qui nécessite une connaissance du domaine d'application. En d'autres termes, la quantité d'informations produites dans le fichier de journalisation doit être minimale mais suffisante pour éviter l'excès d'informations et assurer une bonne analyse.

Description de notre méthode d'observation

L'architecture utilisée pour la collecte des informations est l'architecture centralisée (Voir §5.2.3). Ce choix a été fortement influencé par la nature et la caractéristique de l'application à analyser. En effet, le SMA IBAUTS est un système qui permet d'automatiser les systèmes de chauffage et de ventilation des édifices. Par conséquent, nous n'avons pas de périphériques de stockage dans les cartes électroniques se trouvant dans les pièces de l'immeuble. Cette contrainte fait en sorte qu'il est impossible de choisir l'architecture centralisée. De plus, ce choix a été aussi motivé par le fait que nous ne voulons pas charger les agents par des modules locaux, perturbant ainsi leurs comportements. Nous avons défini alors un seul agent observateur qui se charge de logger la trace d'exécution, en appariant les événements d'envoi et de réception des messages grâce à l'horloge de Lamport (Voir FIG. 5.10). L'enregistrement des événements se fait comme suit : dès que l'agent observateur reçoit un événement d'envoi, il le stocke momentanément dans une structure de données en attendant la réception de l'événement intrinsèque à la réception du message envoyé. Suite à cette réception, les événements seront stockés dans le fichier de la trace. Sinon les événements qui n'ont pas de paire (c'est-à-dire un événement d'envoi de message n'a pas sa correspondance (l'événement

de réception de ce message)), seront stockés à la fin du fichier. Pour l'ordonnancement de ces événements, nous nous sommes basés sur les estampilles qui ont été calculées par les agents observés.

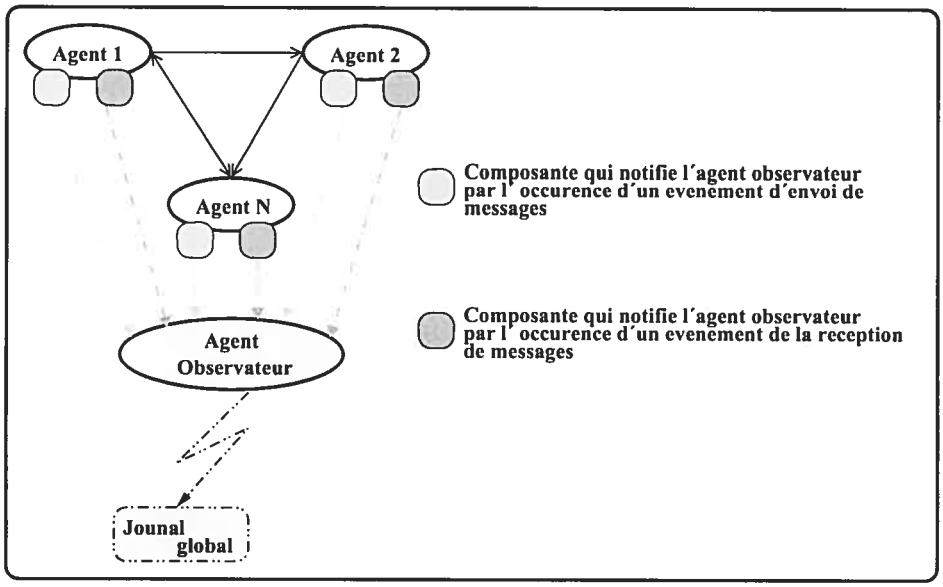


FIG. 5.10: Architecture de la méthode d'observation

La trace est obtenue en ajoutant aux endroits adéquats dans le programme, le code nécessaire à la collecte d'informations pertinentes. En effet, nous avons défini deux modules qui s'abonnent à tout événement envoyé ou reçu dans le SMA. Ces modules s'occupent de prendre une copie du message envoyé (respectivement, reçu) avant son envoi (respectivement, au moment de sa réception) par l'agent et de l'acheminer à l'agent observateur. Après la notification de l'agent observateur de l'occurrence de ces événements, ce dernier se charge de faire la mise en correspondance entre les événements d'envoi et de réception et de les enregistrer dans un fichier.

Une bonne présentation des informations de la trace est primordiale pour l'analyse et la compréhension de cette dernière. C'est pour cette raison qu'il est souhaitable de présenter les informations de trace en utilisant un langage adéquat. Dans notre méthode d'observation, nous avons choisi le langage XML pour formuler l'historique. Le choix du langage XML a été fortement motivé par sa souplesse, sa forte structuration, sa lisibilité, sa portabilité et la facilité de transformer les données sous un autre format. Pour plus de détails concernant la grammaire de cette trace, se rapporter à l'annexe B.

Les informations que nous avons observées concernent le flot d'exécution qui correspond à la suite des actions effectuées pendant cette exécution. Ces actions sont illustrées par la communication entre les agents, ce qui nous permettra de connaître les

agents qui interviennent dans l'exécution, leurs interactions et de suivre le comportement du SMA.

Pour les besoins de la vérification, nous avons été amenés à enrichir les messages en ajoutant de l'information supplémentaire aux événements pour déterminer l'ordre et permettre ainsi la vérification de l'exécution. Cet enrichissement a été fait en ajoutant des nouveaux paramètres aux messages. Les événements instrumentés doivent contenir les informations suivantes :

1. Le type de l'événement observable (événement d'envoi ou de réception de message et les événements locaux si il y a eu lieu).
2. L'estampille de l'événement. Cette estampille est calculée en utilisant l'horloge logique de Lamport qui est gérée par les agents observés.
3. L'identifiant du round de protocole. Cet identifiant est nécessaire lorsque nous sommes amenés à vérifier un protocole d'interaction cyclique. En effet, il permet de délimiter les scénarios d'interactions. Cet identifiant est représenté par le paramètre *correlation-id* dans la trace d'exécution du système.
4. L'identifiant de l'agent.
5. L'adresse ou le nom de l'agent source (respectivement, destinataire) pour l'événement d'envoi de (respectivement, réception) sous forme d'URL.
6. Les paramètres du message qui incluent le nom du message, la façon dont il a été communiqué (point à point, diffusion).

Pour illustrer la structure d'un message sauvegardé dans un fichier de trace, nous donnons un exemple des événements intrinsèques à ce message. En effet, chaque message est composé par deux événements celui de l'envoi et de la réception du message.

```
<event timestamp="60" operation=" Send" type="Communication">
<parameters correlation-id="1357.0" message-id="2817.0" thread-id="BridgeAgent"/>
  <message>
    <parameter name="name" type="string" value="Control"/>
    <parameter name="t_outdoor" type="real" value="-13.3"/>
    <parameter name="t_air_supply" type="real" value="13.0"/>
    <parameter name="t_zone" type="real" value="21.762050805466785"/>
    <parameter name="time" type="real" value="13.694444444444445"/>
    <parameter name="day" type="real" value="2.0"/>
    <parameter name="season" type="real" value="1.0"/>
    <parameter name="waitTime" type="int" value="30"/>
  </message>
</event>
<event timestamp="61" operation="Receive" type="Communication">
<parameters correlation-id="1357.0" message-id="2817.0" thread-id="RoomAgent"/>
  <message>
    <parameter name="name" type="string" value="Control"/>
    <parameter name="t_outdoor" type="real" value="-13.3"/>
    <parameter name="t_air_supply" type="real" value="13.0"/>
    <parameter name="t_zone" type="real" value="21.762050805466785"/>
    <parameter name="time" type="real" value="13.694444444444445"/>
    <parameter name="day" type="real" value="2.0"/>
    <parameter name="season" type="real" value="1.0"/>
    <parameter name="waitTime" type="int" value="30"/>
  </message>
</event>
```

FIG. 5.11: Exemple de la trace d'exécution

5.3 Deuxième phase : Modélisation de la trace d'exécution

5.3.1 Outil de modélisation

Suite à la collecte de la trace, la première tâche est de générer les modèles du système à partir de cette trace d'exécution. Pour la modéliser, l'outil TRAYSIS (**TRA**ce Anal**YSIS**) [Hallal and Petrenko, 2001] a été utilisé. Cet outil accepte des traces sous format XML (suivant la grammaire définie dans l'annexe B) et génère les modèles SDL. Il offre la possibilité de choisir entre la génération d'un modèle SDL asynchrone ou synchrone. Ce choix dépend du type de communication entre les entités du système. Les systèmes qui sont basés sur la communication synchrone (respectivement, asynchrone) utilisent la fonction **Build Synchronous Model** (respectivement, **Build Asynchronous Model**). De plus, il permet d'afficher et d'éditer les modèles générés, ainsi que les résultats de vérification et les propriétés à vérifier dans ces modèles.

Le rôle de l'outil TRAYSIS est primordial dans le processus de vérification puisqu'il est considéré comme l'outil qui permet de faire le lien entre la trace à vérifier et l'outil utilisé pour la vérification. Certes, cet outil permet d'analyser des traces sous forme

d'XML suivant une grammaire prédéfinie (Annexe B) et d'extraire des modèles SDL, soient synchrones ou asynchrones. Cependant, pour créer le modèle, la trace doit être syntaxiquement correcte. Dans ce qui suit nous allons décrire la structure des modèles qui seront extraits de la trace d'exécution.

5.3.2 Structure des modèles en SDL

Dans SDL, la structure d'un système peut être modélisée en utilisant la hiérarchie de structures (système/bloc/processus/procédure) décrites dans la section 3.4.2. Dans notre cas, il est suffisant de présenter l'exécution du SMA comme un seul bloc composé de plusieurs processus représentant les agents en interaction. Le comportement global du système est modélisé par le comportement de tout l'ensemble des processus communicants dans SDL. Cependant, le comportement de chaque agent ou processus sera représenté par une machine à états finis qui est linéaire et séquentielle. La représentation d'un agent est obtenue en projetant la trace rassemblée dans l'ensemble de types d'événements : envoyés, reçus et les événements locaux.

Concernant la communication entre les processus, elle est réalisée par l'intermédiaire des canaux et des signaux avec ou sans paramètres représentant les données échangées. Les signaux d'entrée d'un processus sont stockés dans une file d'attente avant leur lecture. Ceci signifie que les médias de communication du système sont simplement modélisés avec les différentes files d'attente des processus SDL. Les messages échangées entre les agents ont donc été modélisés en SDL par des signaux transmis entre les processus sur des canaux bidirectionnels.

Afin d'expliquer le passage de la trace XML en modèles SDL, nous décrirons dans ce qui suit, un petit exemple. Prenons une partie de la trace étudiée (Voir FIG. 5.11). Lors de son premier parsing, nous déterminons le nombre d'agents et par conséquent le nombre de modèles à générer. Dans cet exemple, nous avons deux agents *bridge* et *room*, donc nous aurons deux modèles représentant chacun de ces agents. Ces modèles seront formés par des automates à états finis séquentiels et linéaires. Au deuxième parsing de la trace, nous déterminons les composantes de chaque modèle. Le modèle du *bridgeAgent* contiendra dans ce cas, tous les événements d'envoi et de réception de messages, ainsi les événements locaux et leurs paramètres. Dans le chapitre suivant, nous montrerons un exemple concret de modèle SDL décrivant le comportement et la séquence d'activités établies par l'agent.

5.4 Troisième et quatrième phase : identification et formalisation des anti-patterns

Lors de la vérification d'un système par la technique de model-checking, il est nécessaire d'exprimer des propriétés sur le comportement dynamique du système étudié. Ces propriétés peuvent être de nature qualitative « *le résultat généré est bon ou non?*», ou de nature quantitative « *telle erreur s'est produite n fois durant l'exécution du système*». Dans ce mémoire, nous nous sommes intéressés uniquement aux propriétés qualitatives. D'une part, la technique que nous avons utilisée permet de détecter ce type de propriétés. D'autre part, il suffit de prendre connaissance de l'existence d'une seule erreur pour intervenir. De cette façon, nous pouvons nous assurer que notre système est exempt de comportement indésirable sans pour autant le certifier.

La spécification des propriétés qualitatives diffère d'un système à un autre. Par exemple, pour les systèmes temps réel, la spécification des propriétés fait intervenir de façon explicite des variables temporelles appelées horloges. Par contre, si nous voulons spécifier seulement des propriétés qui font référence à l'ordonnancement des événements, alors la logique temporelle suffit.

Les propriétés que nous pouvons vérifier dans un SMA par la technique de model-checking sont classées en cinq grandes catégories : sûreté, vivacité, atteignabilité, équité et absence de blocage [Berard *et al.*, 2001]. Cette classification dépend du choix de la logique pour la formalisation des propriétés ainsi que du model-checker utilisé. De plus, cette classification cdéfinit une panoplie d'erreurs qu'un système peut avoir et/ou une série de bons comportements qu'un système doit satisfaire. Elle offre ainsi une référence à l'expert pour identifier les anti-patterns qu'il veut vérifier dans un SMA et permet une structuration du processus d'identification des anti-patterns. Nous présentons, dans ce qui suit, les différents types de propriétés dynamiques.

5.4.1 Propriétés d'atteignabilité

Une propriété d'atteignabilité énonce qu'une situation donnée peut être atteinte.

Les propriétés d'atteignabilité ont pour but de déterminer si lors du parcours du graphe, modélisant tous les états d'un système, une situation est ou non accessible. Il peut s'agir d'un état ou d'un ensemble d'états ayant en commun une propriété donnée. Dans le cas du SMA IBAUTS, lorsque nous voulons vérifier si tous les agents responsables des dispositifs de chauffage soumettent leurs offres avec un coût énergétique supérieur au seuil acceptable. Ceci est considéré comme une propriété d'atteignabilité. Il existe deux

types de familles de propriétés d'atteignabilité : l'atteignabilité simple ou conditionnelle.

Atteignabilité simple : les propriétés d'atteignabilité simple sont celles qui décrivent les états qu'il faut ou qu'il est souhaitable d'atteindre. En général, on s'intéresse plutôt aux situations qu'on ne veut pas atteindre et donc c'est la négation de la propriété d'atteignabilité qui est plus intéressante. Un exemple de propriété d'atteignabilité simple dans un SMA est qu'on ne peut pas entrer dans une situation de blocage. Nous pouvons définir une situation de blocage dans un SMA comme un problème d'attente mutuelle de messages : un agent manager envoie un message à un agent contractant. Ce dernier ne reçoit pas ce message. L'agent manager s'attend à une réponse suite à son message sans prendre conscience que son message n'a pas été reçu. Et l'agent contractant reste dans un état d'attente du message envoyé par l'agent manager.

Atteignabilité conditionnelle : les propriétés d'atteignabilité conditionnelle sont plus complexes que les propriétés d'atteignabilité simple puisqu'elles sont assorties de conditions. Un exemple de propriété d'atteignabilité conditionnelle dans un SMA est qu'un agent ne réagit qu'à la suite d'un envoi de message par un agent spécifique.

Cette propriété est exprimée dans CTL par la formule EFp où p est une formule propositionnelle et EF sont les quantificateurs respectivement de chemins et temporels. Cette formule se lit «*il existe un chemin partant de l'état courant et sur lequel il existe un état vérifiant p*». Par contre, la logique LTL n'est pas adaptée à l'expression de ces propriétés.

5.4.2 Propriétés de sûreté (safety)

Une propriété de sûreté énonce que, sous certaines conditions, une situation ou une configuration indésirable ne peut se produire.
La vérification d'une telle propriété dans les protocoles d'interactions des SMAS nous assure de la fiabilité du protocole. La preuve d'une telle propriété consiste à raisonner sur toutes les instructions (actions) du programme, en d'autres termes à vérifier la stabilité. En général, les propriétés de sûreté sont le produit de la négation des propriétés d'atteignabilité. Comme les propriétés d'atteignabilité, il y a aussi des propriétés de sûreté qui sont considérés comme simples ou complexes. Un exemple de propriété de sûreté simple dans un SMA est qu'on doit s'assurer qu'au niveau du protocole d'interaction, il n'y a pas eu une perte de message c'est-à-dire que tout message émis sera forcément reçu. Un autre exemple de propriété de sûreté dans un SMA est qu'il y a eu une conservation de l'ordre des messages. Dans le SMA IBAUTS, lorsque nous voulons vérifier si un message arrive en retard après une série du protocole, ceci

est considéré comme une propriété de sûreté.

Cette propriété est exprimée dans CTL par la formule AGp où p est une formule du passé et AG sont les quantificateurs respectivement de chemins et temporels. Pour la logique LTL, cette propriété est exprimée avec le combinateur G.

5.4.3 Propriétés de vivacité (liveness)

Une propriété de vivacité énonce que, sous certaines conditions, une situation ou une configuration souhaitable finit par avoir lieu.
À première vue, la propriété de vivacité semble être la négation de la propriété de sûreté, ce qui n'est pas le cas. En effet, la propriété de sûreté déclare que l'utilisateur *ne peut jamais* entrer dans une mauvaise situation, tandis que la propriété de vivacité déclare que l'utilisateur *fait toujours* quelque chose de correct [Butterworth *et al.*, 1998]. Cependant, il y a une similitude entre la propriété d'atteignabilité et celle de vivacité, à une différence près : la propriété de vivacité est plus forte que celle d'atteignabilité puisque cette dernière indique juste qu'il est **possible** que quelque chose se passe, alors que la vivacité indique que quelque chose **doit** se passer. Il existe deux familles de propriétés de vivacité : la vivacité simple et la vivacité répétée nommée aussi équité que nous définirons dans le paragraphe suivant. Un exemple de propriété de vivacité dans le SMA IBAUTS est l'établissement d'un contrat après négociation ou toute demande de modification de température sera satisfaite tôt ou tard. Ce dernier exemple est représenté par $AG(requete \Rightarrow AFsatisfaite)$. C'est le combinateur F qui caractérise les propriétés de vivacité.

5.4.4 Propriétés d'équité (fairness)

Une propriété d'équité énonce que, sous certaines conditions, quelque chose aura lieu (ou n'aura pas lieu) un nombre infini de fois.
Les propriétés d'équité sont utilisées pour décrire la forme de certaines suites de choix non-déterministes [Berard *et al.*, 2001]. Il existe deux types d'équité :

Équité faible : on parle d'équité faible lorsqu'une ressource est demandée continuellement un nombre infini de fois et sans interruption.

Équité forte : on parle d'équité forte lorsqu'une ressource est demandée continuellement un nombre infini de fois, mais éventuellement avec des interruptions.

La propriété de l'équité forte entraîne la propriété d'équité faible, la réciproque n'est

pas toujours vraie [Berard *et al.*, 2001]. Cependant, au niveau de la vérification, il n'y a pas de différence de complexité entre ses deux types d'équité.
Voici un exemple de la propriété d'équité : dans un protocole contract-net du SMA IBAUTS, aucune offre de chauffage n'est envoyée un nombre infini de fois à l'agent manager (*RoomAgent*).

5.4.5 Absence de blocage (deadlock freeness)

L'absence de blocage est une propriété, énonçant que le système ne se trouve jamais dans une situation où il est impossible de progresser
Cette propriété est particulièrement importante dans le cas où elle constitue une propriété de correction pour les systèmes supposés ne jamais se terminer. L'absence de blocage est souvent classée parmi les propriétés de sûreté parce que pour un automate donné, il est en général possible de décrire explicitement les états de blocage : il suffit alors d'exprimer leur non-atteignabilité, ce qui est une propriété de sûreté. La propriété d'absence de blocage est exprimée en CTL sous la forme $AGEXtrue$.

Dans les protocoles d'interaction, la propriété d'absence de blocage nous garantit qu'étant donné un état initial, au moins une opération peut être exécutée quel que soit l'état atteint au cours d'une conversation. Comme exemple de propriété d'absence de blocage pour le SMA IBAUTS, celle ou les agents contractants ont reçu la demande de modification de température par l'agent manager (roomAgent).

5.4.6 Récapitulatif

À l'aide de cette classification de propriétés, nous pouvons identifier une liste des anti-patterns recherchés dans un SMA. Certes, le contenu de ces anti-patterns dépendra du système analysé, mais au moins nous avons cerné les différentes catégories auxquelles appartiennent reposent les anti-patterns.

Ces anti-patterns sont spécifiés en utilisant soit la logique temporelle CTL ou LTL. Cependant, l'utilisation de l'une ou l'autre nécessite une maîtrise de ces logiques. Il existe des langages de spécification d'un niveau d'abstraction plus élevé qui sont équivalents à ces logiques, plus explicites et simples à manipuler et à comprendre. Parmi ces langages, nous trouvons le langage GOAL (Voir §3.4.3) que nous avons utilisés pour la formalisation des antipatterns (Voir §6.5).

5.5 Cinquième phase : vérification avec ObjectGEODE

Comme tout outil de vérification fondé sur l'utilisation de la technique du model-checking, ObjectGEODE permet de construire un graphe global du système à vérifier (un treillis des événements observables). Pour construire ce graphe, nous commençons par modéliser chacune des composantes du système. L'automate ou le graphe d'états global est formé par tous les états globaux du système et les transitions entre états. Il est obtenu à partir des automates des composantes en faisant coopérer ces différents automates. La construction de ce graphe global dépend de l'outil utilisé pour implanter la technique du model-checking. En effet, le graphe peut être construit totalement ou partiellement en se basant sur la coopération. Cette coopération peut s'effectuer de différentes manières : les méthodes sans synchronisation, les méthodes avec synchronisation et le produit synchronisé [Berard *et al.*, 2001]. ObjectGEODE utilise le produit synchronisé qui permet de représenter un système de processus en tenant compte des interactions qui ont lieu dans le système.

Le produit synchronisé de n automates $A_i =< Q_i, E_i, T_i, q_{0,i}, l_i >,\ i = 1, \ldots, n$ est simplement l'automate $A =< Q, E, T, q_0, l >$ où :

$$
\begin{cases}
\mathrm{Q}{=}\mathrm{Q}_1 * \ldots * Q_n \textit{ avec } Q_i \textit{ est un ensemble fini d'états} \\
\text{et} \\
\mathrm{E}{=}\prod_{1 \leq i \leq n}(E_i \cup \{-\}) \textit{ avec } E_i \textit{ est l'ensemble fini des étiquettes des transitions} \\
\text{et} \\
\mathrm{T}{=} \begin{cases}
((\mathrm{q}_1, \ldots, q_n), (e_1, \ldots, e_n), (q_1^{'}, \ldots, q_n^{'}) | (e_1, \ldots, e_n) \in E_i \cup \{-\}) \\
\text{et } \forall i, e_i =' -' \textit{ et } q_i^{'} = q_i, \textit{ ou } e_i \neq' -' \textit{ et } (q_i, e_i, q_i^{'}) \in T_i \quad \textit{avec } T_i \subseteq Q_i * E_i * Q_i \\
\textit{est l'ensemble de transitions}
\end{cases} \\
\text{et} \\
\mathrm{q}_0 = (q_{0,1}, \ldots, q_{0,n}) \textit{ avec } q_{0,i} \textit{ est l'état initial de l'automate} \\
\text{et} \\
\mathrm{l}((\mathrm{q}_1, \ldots, q_n)) = \cup_{1 \leq i \leq n} l_i(q_i) \textit{ avec } l_i \textit{ est l'application qui associe à tout état de } Q_i \\
\textit{l'ensemble fini des propriétés élémentaires vérifiées dans cet état}
\end{cases}
$$

Le produit synchronisé peut se faire selon deux modes :

- La synchronisation par envoi/réception de messages. Ce type est un cas particulier du produit synchronisé ou seules les transitions autorisées sont celles où toute émission est accompagnée de la réception correspondante. Pour différencier les événements d'envoi et de réception, ces derniers sont étiquetés par $!m$ (envoi du message m) et $?m$ (réception du message m).

- La synchronisation par variables partagées. Les interactions entre les différents processus peuvent ne pas se faire par envoi et réception de messages mais plutôt par partage de variables. L'automate global de tout le système est construit à partir des automates et des différentes valeurs des variables partagées.

En faisant une simulation exhaustive[3], l'outil ObjectGEODE génère le graphe d'états complet du système. Ce graphe représente tous les entrelacements possibles des événements de la trace collectée. Concernant le principe de vérification proposée par ObjectGEODE, ce dernier consiste en une analyse globale du système, en vérifiant qu'une propriété est satisfaite par tous les états atteignables ou bien par toutes les exécutions de ce système. Pour effectuer cette vérification du comportement du système, ObjectGEODE élabore un produit synchronisé, basé sur la synchronisation par envoi et réception de messages, des propriétés exprimées en GOAL et des spécifications du système (les différents modèles SDL) [SA, 1999]. Par la suite, le simulateur d'ObjectGEODE produit les résultats de cette analyse qui sont soit des résultats statistiques comme le nombre d'erreurs et de succès retrouvés, soit des résultats qualitatifs comme les scénarios qui mènent aux erreurs.

5.6 Conclusion

Ce chapitre a décrit l'approche de vérification que nous suggérons pour l'analyse des protocoles d'interaction dans les SMAs. Cette dernière a nécessité un mécanisme d'observation dans le but de générer la trace d'exécution du système et ainsi analyser le comportement social des agents à des fins de détection des erreurs protocolaires et applicatives. Cette phase est considérée comme une étape importante de notre approche de vérification. La pertinence du résultat de la vérification dépend du processus d'observation. Ce chapitre a couvert, entre autres, les problèmes pouvant se déclencher lors de l'observation d'une exécution distribuée ou parallèle. Garantir la notion d'ordre est l'un des aspects les plus fondamentaux pour assurer une bonne instrumentation de la trace. En outre, nous avons présenté un panorama de techniques et d'architectures des outils d'observation pour la mise au point des systèmes distribués asynchrones. Nous avons construit un outil d'observation pour les protocoles d'interactions dans les SMAs, en nous inspirant de ceux des systèmes distribués. En particulier, nous avons retenu une architecture centralisée pour l'agent observateur et nous avons choisi de placer les sondes logicielles d'observation en instrumentant les agents. Cette collecte de trace a été

[3]La simulation exhaustive des modèles SDL permet de construire le graphe d'états du système qui couvrira toutes les exécutions possibles du système en générant les entrelacements des exécutions respectives de chacune des composantes.

dirigée par les événements et non pas par les messages. Ce chapitre a aussi fait le survol de la phase de modélisation et décrit une classification des propriétés comportementales aidant ainsi les experts à situer les catégories des anti-patterns recherchés.

Le chapitre suivant est une mise en oeuvre de notre approche pour l'analyse d'un système multiagent industriel.

Chapitre 6

Mise en application de l'approche : cas du SMA du projet IBAUTS

Nous avons présenté au chapitre précédent la méthodologie de la vérification et de l'observation d'une exécution multiagents. Nous avons notament étudié les problèmes posés lors de l'observation et la journalisation de l'historique. Nous présentons ci-après l'application de notre approche de vérification afin d'analyser le protocole d'interaction du SMA du projet IBAUTS[1] et par conséquent analyser le comportement externe et global du SMA.

Nous commençons par présenter le SMA IBAUTS dont nous avons vérifié le comportement. Puis, nous décrivons l'architecture du module d'observation en termes de fonctionnalités. Ensuite, nous présentons la phase de modélisation de la trace récoltée lors de l'étape d'observation. Subséquemment, nous identifions les anti-patterns d'interaction et nous donnons quelques exemples d'anti-patterns dont nous avons vérifié l'existence dans les modèles extraits de la trace d'exécution. Et enfin, nous analysons les résultats obtenus lors de la phase de vérification. Mais avant tout, nous allons décrire l'environnement de travail choisi pour la conception et le développement de cette approche.

[1] Dorénavant et par abus de langage, nous le noterons SMA IBAUTS

6.1 Environnement de travail

6.1.1 Langage de conception et de programmation

La conception du module d'observation de notre approche a été réalisée à l'aide du langage de modélisation UML. Ce dernier est un langage semi-formel, ayant plusieurs notations, parmi lesquelles, la notation graphique qui est la plus utilisée puisqu'elle permet de visualiser une solution objet. La puissance de ce langage réside dans le fait qu'il offre différents diagrammes d'analyse et de conception permettant ainsi de couvrir les différents aspects, que ce soit statique ou/et dynamique des systèmes à concevoir. De plus, son indépendance par rapport aux langages d'implantation, au domaine d'application et aux processus font de lui un langage universel.

Le langage JAVA a été utilisé pour le développement de ce module. Ce choix a été motivé par la simplicité, la portabilité, le multi-threading et la notion d'objet présent dans ce langage. De plus, JAVA offre une variété d'API qui facilite énormément la tâche au programmeur et qui cible différents aspects de la programmation : interface graphique, réseau, sécurité, accès aux bases de données, etc.

Nous avons aussi utilisé des librairies de la plateforme GUEST que nous détaillons dans la section suivante.

6.1.2 Plateforme Guest

Il existe plusieurs plates-formes multiagents qui offrent des environnements différents et dont les fonctionnalités de base ne sont pas les mêmes. Cette diversité offre donc aux agents différentes manières de s'exécuter et de fonctionner. En revanche, la dépendance des agents à la plate-forme où ils ont été créés est une contrainte limitative pour le développement des applications hétérogènes dans lesquels des agents aglets [URL2,] et des agents Jade [URL3,] peuvent cohabiter et coopérer [Magnin *et al.*, 2002]. La plate-forme GUEST résout ce problème. En effet, GUEST offre la possibilité d'exécuter des agents sur des plates-formes hétérogènes. De plus, elle permet la communication entre agents et la migration des agents d'une plate-forme à une autre. Pour cela, elle fournit une couche d'interface avec les autres plates-formes pour qu'elles soient en mesure d'accueillir des agents qui ne leur étaient pas destinés initialement. Elle permet ainsi l'uniformisation de l'utilisation et l'interaction avec les autres plates-formes (Voir FIG.6.1). Ces nouvelles fonctionnalités sont mises en oeuvre grâce à un mécanisme de *plug-ins* qui permet d'étendre de façon dynamique et transparente les capacités et le comportement des agents.

GUEST facilite le développement, le test et le déploiement des SMAs. De plus, elle fournit des outils de développement d'agents, de test, de déboguage et de communication entre les agents GUEST.

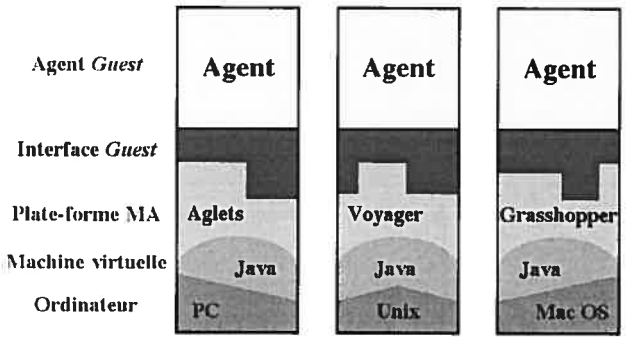


FIG. 6.1: La plateforme Guest

6.1.3 L'environnement ObjectGEODE

Pour l'opération de la vérification proprement dite, nous avons utilisé l'environnement ObjectGEODE. Ce dernier résulte de la combinaison de l'environnement GEODE pour SDL et de l'environnement LOV pour OMT[2], tous les deux réalisés et commercialisés au début par la société Verilog. Présentement, c'est la société Telelogic [URL4,] qui se charge de la commercialisation d'ObjectGEODE.

Cet environnement utilise en parallèle les notations SDL et MSC[3], pour l'édition, la simulation, la génération du code et le débogage, ainsi que le standard OMT. La notation MSC permet de décrire les scénarios représentant les interactions entre les différents composants du système et le standard OMT permet de décrire les données d'une manière orientée objet.

La puissance de l'environnement d'ObjectGEODE repose sur son aptitude à gérer la complexité en modélisant et en décrivant, grâce au langage SDL, la structure, le comportement et les données à partir d'une approche formelle garantissant la fonctionnalité du système. De plus, il peut être utilisé soit au début du processus de développement d'un système distribué pour analyser la spécification du système, soit à la fin du processus pour vérifier si le système déjà développé est exempt d'erreurs.

Dans ce qui suit nous allons décrire les différents composants fournis par l'environnement ObjectGEODE. Nous avons utilisé l'éditeur et le simulateur d'ObjectGEODE

[2]Object Modeling Technique
[3]Message Sequence Chart

pour vérifier l'absence des anti-patterns dans la trace d'exécution du système.

L'éditeur ObjetGEODE-SDL : cet éditeur fournit des moyens pour créer, modifier, et visualiser les diagrammes d'une description d'ObjectGEODE : architecture, communication, machine d'état et MSC.

Le simulateur ObjectGEODE : le simulateur fournit des techniques pour la correction, la vérification et la validation des systèmes. Il permet de détecter, à partir de la spécification d'un système, les erreurs de modélisation afin de montrer que le système est conforme à ses exigences. Durant la simulation, certains problèmes peuvent apparaître comme des situations de blocage (deadlocks[4], livelocks[5]), des pertes de signaux, un dépassement de la capacité des files d'attente, etc.

Le simulateur offre trois modes de simulation : interactive, aléatoire ou exhaustive. Le choix de l'utilisation de l'un des modes dépend de l'étape du développement du système et du but de la simulation. Par exemple, pour analyser une erreur, on utilise le mode interactif pour re-exécuter le scénario et connaître l'origine et la raison de la présence de cette erreur. Par contre, si nous voulons détecter rapidement les erreurs alors la simulation aléatoire est utilisée. Enfin, le mode exhaustif est utilisé pour détecter toutes les erreurs possibles. Chaque scénario d'erreur qui est détecté est enregistré pour l'analyse interactive. À la fin de cette simulation, nous pouvons être sûr que tous les scénarios possibles d'erreurs ont été détectés et que le comportement désiré du système a été réalisé.

Test Composer & TTCN Test Suite Publisher : le *test composer* a pour but de générer des jeux de tests à partir des modèles SDL et MSC. *TTCN* [6] *Test Suite Publisher* est une extension de *Test Composer* qui permet de traduire les jeux de test en format de TTCN pour la suite TAU TTCN de Telelogic. Cette suite est un environnement standard développé par Telelogic pour le test de conformité des systèmes de communication.

Le générateur d'application d'ObjectGEODE : ce générateur génère le code source du système final. Ce code est écrit avec le langage C et il est obtenu en se basant sur la spécification SDL. Aussi, il génère les fichiers *makefiles* pour automatiser la construction de ce processus. Grâce à la bibliothèque d'ObjectGEODE, le code généré est mappé aux systèmes d'exploitation tels que CHORUS, NOYAU, OSE, OSEK, pSOS+, VRTXsa®, VxWORKS, WIN2 et aux protocoles de communication tels que TCP/IP [URL4,].

[4] La situation de deadlocks engendre un arrêt immédiat du système

[5] La situation de livelocks est un peu difficile à détecter puisque le système continu à fonctionner mais de façon dégradé

[6] Tree and Tabular Combined Notation (ISO standard)

6.2 Description du SMA IBAUTS

Le prototype du contrôleur IBAUTS est un système multiagents, construit en utilisant la plate-forme GUEST. Ce prototype permet de contrôler et d'optimiser le fonctionnement des systèmes de chauffage, de ventilation et de climatisation des édifices tout en gérant de façon intelligente un ensemble d'effecteurs (dispositifs de chauffage). Ainsi, il permet de minimiser la consommation d'énergie, l'usure de ces dispositifs et de maximiser l'efficacité des systèmes de chauffage.

La première version du prototype permet de contrôler la température d'une pièce donnée et gère trois dispositifs de chauffage. Son architecture est composée de trois modules.

- **Module des dispositifs de chauffage**. Il est composé de trois agents (*Damper Agent, Reheat coil Agent et Perimeter Heating Agent*) qui représentent chacun un dispositif de chauffage.
- **Module de la pièce**. Il est représenté par un seul agent (*Room Agent*). Le *Room Agent* joue le rôle de l'agent manager au sens du protocole contract-net (Voir §2.5.1).
- **Module de jonction**. Il établit le lien entre le simulateur MATLAB et le contrôleur multiagent formé par les agents du premier et deuxième niveau. Il est représenté par un seul agent (*Bridge Agent*). Le simulateur MATLAB a été utilisé pour simuler les différentes conditions extérieures (la température extérieure, le nombre d'occupants de la pièce, la saison, la date, l'heure, la réaction thermique des murs et de l'intérieur de la pièce). Ces conditions sont nécessaires pour la mise en oeuvre du contrôleur multiagents.

La coordination entre les agents des différents niveaux se fait par le biais de protocoles de communication. Ces protocoles couvrent l'étape de la création et de l'initialisation des agents ainsi que la coordination entre eux. Lors de la phase de création, les agents doivent être en mesure de se découvrir automatiquement grâce au principe de *Plug & Play*. Chacun des agents diffuse un message à tous les autres agents existants. Ces derniers mémorisent la présence d'un nouvel agent et lui renvoie un message asynchrone pour l'informer de leur existence et par conséquent de leur identité. Les agents doivent garder la liste de ses connaissances. Suite à la phase de création, le protocole itéré de contract-net sera utilisé pour la coordination entre les agents (*Damper Agent, Reheat coil Agent, Perimeter Heating Agent, Room Agent*) et plus spécifiquement pour l'allocation des tâches par l'agent responsable de la pièce. Le début d'un round de protocole contract-net est déclenché par un envoi d'un message synchrone par le *Bridge*

Agent. Ce message contient toutes les informations nécessaires pour l'élaboration d'une demande de changement de température. Par conséquent, le *Bridge Agent* joue le rôle d'initiateur du protocole d'interaction entre l'agent responsable de la pièce et ceux responsables des dispositifs. De plus, il permet de faire le lien entre le simulateur MATLAB et les autres agents qui forment le contrôleur.

Dans ce qui suit, nous présenterons une brève explication du fonctionnement des dispositifs de chauffage.

6.2.1 Description des dispositifs du chauffage

Dispositif de ventilation (Damper Device) : l'agent *damper* a pour tâche de contrôler l'ouverture et la fermeture du dispositif *damper* pour lui permettre de laisser passer l'air extérieur pour satisfaire l'exigence minimale de ventilation. Lorsque le dispositif *damper* est fermé, il n'est pas totalement fermé. En effet, pour permettre le renouvellement de l'oxygène, l'ouverture minimale du dispositif doit être toujours égale à 10% de son ouverture maximale.

Radiateur de réchauffage (Reheat Coil Device) : l'agent responsable du *reheat coil* permet de réchauffer l'air provenant de l'extérieur, lorsque cela est nécessaire, et de l'acheminer dans la pièce. Le dispositif de *reheat coil* permet de chauffer l'air traversant le *damper*.

Système de chauffage périmétrique (Perimeter Heating Device) : ce dispositif mural permet de chauffer l'air de la pièce dans le cas où le *reheat coil* n'arrive pas à procurer suffisamment d'air chaud pour maintenir la température souhaitée.

6.2.2 Règles de chauffage

Pour garantir le bon fonctionnement du contrôleur multiagent, ce dernier est soumis à quelques règles et heuristiques qui sont :

- Si le *damper* est ouvert (plus que son minimum) alors on ne doit pas utiliser le *perimeter heating* pour chauffer la pièce. En effet, lorsque le *damper* est ouvert, alors il y a une entrée d'air froid pour la pièce. En parallèle, on utilise le *perimeter heating* pour chauffer la pièce. Par conséquent, il y aura un gaspillage d'énergie. Ainsi, si on chauffe la pièce avec le *perimeter heating* alors on doit fermer le *damper*.

- Si le *damper* est ouvert à son maximum alors on ne doit pas utiliser le *reheat coil* pour chauffer la pièce. Ceci est dû à la limite de la capacité du *reheat coil* à chauffer l'air provenant du damper. Par conséquent, si le damper est ouvert à son maximum alors la quantité d'air à chauffer par le *reheat coil* sera importante et donc il n'arrivera pas à maintenir la température souhaitée.
- Si la température de la pièce est assez proche de celle souhaitée, ce n'est plus nécessaire de modifier la température.

6.2.3 Protocole d'interaction du SMA IBAUTS

Pour aboutir au maintien de la température de la pièce, les agents doivent se coordonner. Cette coordination se fait en utilisant un protocole d'interaction qui est une variante du protocole contract-net (se référer au §2.5.1). Ce protocole est utilisé chaque fois que la pièce a besoin d'une participation des dispositifs de chauffage pour maintenir ou atteindre la température voulue.

Le contrôleur multiagent que nous avons vérifié est un système qui est supposé fonctionner en permanence. Nous avons donc utilisé le protocole contract-net de façon itérative, répétée et infinie. En effet, le *Bridge agent* envoie, à chaque laps de temps fixé, un message afin d'initier, à nouveau, une série de protocole.

Une série typique de ce protocole est décrit comme suit : dès que l'agent responsable de la pièce reçoit un message du *Bridge agent* avec tous les paramètres nécessaires pour effectuer une demande de modification de température alors l'agent responsable de la pièce diffuse le message de modification de température à tous les agents responsables des dispositifs (Voir FIG. 6.2). Le Room Agent joue le rôle de l'agent manager au sens du protocole contract-net. Les agents responsables des dispositifs élaborent leurs propositions et les soumettent au *Room Agent*. Ensuite, ce dernier sélectionne la meilleure proposition et envoie un message de contrat si c'est nécessaire. Enfin, l'agent sélectionné exécute la commande précisée dans le message du contrat.

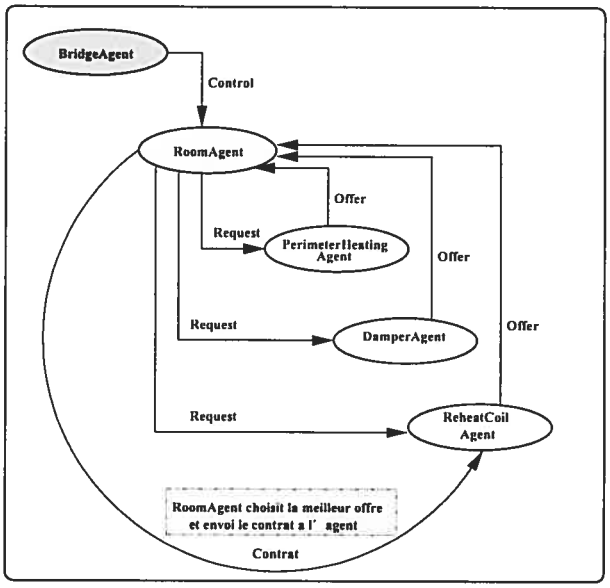


FIG. 6.2: Architecture du système Ibauts

6.3 Génération et modélisation de la trace

Nous avons présenté dans le chapitre précédent l'intérêt et l'importance d'un outil d'observation de l'exécution pour notre approche de vérification des protocoles d'interaction dans les SMAS. Nous étudions ci-après comment cette observation a été réalisée dans le cadre du projet IBAUTS. Nous expliquons les étapes de l'observation, de la génération et de la modélisation de la trace. Ainsi, nous présentons comment nous avons pu ordonné les événements associés aux interactions.

6.3.1 Observation des interactions dans le SMA IBAUTS

Le mécanisme d'observation que nous avons retenu, a une architecture centralisée. Ce choix est basé sur le principe que l'observation est une forme d'intrusion qui peut changer le comportement de l'application observée. Par conséquent, nous sommes amenés à limiter cette intrusion. Nous avons donc tenté de minimiser la perturbation du comportement des agents en évitant qu'ils aient leurs propres modules d'observation. De cette façon, les agents seront consacrés uniquement à l'exécution des tâches pour lesquelles ils ont été créés. En outre, la centralisation de l'observation permet de reconstruire une vue globale du comportement du système. Dans ce sens, nous avons choisi d'observer les événements relatifs aux créations des agents, aux messages échangés entre eux et aux paramètres de ces messages. Nous ne nous sommes pas intéressés à la sauve-

garde des événements internes ou locaux, puisque les événements liés aux envois et réceptions de messages contiennent toutes les informations relatives à l'état des agents. Les événements observés sont les suivants :

- les événements liés à la création des agents. Nous avons enregistré les informations relatives lors de la création des agents et leurs états.
- les événements externes sont les événements qui illustrent une communication entre plusieurs agents. Ces événements sont décomposés en deux catégories : les événement liés à une action d'émission de message et les événements liés à une action de réception de message.

Nous avons développé un module d'observation composé de deux sous-modules qui permettent d'intercepter les événements lors de l'envoi ou de la réception des messages synchrones et asynchrones. Ces modules ont été élaborés grâce aux mécanismes des plugins de la plate-forme Guest (Voir FIG. 6.3) qui peuvent être ajoutés ou retirés de l'agent. Dès qu'un agent charge un plugin, ce dernier s'abonne à tout message envoyé ou reçu pour chaque agent observé du SMA, prend une copie du message et l'achemine à l'agent central (agent observateur). Cet agent central se charge alors de faire l'appariement des différents événements au moyen d'un référentiel de temps logique.

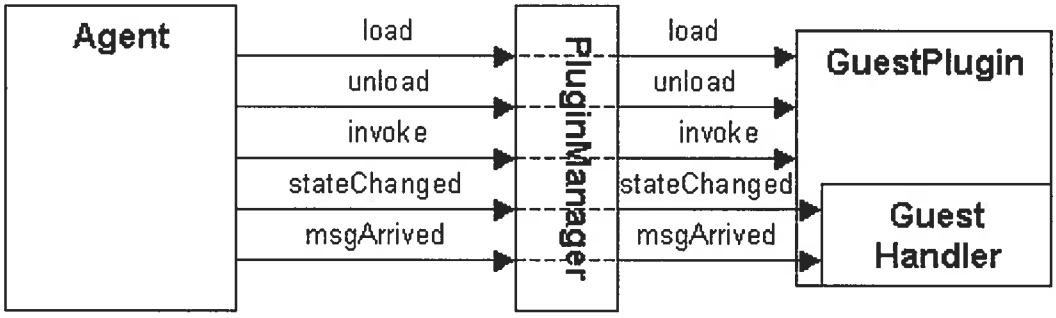
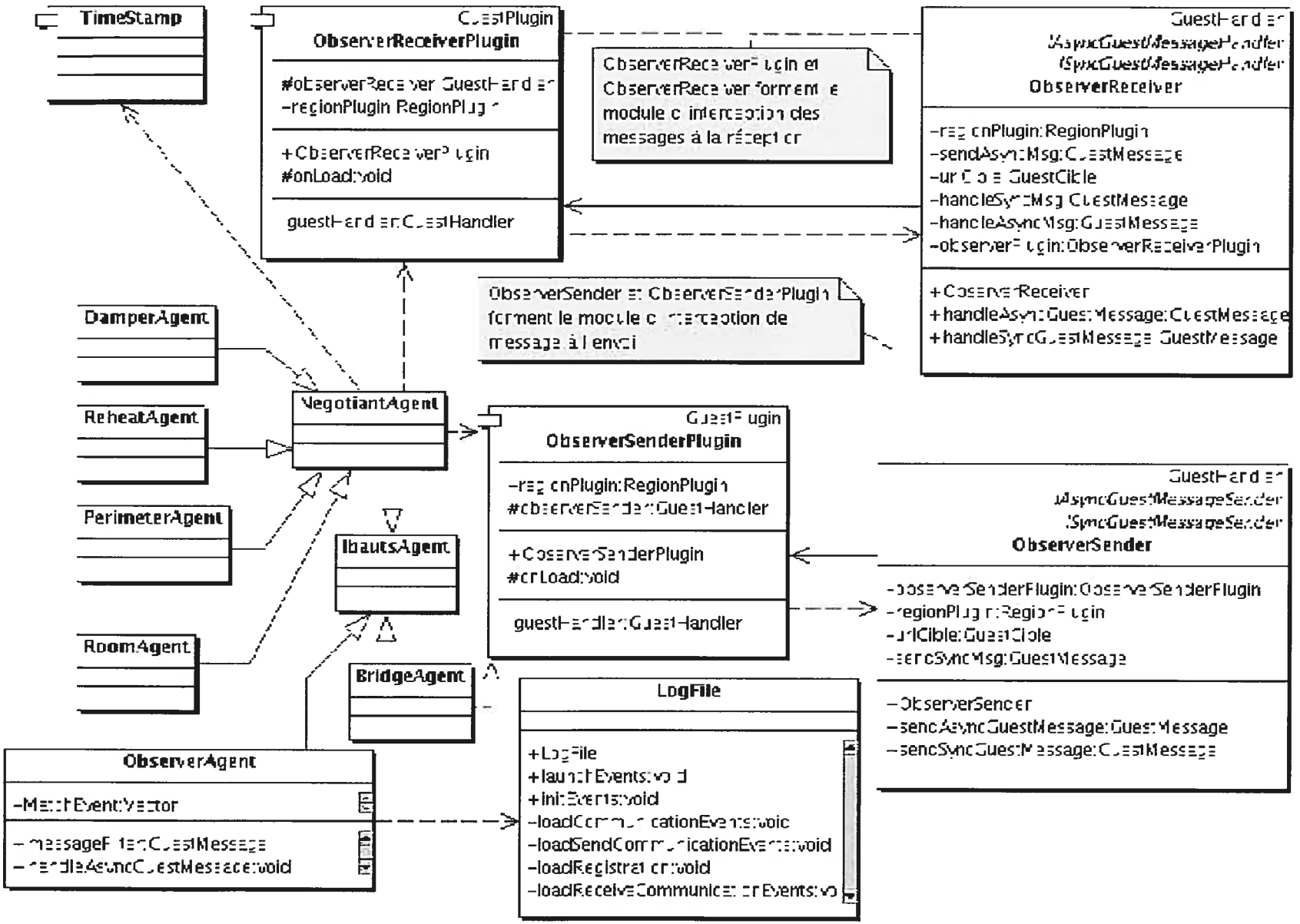


FIG. 6.3: Modèle d'extension par plugins

Le diagramme de classe du module d'observation se compose de 7 classes (Voir FIG. 6.4). Ce diagramme a été conçu avec UML et a été traduit sous forme de classes, écrites en JAVA, décrivant ainsi le comportement, les fonctionnalités et les données du module d'observation et d'interception des communications.


TimeStamp
GuestPlugin
ObserverReceiverPlugin
#observerReceiver GuestHandler
-regionPlugin RegionPlugin
+ObserverReceiverPlugin
#onLoad:void
guestHandler:GuestHandler
ObserverReceiverPlugin et ObserverReceiverFormant le module d'interception des messages à la réception
GuestHandler
IAsyncGuestMessageHandler
ISyncGuestMessageHandler
ObserverReceiver
-regionPlugin:RegionPlugin
-sendAsyncMsg:GuestMessage
-uriCible:GuestCible
-handleSyncMsg:GuestMessage
-handleAsyncMsg:GuestMessage
-observerPlugin:ObserverReceiverPlugin
+ObserverReceiver
+handleAsyncGuestMessage:GuestMessage
+handleSyncGuestMessage:GuestMessage
ObserverSenderPlugin et ObserverSenderPlugin forment le module d'interception de message à l'envoi
DamperAgent
ReheatAgent
PerimeterAgent
RoomAgent
NegotiantAgent
IbautsAgent
BridgeAgent
GuestPlugin
ObserverSenderPlugin
-regionPlugin:RegionPlugin
#observerSender:GuestHandler
+ObserverSenderPlugin
#onLoad:void
guestHandler:GuestHandler
GuestHandler
IAsyncGuestMessageSender
ISyncGuestMessageSender
ObserverSender
-observerSenderPlugin:ObserverSenderPlugin
-regionPlugin:RegionPlugin
-uriCible:GuestCible
-sendSyncMsg:GuestMessage
-ObserverSender
-sendAsyncGuestMessage:GuestMessage
-sendSyncGuestMessage:GuestMessage
LogFile
+LogFile
+launchEvents:void
+initEvents:void
-loadCommunicationEvents:void
-loadSendCommunicationEvents:void
-loadRegistration:void
-loadReceiveCommunicationEvents:void
ObserverAgent
-MesEvent:Vector
-messageFilter:GuestMessage
-handleAsyncGuestMessage:void

Sous-Module d'interception des événements lors de l'envoi du message

Ce module décrit l'étape d'interception des messages lors de leur envoi. Il est composé de deux classes *ObserverSender* et *ObserverSenderPlugin* qui héritent respectivement des classes *GuestHandler* et *GuestPlugin*. C'est la classe *ObserverSender* qui permet d'intercepter tous les événements intrinsèques aux messages (asynchrone ou synchrone), échangés entre les agents abonnés à ce sous-module, grâce aux fonctions *sendAsyncGuestMessage* et *sendSyncGuestMessage* et de les acheminer vers l'agent observateur. Ces deux fonctions sont appelées avant l'envoi d'un message asynchrone ou synchrone de l'agent auquel ce plugin est attaché.

L'abonnement des agents à ce sous-module d'observation se fait, lors de la création d'un agent, en instantiant la classe *ObserverSenderPlugin* qui fait appel à la classe *ObserverSender*.

Sous-Module d'interception des événements lors de la réception du message

Ce module décrit l'étape d'interception des messages lors de leurs réception. Il est composé de deux classes *ObserverReceiver* et *ObserverReceiverPlugin* qui héritent respectivement des classes *GuestHandler* et *GuestPlugin*. C'est la classe *ObserverReceiver* qui permet d'intercepter tous les événements intrinsèques aux messages (asynchrone ou synchrone), échangés entre les agents abonnés à ce sous-module, grâce aux fonctions *handleAsyncGuestMessage* et *handleSyncGuestMessage* et de les acheminer vers l'agent observateur. Ces deux fonctions sont appelées quand un message asynchrone ou synchrone arrive à l'agent auquel ce plugin est attaché.

L'abonnement des agents à ce sous-module d'observation se fait, lors de la création d'un agent, en instantiant la classe *ObserverReceiverPlugin* qui fait appel à la classe *ObserverReceiver*.

6.3.2 Ordre des événements et journalisation

Comme l'observation, la journalisation est aussi élaborée de façon centralisée. Pour avoir une vue globale et cohérente de la collaboration entre les différents agents, nous avons utilisé une technique d'estampillage pour préserver l'ordre temporel dans la trace collectée. La technique que nous avons choisi pour ordonner les événements est celle de l'horloge logique de Lamport (Voir §5.2.2). Ce sont les agents observés qui calculent les estampilles des messages qu'ils envoient et qu'ils reçoivent. L'agent Observateur est chargé d'ordonner les événements envoyés par les différents agents et de

faire l'appariement (le mot en anglais est *matching*) entre les événements d'envoi et de réception de message. Lors de l'appariement, l'agent observateur filtre les événements grâce à la fonction *messageFilter*. Si un événement d'envoi a sa correspondance (événement de réception du message déjà envoyé) alors l'agent observateur stocke ces informations dans un fichier XML. Sinon, si l'événement d'envoi n'a pas de correspondance alors l'agent observateur stocke ces informations dans un tableau en attendant la réception de l'événement manquant. Dans le cas où un événement n'aurait pas de correspondant, celui-ci sera stocké à la fin de l'étape d'observation dans le même fichier et l'ordre de causalité entre les événements sera assuré par l'estampille. La classe *LogFile* se charge de sauvegarder les événements.

Nous avons observé et enregistré le comportement du SMA IBAUTS durant une journée. Les traces collectées représentent donc une simulation d'une journée complète d'exécution, soit 2380 événements (émission ou réception de message).

6.3.3 Modélisation de la trace

Le modèle du SMA IBAUTS est présenté par les modèles des différents agents qui le composent. Le modèle d'un agent est obtenu en projetant la trace rassemblée dans l'ensemble de types d'événements : envoyés, reçus et les événements locaux. Nous avons donc eu autant de modèles SDL que d'agents observés qui sont le *BridgeAgent, RoomAgent, ReheatAgent, PerimeterHeatingAgent* et *DamperAgent.*

La figure 6.5 représente une partie du comportement du *RoomAgent.* Comme le montre la figure, chaque message est identifié par son nom et la liste de ses paramètres, présentés comme des variables. Aussi, la figure révèle, que le modèle SDL fait une distinction entre les messages reçus et envoyés[7] par l'agent. Nous avons utilisé, comme paramètre de message, la variable *tranid* qui nous a permis de définir l'appartenance des messages au bloc de contrôle. Ce bloc est défini par une séquence de messages échangés par les agents entre deux messages de *control* émis par le *BridgeAgent.*

Plusieurs difficultés ont été rencontrées lors de la modélisation du système. Nous avons choisi au début d'instrumenter le comportement d'une exécution durant une semaine. Cependant, le fait d'extraire des modèles d'une trace assez volumineuse (30 Mb, environ 20000 événements) a été ralenti par l'impossibilité de l'outil à accomplir cette tâche dans une courte durée à cause du problème de l'explosion combinatoire. Ces problèmes ne sont pas liés aux composantes matérielles de l'ordinateur, mais plutôt à l'outil de vérification. Nous nous sommes alors contenté de vérifier une exécution d'une

[7]Les rectangles femelles représentent les messages reçus et les rectangles mâles illustrent les messages envoyés par l'agent

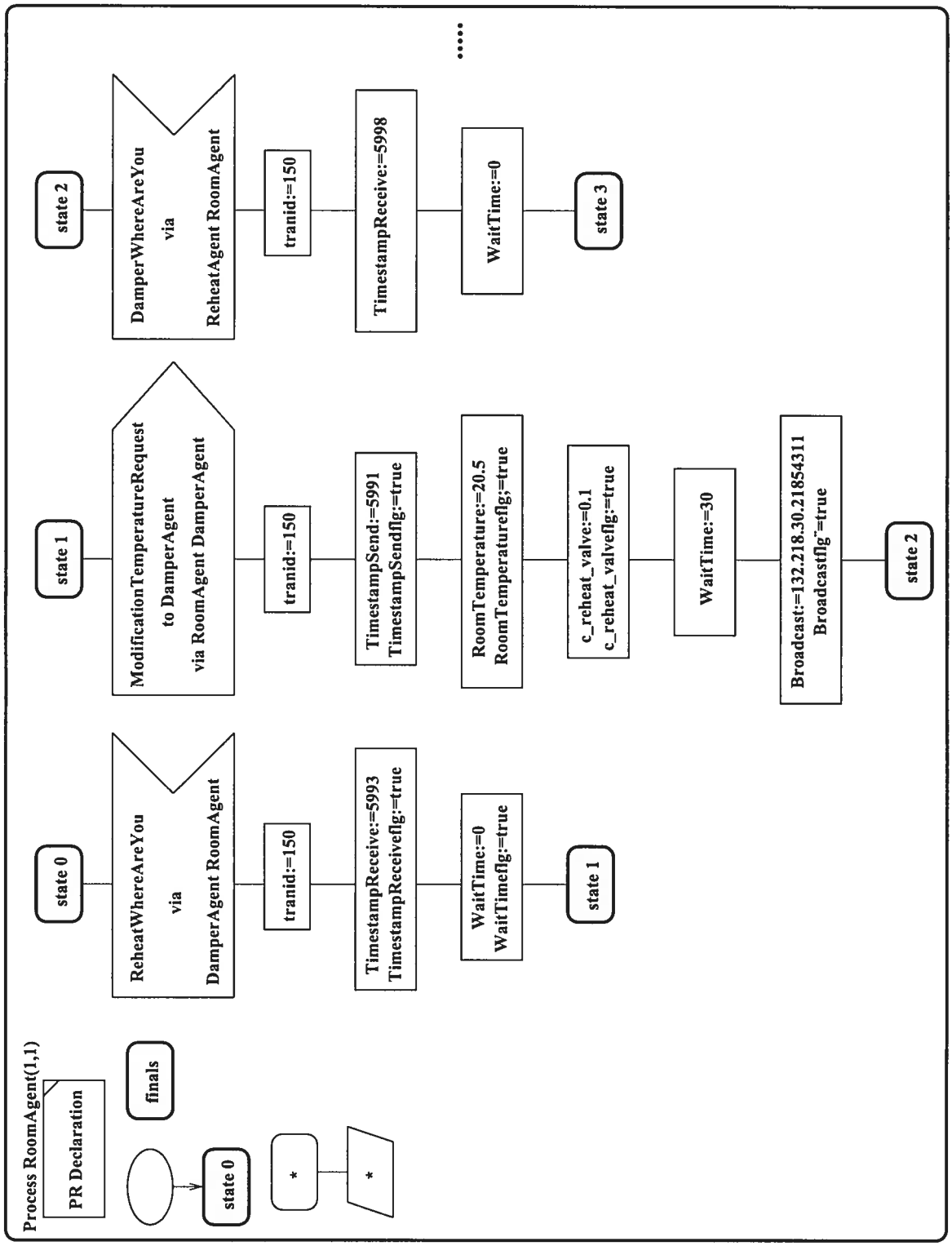


FIG. 6.5: Le modèle SDL du *RoomAgent*

seule journée. Cependant, si l'outil a pu générer des modèles à partir d'une trace assez volumineuse, il sera possible d'exprimer des anti-patterns portant sur des données de plusieurs jours.

6.4 Identification des anti-patterns d'interaction

Dès qu'on veut vérifier un système, nous sommes amenés à déterminer une liste de propriétés les plus pertinentes que le système doit satisfaire. Nous avons choisi de définir des propriétés qui reflètent non pas des caractéristiques de bon fonctionnement qu'un protocole d'interaction doit remplir, mais plutôt une liste de problèmes et de dysfonctionnements qui ne doivent pas figurer au niveau du protocole. Ce que nous avons appelé anti-patterns d'interaction. Ce choix a été motivé par la difficulté de garantir la conformité du système à sa spécification. Les anti-patterns sont des instances indésirables du protocole d'interaction impliquant plusieurs agents. Leur présence dans le protocole montre qu'il y a soit une erreur soit un mauvais fonctionnement du protocole et leur absence permet de garantir que cette exécution du système est correcte par rapport à cet anti-pattern.

La classification de ces anti-patterns peut être considérée selon plusieurs points de vue. D'une part, nous pouvons faire la distinction selon la nature de ces anti-patterns : qualitatifs ou quantitatifs. Ce que nous appelons par anti-patterns quantitatifs sont les propriétés qui portent sur le nombre d'occurrence d'un problème ou sur les valeurs de certains paramètres de message ; et par qualitatifs sont les propriétés qui portent sur la cohérence des messages. D'autre part, nous pouvons différencier ces anti-patterns selon leurs catégories : protocolaires ou applicatifs. Ce que nous nommons par anti-patterns protocolaires sont les propriétés qui portent sur la structure du protocole d'interaction comme les délais de réponse, la perte des messages, la mort des agents, etc ; et par anti-patterns applicatifs sont les propriétés qui portent sur l'application comme la cohérence du contenu des offres effectuées par l'agent contracteur, la validité des réponses offertes par les agents contractés, etc. Nous avons identifié quatorze anti-patterns : six sont protocolaires, huit applicatifs que nous allons détailler dans les sections qui suivent.

6.4.1 Absence de réponse dans un tour de coordination

Cette propriété nous permet de vérifier que tous les agents répondent bien à chaque offre. Elle couvre ainsi deux problèmes : le problème de réception de message (par exemple, l'agent contractant n'a pas connaissance d'un message qui lui a été envoyé par l'agent manager) et le problème de non réaction des agents (par exemple, l'agent

contractant a reçu un message de l'agent manager mais il n'élabore pas une réponse).

Nous avons défini alors quatre anti-patterns couvrant tous ces aspects et permettant ainsi de vérifier l'exactitude de la correction de l'algorithme de proposition d'offre des agents responsables des dispositifs de chauffage. Tous ces anti-patterns sont considérés comme des anti-patterns protocolaires et sont détaillés dans le tableau suivant. La troisième colonne du tableau fait référence à la classification des propriétés selon la technique du model-checking (Voir §5.4)

Numéro	**Anti-patterns d'interactions**	**Classes de propriétés**
1	*RoomAgent* envoi une demande de modification de température aux agents responsables des dispositifs de chauffage. Cependant, parmi ces derniers, certains ne reçoivent pas ce message.	Blocage
2	*RoomAgent* envoi une demande de modification de température aux agents responsables des dispositifs de chauffage. Cependant, aucun de ces derniers ne reçoit ce message.	Blocage
3	Quelques un des agents responsables des dispositifs de chauffage ne soumettent pas leurs offres suite à la réception d'une demande de modification de température de *roomAgent.*	Propriété de sûreté
4	Aucun des agents responsables des dispositifs de chauffage ne soumet son offre suite à la réception d'une demande de modification de température de *roomAgent.*	Propriété de sûreté

TAB. 6.1: Les anti-patterns portant sur l'absence de réponse

À première vue, le deuxième anti-pattern est une généralisation du premier anti-pattern ainsi que le quatrième pour le troisième anti-pattern. Nous avons opté à différencier ces anti-patterns pour pouvoir commenter avec exactitude les résultats de vérification. En effet, l'impact de la présence du premier sur la suite d'exécution du système est différent de celui du deuxième anti-pattern. Tandis que le premier, ne bloque pas l'exécution du système mais participe à la dégradation de sa performance, le deuxième engendre un blocage momentané et révèle une panoplie plus large de causes de ce problème (mort des agents, problème au niveau du réseau, problème au niveau du contrôle des dispositifs physiques, etc).

6.4.2 Réponse trop tardive d'un agent

Cette propriété permet de garantir que tous les messages échangés entre les agents arrivent à temps. Ceci implique que les agents ont assez de temps pour l'élaboration et la soumission de leurs réponse. L'absence de cette propriété de la trace d'exécution permet d'affirmer que les délais mis pour l'exécution d'une tâche par un agent sont cohérents avec la durée d'un tour de coordination.

Nous avons défini alors un seul anti-pattern qui permet de s'assurer que les évolutions futures de l'algorithme utilisé par chaque agent, respecteront les contraintes de temps réel de l'application. Cet anti-pattern est de type protocolaire et il est détaillé dans le tableau suivant.

Numéro	**Anti-patterns d'interactions**	**Classes de propriétés**
5	Un message arrive en retard, c'est-à-dire après un tour du protocole.	Propriété de sûreté

TAB. 6.2: Les anti-patterns portant sur les délais de transmission

6.4.3 Réponse non cohérente par les agents contractants

Cette propriété permet de s'assurer de la fiabilité du mécanisme de raisonnement des agents contractants ainsi que de l'agent manager. L'absence de cette propriété de la trace d'exécution permet de garantir que le système développé implémente correctement les règles du domaine.

Nous avons défini alors quatre anti-patterns qui permettent de vérifier l'exactitude des prises de décisions, des états et des comportements des agents. Tous ces anti-patterns sont considérés comme anti-patterns applicatifs. Le tableau 6.3 les décrivent.

Numéro	Anti-patterns d'interactions	Classes de propriétés
6	*RoomAgent* envoi une demande de modification de température malgré que cette dernière est proche de celle souhaitée. Cependant, les agents responsables des dispositifs de chauffage répondent favorablement à cette demande.	Propriété de sûreté
7	Dans un tour de protocole et suite à une demande de modification de température par le *RoomAgent*, *DamperAgent* propose de se fermer. Cependant, *RoomAgent* choisit *PerimeterAgent* pour modifier la température. Ceci engendre un problème puisque c'est moins coûteux de fermer la source de la provenance de l'air froid que de chauffer la pièce.	Propriété de sûreté
8	Dans le cas où le *DamperAgent* est ouvert à son maximun et le *ReheatAgent* est sélectionnée pour chauffer la pièce alors ceci cause un problème.	Propriété de sûreté
9	Tous les agents responsables des dispositifs de chauffage soumettent leurs propositions à l'agent responsable de la pièce. Cependant, le coût énergétique est supérieure au seuil acceptable (coût calculé grâce à un système d'apprentissage)	propriété d'atteignabilité

TAB. 6.3: Les anti-patterns portant sur la non cohérence d'une réponse

Lors de la formalisation du sixième anti-pattern, nous l'avons décomposé en deux anti-patterns différents dépendament de la nature de la commande de modification température (refroidissement, chauffage) et des valeurs des paramètres du message (saison, heure, etc).

6.4.4 Réponse validée d'un agent, mais non suivie d'effet

Ce type de propriété permet de vérifier qu'une tâche a été belle et bien exécutée par les dispositifs physiques de chauffage. Cette propriété nous garantit que le système physique de chauffage, contrôlé par le SMA IBAUTS, se trouve toujours dans un état cohérent. La vérification de telle propriété nous est rendue possible par le fait que les messages de contrat, émis par l'agent contracteur, contiennent des informations issues des capteurs du système physique. Nous pouvons donc exprimer des propriétés sur ces informations [Elleuch Benayed *et al.*, 2003].

Nous avons défini alors deux anti-patterns qui permettent de vérifier l'exactitude des prises de décisions, des états et des comportements des agents. Le dixième anti-pattern est protocolaire, par contre le onzième est applicatif. C'est un anti-pattern qui permet de détecter des pannes physiques au niveau des dispositifs de chauffage ou la présence des éléments externes. Le tableau 6.4 décrit ces différents anti-patterns.

Numéro	**Anti-patterns d'interactions**	**Classes de propriétés**
10	Un des agents responsables des dispositifs de chauffage reçoit le contrat mais ne l'exécute pas.	Propriété de sûreté
11	Un agent est utilisé pour modifier la température de la pièce. Cependant, la température ne change pas (fenêtre ouverte ou problèmes au niveau du matériel des dispositifs de chauffage).	Propriété d'atteignabilité

TAB. 6.4: Les anti-patterns portant sur l'absence d'exécution d'une tâche ou de son résultat

6.4.5 Usure ou sous-utilisation des dispositifs de chauffages

Ce type de propriété permet de vérifier soit la surcapacité ou la sous-capacité d'un dispositif de chauffage. Certes, cette propriété ne vérifie pas la cohérence et l'évolution de l'algorithme du SMA IBAUTS. Mais plutôt, elle essaie d'aider les spécialistes du domaine de bâtiments à avoir une analyse statistique de l'utilisation de ces dispositifs.

Nous avons défini alors deux anti-patterns qui permettent de vérifier cette propriété. Ce sont deux anti-patterns applicatifs. Le tableau 6.5 décrit ces différents anti-patterns.

Numéro	Anti-patterns d'interactions	Classes de propriétés
12	Surcapacité d'un appareil : sur 1 an, un appareil n'a jamais fonctionné à fond (le seuil qui détermine si un appareil a fonctionné à fond pourra être calculé grâce à un mécanisme d'apprentissage qui ne fait pas l'objet de ce mémoire).	Propriété d'atteignabilité
13	Sous-capacité d'un appareil : sur 1 an, un appareil a toujours fonctionné à fond.	Propriété d'atteignabilité

TAB. 6.5: Les anti-patterns portant sur la sous-utilisation des dispositifs de chauffage

6.5 Formalisation des anti-patterns en GOAL

Nous avons pu formaliser juste sept anti-patterns et les tester sur l'exécution du système. Certes, nous avons voulu tester tous les anti-patterns. Mais comme nous n'avons pas un accès quotidien à l'ObjectGEODE, nous nous sommes contentés de vérifier juste sept anti-patterns que nous avons modélisés dans le formalisme de GOAL. Parmi ces sept, nous avons pu vérifier six anti-patterns (trois applicatifs et trois protocolaires) sur les *logs* de l'application. Ces anti-patterns sont exprimés en *observer* dans le langage GOAL [Hallal and Petrenko, 2001] comme un automate d'états fini.

6.5.1 Formalisation des anti-patterns protocolaires

Les anti-patterns protocolaires que nous avons vérifiés sur le fichier *log* sont le premier, le deuxième et le cinquième anti-pattern. En réalité, pour le premier et le deuxième anti-pattern, nous n'avons pas eu à les formaliser avec le langage GOAL puisque ce sont des propriétés qui décrivent une situation de blocage. De tels propriétés sont détectables dès la phase de la modélisation de la trace d'exécution. En effet, lors du parsing du fichier par l'outil TRAYSIS, ce dernier vérifie si l'événement d'envoi est jumelé par un événement de réception et que ces deux événements ont le même identifiant du message.

Cependant, le cinquième anti-pattern a été modélisé en GOAL. La modélisation de cet anti-pattern a été faisable puisque le protocole que nous avons vérifié est cyclique. De plus, nous avons ajouté un autre paramètre (*coorelation_id*) aux messages échangés entre les agents afin de définir une série de protocole. Le début d'une série de protocole est initié par le message de *control* envoyé par le *BridgeAgent.*

Les trois figures qui suivent représentent chacune un automate fini. Les losanges représentent les conditions «si...alors...sinon», les rectangles sont des zones d'affectation de variables, les polygones décrivent une boucle «tant que» et les états sont schématisés par les rectangles à bord arrondi.

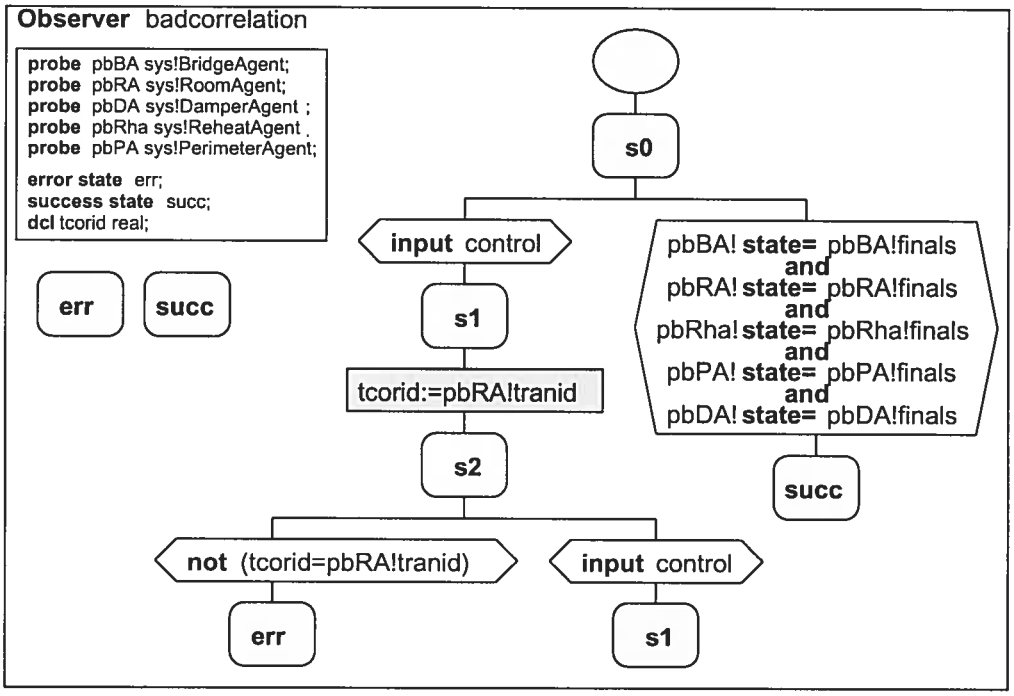


FIG. 6.6: Modélisation du cinquième anti-pattern en Goal

La figure 6.6 décrit la modélisation de cet anti-pattern et se lit de la façon suivante : au début, on définit une étiquette *s0*. Tant que le *RoomAgent* reçoit le message «*control*» du *BridgeAgent*, il définit une autre étiquette *s1* et il sauvegarde la valeur du paramètre «*coorelation_id*» de ce message. Une autre étiquette *s2* sera alors défini. En parcourant la trace, deux chemins d'exécution sont possibles : le premier est lorsque le message qui suit est un message de «*control* », dans ce cas nous rebouclons au niveau de l'étiquette *s1*. Le deuxième chemin sera parcouru lorsque le message est différent de «*control*», on compare alors la valeur du paramètre «*coorelation_id*» de ce message avec celle du message «*control*». Si les deux valeurs sont différentes alors nous pouvons conclure que l'agent a répondu de façon tardive. Sinon, nous parcourons tous les événements de la trace jusqu'à la fin. Si toute la trace a été parcourue sans passer par un état d'erreur, alors nous pouvons conclure que la trace est exempte de cet anti-pattern.

Durant la vérification de notre trace d'exécution avec ObjectGEODE, cet anti-pattern sera déclenché par l'occurrence du message «*control*». Si l'état d'erreur est rencontré

alors le processus de vérification s'arrête.

6.5.2 Formalisation des anti-patterns applicatifs

Nous avons formalisé le septième anti-pattern et les deux variantes du sixième anti-pattern en GOAL. Nous avons ainsi vérifié leurs absences du fichier *log*.

La variante du sixième anti-pattern, que nous allons décrire ci-après, porte sur la non cohérence du raisonnement des agents. En effet, nous avons voulu détecter, par cet anti-pattern, l'absence des interventions indisérables effectuées par les dispositifs de chauffage lorsque la pièce est dans un état satisfaisant. Pour modéliser cet anti-pattern en GOAL, il fallait voir qu'à partir du moment où la température de la pièce est confortable et proche mais non nécessairement égale à celle souhaitée et pour tous les chemins d'exécution qui suivent, il n'existe pas de chemin où il y a un changement des commandes des dispositifs de chauffages.

La figure 6.7 décrit la modélisation de cet anti-pattern et se lit de la façon suivante: au début, on définit une étiquette *s0*. Deux chemins d'exécutions peuvent être rencontrés. Le premier décrit le fait que lorsque le *RoomAgent* reçoit le message «*control*» du *BridgeAgent*, il teste les valeurs des paramètres «*time, season*» de ce message. Si on est aux heures de travail et on est en hiver alors on passe à l'état *s1*, sinon on arrête.
À partir de l'étiquette *s2*, deux autres chemins d'exécution sont possibles. Le premier s'explique comme suit : tant qu'on a un message de contrat, on sauvegarde alors la commande effectuée par l'agent choisi et on incrémente la variable *c* qu'on définit, puis on passe à l'état *s0*. Le deuxième chemin d'exécution est traduit par : tant que le *RoomAgent* reçoit une offre de l'un des agents *reheat coil, perimeter heating et damper*, on teste si on a déjà passé par le premier chemin d'exécution et si la source de l'offre est la même que le destinataire du contrat. Si la réponse est affirmative alors on compare la commande de l'agent contractant du round précédent avec celle du round présent. Si l'agent propose de modifier la température alors on passe à un état d'erreur sinon on reboucle à l'état s0.
Le deuxième chemin du niveau de l'étiquète *s0* décrit le fait que toute la trace a été parcourue. Dans ce cas, si nous ne sommes pas passés par un état d'erreur, alors nous pouvons conclure que la trace est exempte de cet anti-pattern.

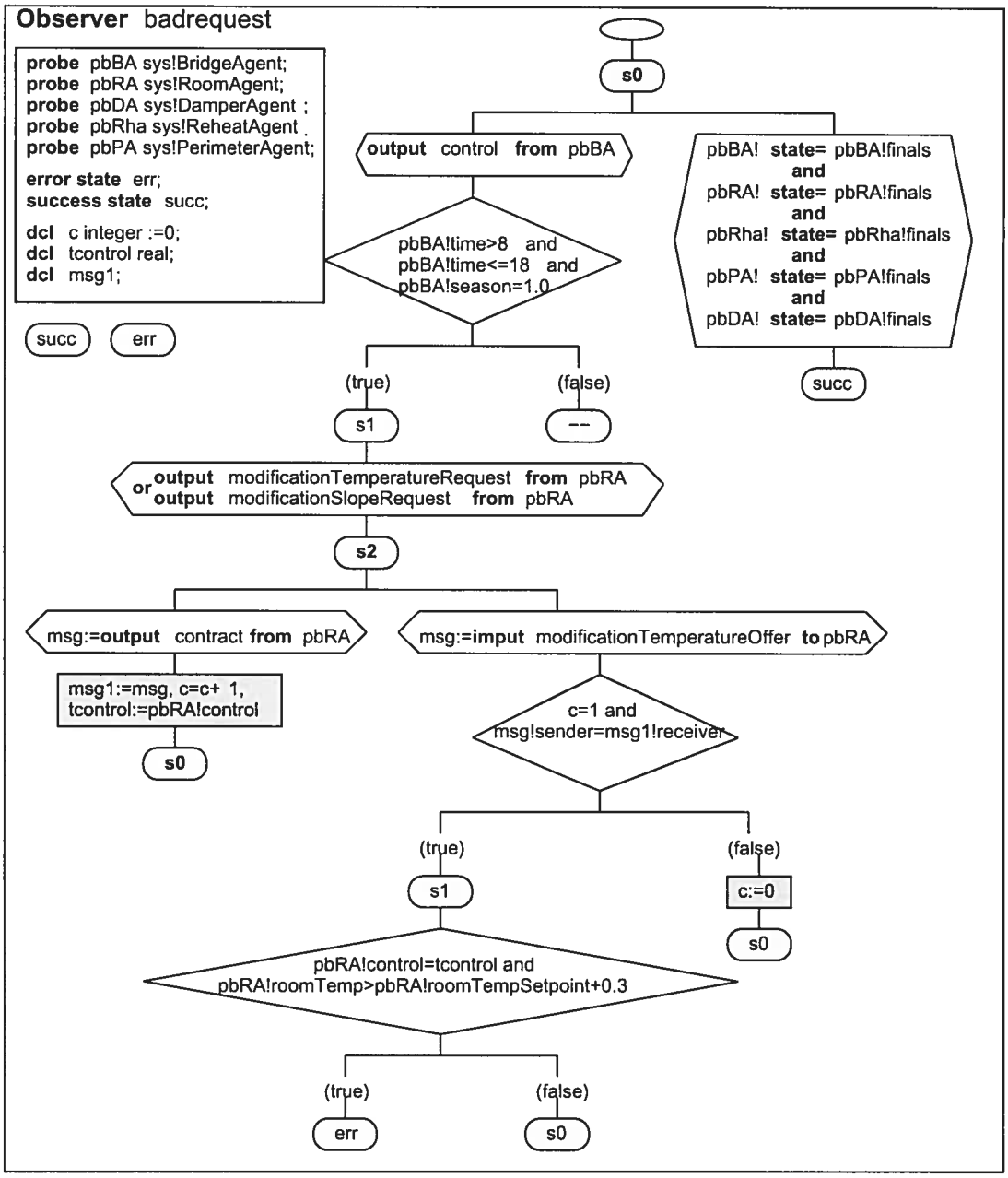


FIG. 6.7: Modélisation d'une variante du sixième anti-pattern en Goal

La figure 6.8 décrit la modélisation du septième anti-pattern qui teste la cohérence des réponses des agents contractants.

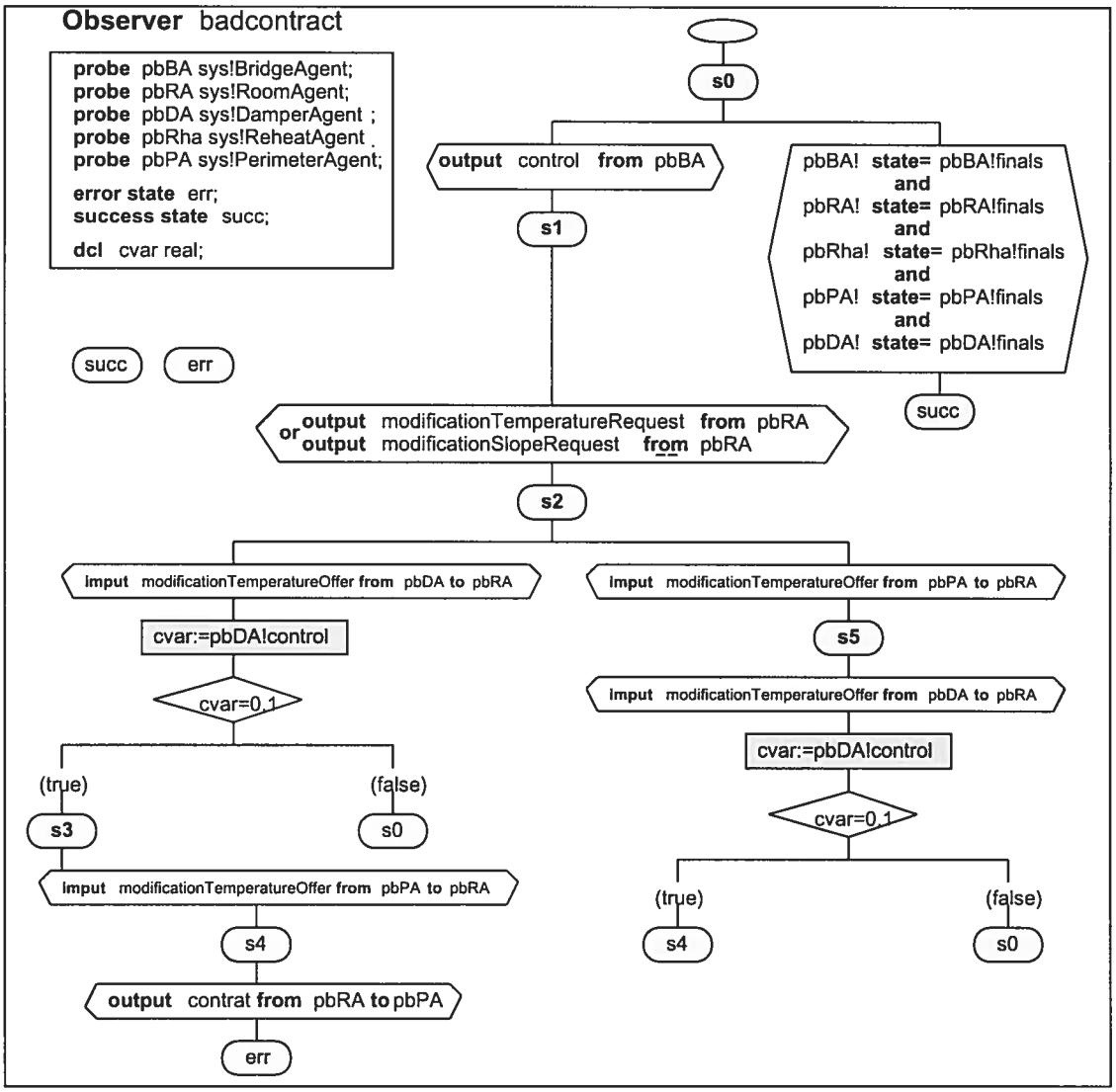


FIG. 6.8: Modélisation du septième anti-pattern en Goal

La vérification de ces deux anti-patterns ont été rendus possible grâce au fait que les messages de contrat, émis par l'agent contracteur, contiennent des informations issues des capteurs du système physique.

Il est à noter que la phase de formalisation des anti-patterns est aussi importante et pertinente que la phase de modélisation du système pour la verification des SMAs. Cette étape nécessite une expertise du langage GOAL et une connaissance du domaine.

6.6 Résultats de la vérification

Suite à la modélisation du système et à la formalisation des anti-patterns, nous avons utilisé l'environnement ObjectGEODE, plus spécifiquement le model-checker et le simulateur de ce dernier, pour vérifier l'absence des anti-patterns dans le modèle du système. Nous avons utilisé des traces représentant chacune une journée complète d'exécution du système, soit environ trois mille événements (émission ou réception de messages) par trace. Nous avons vérifié l'absence de six anti-patterns dont quatre ont été modélisés en langage GOAL. Parmi ces six anti-patterns, une seule propriété indésirable (le cinquième anti-pattern) a été détectée dans la trace, où elle apparaît de façon occasionnelle.

La figure 6.9 est le résultat de la vérification du cinquième anti-pattern illustrant si un message arrive en retard dans un round de protocole. La figure montre que dès que le model-checker a détecté la présence de cet anti-pattern, le processus de vérification s'est arrêté automatiquement. En effet, l'outil spécifie le nombre d'état et de transitions non couverts, ainsi que le nombre de succès ou d'erreurs.

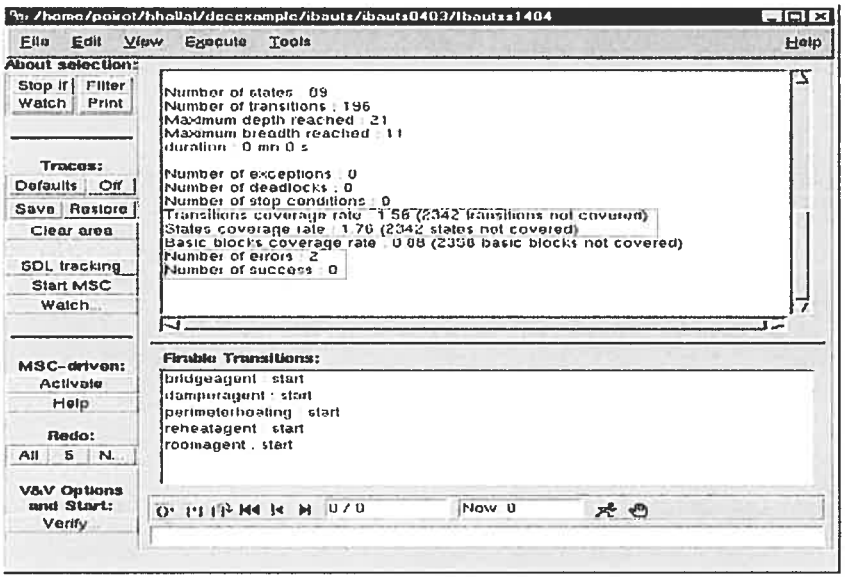


FIG. 6.9: Résultat de la vérification du cinquième anti-pattern

L'analyse du résultat de la vérification de cet anti-pattern nous a été rendue possible grâce au contre exemple, généré par l'outil suite à la détection de ce dernier. Ce contre exemple est un scénario de la séquence des interactions qui ont conduit à ce mauvais comportement et qui est illustré par le diagramme de MSC. Pour comprendre le problème, nous avons schématisé deux scénarios afin de les comparer. Le premier scénario (Voir FIG. 6.10) illustre un comportement normal du protocole contract-net.

Tandis que le deuxième scénario (Voir FIG. 6.11) décrit le comportement erroné qui a été détecté.

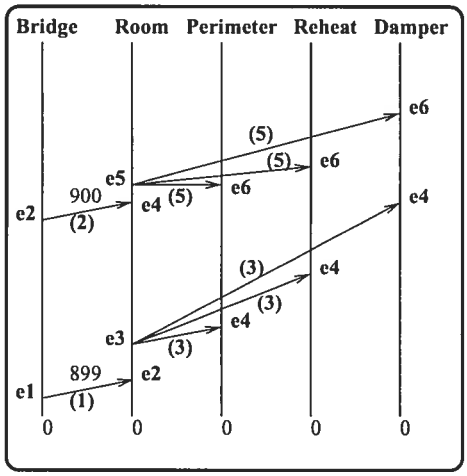


FIG. 6.10: Scénario normal du protocole contract-net

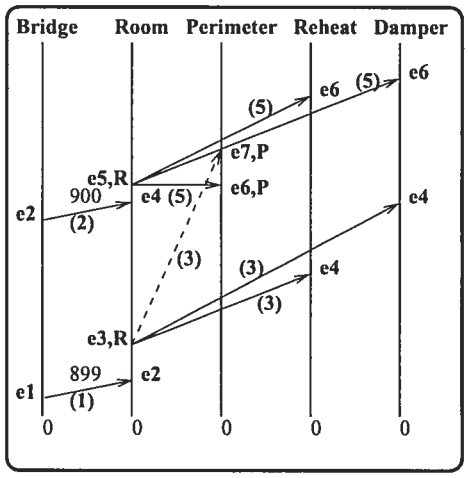


FIG. 6.11: Scénario décrivant le comportement érroné

Dans le premier scénario, nous remarquons que les messages sont ordonnés. Les événements inhérents aux messages envoyés et reçus dans la série 899 arrive avant ceux de la série 900. Par contre, dans le deuxième scénario, tous les événements intrinsèques aux messages envoyés de la série 899 arrive avant celle de la série 900. Cependant, le problème réside lors de la réception des messages. Malgré que l'événement $e_{3,R}$ précède l'envoi de l'événement $e_{5,R}$, le *PerimeterAgent* reçoit $e_{5,R}$ bien avant $e_{3,R}$. La cause la plus probable de ce problème est engendrée par les différents délais d'exécution. La difficulté de déterminer avec exactitude la cause de cet anti-pattern est occasionné par la variation du comportement du système d'une exécution à une autre. Effectivement, cet anti-pattern ne s'est présenté que dans une seule exécution du système.

Concernant les autres anti-patterns, le résultat de vérification a montré leurs absences de la trace d'exécution. Cette absence a prouvé l'exactitude, la consistance et la cohérence du protocole d'interaction. En outre, ces résultats nous ont confirmé que les mécanismes de raisonnement des agents, leurs stratégies pour prendre des décisions et leurs connaissances sont conformes et compatibles avec les règles et contraintes du domaine. Ceci est considéré comme un apport positif pour les changements futurs de l'application. En effet, en s'assurant que le système est exempte de ces anti-patterns dans la version antérieure, nous pouvons réaliser, de façon automatique, les tests de non-régression lors de l'évolution du prototype du SMA IBAUTS.

6.7 Conclusion

Dans ce chapitre, nous avons mis en oeuvre notre approche et avons pu l'appliquer pour vérifier un système multiagents destiné à une utilisation industrielle. Du même coup, nous avons mis en pratique les bases et les notions théoriques sur lesquelles nous nous sommes fondés pour la proposition de cette approche. Certes nous n'avons pas eu l'occasion d'implanter tous les anti-patterns que nous avons identifiés pour cet SMA. Cependant, nous estimons que nous avons exploré et passé par toutes les étapes de notre approche de vérification et que nous avons montré sa faisabilité.

Le système multiagents IBAUTS nous a permis non seulement d'appliquer cette approche de vérification mais aussi d'envisager d'autres champs d'application à cette approche. Actuellement, d'autres agents de recherche de l'équipe GLIC du CRIM songe à y étendre cette dernière pour la vérfication des applications basées sur l'utilisation des services Web.

Chapitre 7

Conclusion et Perspectives

Ce mémoire traite le problème de la vérification des protocoles d'interaction dans les SMAs. Pour ce faire, nous avons effectué une revue de littérature des travaux de recherche qui existent dans le domaine et nous avons proposé notre démarche pour spécifier, observer, analyser et vérifier les interactions entre les agents. Cette approche a nécessité la connaissance et la fouille dans plusieurs domaines allant des systèmes multiagents, du génie logiciel, de la spécification et vérification formelles jusqu'aux mécanismes de monitoring des systèmes distribués.

7.1 Conclusion

Nous avons présenté une approche de vérification des systèmes multiagents, basée sur l'analyse des traces d'exécution de ces systèmes en utilisant la technique du model-checking (destinée à l'origine à la vérification des systèmes distribués). Elle a comme objectif d'analyser les exécutions d'un SMA à travers les interactions, de vérifier la cohérence de ces protocoles et de prouver que telle exécution de ce système est exempte d'erreurs.

Cette approche a nécessité, en premier lieu, l'élaboration et le développement d'une architecture de collecte de traces, devant répondre aux problèmes : de l'absence d'un référentiel de temps absolu, de la préservation de l'ordre des événements pour certifier que le comportement observé est similaire au comportement réel, de la variation des comportements d'une exécution à une autre et de la dégradation des performances des systèmes. Cette étape est importante pour le processus de vérification de part la problématique qu'elle soulève mais aussi de part son influence sur ce processus. Nous avons résolu, dans la mesure du possible, certains problèmes. D'abord, nous avons

implanté un mécanisme d'instrumentation centralisé pour éviter de perturber le comportement des agents, de modifier les délais d'exécution des différentes tâches et donc éviter de dégrader la performance du système. En outre, le problème de l'absence d'un référentiel de temps absolu a été réglé en utilisant une notion de temps logique permettant la synchronisation des horloges basée sur la technique d'estampillage de Lamport. Cependant, nous avons fait l'hypothèse que nous ne tenons pas compte de la variation des comportements d'une exécution à une autre. Cette instrumentation s'est basée sur l'occurrence des événements d'envoi et de réception d'un message et a été élaboré en ajoutant des sondes logicielles aux agents.

Cette observation et instrumentation des systèmes multiagents nous a permis d'analyser le comportement des agents et de détecter les erreurs protocolaires et applicatives. Pour cette analyse de la trace d'exécution, nous avons utilisé un outil formel de vérification de modèles des systèmes distribués. Il permet de produire les modèles d'exécution d'un système d'après sa trace et de vérifier l'existence ou l'absence de certaines propriétés du graphe global construit à partir de ces modèles. Dans cette phase, nous avons été amenés à choisir un langage de modélisation pour la trace d'exécution du système, d'identifier les différentes propriétés recherchées au niveau du protocole, de trouver un langage de formalisation pour ces propriétés et enfin de trouver un vérificateur de modèle acceptant tous ces langages et permettant de faire une vérification exhaustive. Pour cela, nous avons abordé une panoplie de langages et de méthodes formels de modélisation qui sont adaptés à la description des aspects tels que le parallélisme, la concurrence, la synchronisation et le non-déterminisme que présente un SMA. Puis, nous avons donné une classification des propriétés comportementales qui peuvent être recherchées dans de tels systèmes afin de faciliter l'identification des anti-patterns.

Nous avons modélisé dans l'outil ObjectGEODE choisi, les propriétés recherchées dans l'application industrielle IBAUTS que nous voulons vérifier. Ces anti-patterns ont été formalisés grâce au langage GOAL et la trace d'exécution a été modélisée avec le langage SDL en autant de modèles que d'agents différents. Après la phase de modélisation du système et de formalisation des propriétés, le simulateur d'ObjectGEODE a été utilisé pour construire le treillis global du système en faisant un produit synchronisé des différents modèles représentant les agents. Puis, le model-checker d'ObjectGEODE nous a permis de vérifier l'existence de ces propriétés dans ce treillis.

Les résultats de la phase de vérification sont assez intéressants. Sur les six anti-patterns dont nous avons cherché l'inexistence dans la trace d'exécution, cinq n'apparaissent effectivement pas dans la trace d'exécution. La non-présence de ces cinq anti-patterns nous a permis de montrer que le système répondait correctement à sa

spécification et que le mécanisme de raisonnement des agents ainsi que les stratégies adoptées par ces derniers sont conformes à ce qu'elles doivent être. De plus, ces résultats nous ont permis de réaliser, de façon automatique, les tests de non-régression lors de l'évolution du prototype du SMA IBAUTS. Ces tests consistent à vérifier et à s'assurer qu'une nouvelle version de l'application ne brise pas le comportement et le fonctionnement – *déjà correct* – de la version précédente de l'application. Cependant, nous avons détecté la présence d'un anti-pattern dans la trace d'exécution. Même si sa présence n'a été constaté qu'une seule fois et dans une seule exécution du système, nous nous sommes penchés sur ce problème pour déterminer la cause dont le but de le corriger.

Pour conclure, l'approche que nous avons élaboré pour la vérification des systèmes multiagents englobe plusieurs disciplines. Nous avons ainsi élaboré et donné une preuve de concept du mécanisme de collecte de trace synchronisée avec une notion de temps logique. En outre, nous avons modélisé le système à travers sa trace d'exécution, identifié et formalisé les anti-patterns que le développeur d'une application multiagents souhaite vérifier et effectué une vérification totalement automatique grâce à la technique de *model-checking* et plus spécifiquement en utilisant le *model-checker* de l'environnement ObjectGEODE.

7.2 Perspectives

Ce mémoire a permis d'ouvrir de nouveaux horizons de recherche. Parmi ces axes, trois nous ont paru intéressants. Le premier axe de recherche est d'étendre cette approche pour l'instrumentation des systèmes réactifs. Au lieu d'observer les communications entre agents, on pourrait observer et instrumenter les actions élaborées par ces derniers. La seule modification qui doit être apportée à notre approche est celle du module d'observation. En effet, c'est un travail de généralisation de l'outil d'observation. Au lieu d'avoir des plug-ins qui s'abonnent aux agents et se déclenchent à l'envoi ou à la réception d'un message, on doit alors élaborer un nouveau mécanisme qui permet de capter les occurrences des actions. Pour la synchronisation des activités, on peut utiliser les horloges de Lamport.
Comme variante de cet axe, on peut aussi étendre cette approche pour la vérification du comportement interne des agents cognitifs. Cette généralisation est plus compliquée que la première et nécessite un accès au code source du système et au mécanisme de raisonnement des agents.

La deuxième voie de recherche est celle de l'automatisation du processus d'identification des anti-patterns. Cela sera avantageux d'arriver à implanter un modèle de transformation automatique permettant ainsi de produire les propriétés intéressantes

à partir d'une spécification formelle du protocole de communication, avec le minimum d'intervention humaine. Ces propriétés seront ensuite utilisées dans le but de vérifier la trace d'exécution. Cette perspective entre dans le cadre du projet GUEST en permettant ainsi d'enrichir cette plate-forme par les techniques et les outils nécessaires à automatiser ce processus et par conséquent, à tenir compte du processus de vérification au début du cycle de développement d'un SMA et non pas à sa fin.

La troisième voie est celle de l'apprentissage : il serait intéressant d'enrichir le mécanisme de raisonnement des agents par un processus d'apprentissage. En effet, les scénarios obtenus lors de la phase de vérification peuvent être utilisés dans le futur, comme moyen de prévention et de détection d'un cas d'erreur. De cette manière, l'agent saura d'avance que tels paramètres ou telles situations peuvent provoquer un dysfonctionnement au niveau du protocole d'interaction. Ainsi, il pourrait changer sa stratégie et améliorer son comportement.

Bibliographie

[Benerecetti and Cimatti, 2001] M. BENERECETTI and A. CIMATTI. Symbolic model checking for multi-agent systems. Technical Report, Istituto Trentino di Cultura, Décembre 2001. In Proceedings of the ICLP'01 Workshop on Computational Logic in Multi-Agent Systems (CLIMA-01), citeseer.nj.nec.com/benerecetti01symbolic.html.

[Berard *et al.*, 2001] B. BERARD, M. BIDOIT, A. FINKEL, F. LAROUSSINIE, A. PETIT, L. PETRUCCI, P. SCHNOEBELEN, and P. MCKENZIE. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer Verlag, Août 2001. ISBN: 3540415238.

[Butterworth *et al.*, 1998] R. J. BUTTERWORTH, A. BLANDFORD, and D. J. DUKE. The role of formal proof in modelling interactive behaviour. In P. MARKOPOULOS and P. JOHNSON, editors, *Design, Specification and Verification of Interactive Systems*, pages 87–101, Wien, 1998. Springer-Verlag.

[Davis and Smith, 1983] R. DAVIS and R. G. SMITH. Negotiation as a metaphor for distributed problem solving. In A. H. BOND and L. GASSER, editors, *Readings in Distributed Artificial Intelligence*, volume 20, pages 63–109. Kaufmann, San Mateo, CA, January 1983.

[Demazeau, 1995] Y. DEMAZEAU. From interactions to collective behaviour in agent-based systems. In *V. Lesser, editor, AAAI-Press, 1st International Conference on Multi-Agents Systems*, San Francisco, June 1995.

[Doran *et al.*, 1997] J. E. DORAN, S. FRANKLIN, N. R. JENNINGS, and T. J. NORMAN. On cooperation in multi-agent systems. *The Knowledge Engineering Review*, 12(3):309–314, 1997. http://www.ecs.soton.ac.uk/ñrj/pubs.html visité le 22 janvier 2003.

[Dury, 2000] A. DURY. *Modélisation des interactions dans les systèmes multi-agents.* PhD thesis, Université Henri Poincaré- Nancy 1, Décembre 2000.

[Elleuch Benayed *et al.*, 2003] N. ELLEUCH BENAYED, A. DURY, H. HALLAL, and MAGNIN. Vérification des systèmes multiagents ouverts. a été soumis pour la conférence JFIADSMA2003, avril 2003.

[Ferber, 1995] J. FERBER. *Les systèmes multi-agents : Vers une intelligence collective.* InterEditions, 1995.

[Fidge, 1988] J. FIDGE. Timestamps in message passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, Février 1988.

[Fisher, 1995] M. FISHER. Towards a semantics for concurrent metatem. In M. FISHER and R. OWENS, editors, *Executable Modal and Temporal Logics: Proc. of the IJCAI-93 Workshop*, pages 86–102. Springer, Berlin, Heidelberg, 1995.

[Garcia-Molina *et al.*, 1984] H. GARCIA-MOLINA, F. GERMANO, and W. H. KOHLER. Debugging a distributed computing system. In *IEEE Transactions on Software Engineering*, volume 10, pages 210–219, March 1984.

[Gasser *et al.*, 1987] L. GASSER, C. BRAGANZA, and N. HERMAN. Implementing distributed ai systems using mace. In *Proceedings of the 3rd IEEE Conference on Artificial Intelligence Applications, Orlando*, pages 315–320, February 1987.

[Gonzalez and Barr, 2000] A. J. GONZALEZ and V. BARR. Validation and verification of intelligent systems : what are they and how are they different? *JETAI : Journal of Experimental and Theoretical Artificial Intelligence*, 12:407–420, Octobre 2000. http://isl.engr.ucf.edu/publication/papers/VVIS/JETAI-00-Valerie.PDF visité le 31 janvier 2003.

[Hallal and Petrenko, 2001] H. HALLAL and A. PETRENKO. Pattern based analysis of communication traces for distributed systems. Technical Report, CRIM, octobre 2001. www.crim.ca.

[Hilaire *et al.*, 2000] V. HILAIRE, A. KOUKAM, P. GRUER, and J. P. MÜLLER. Formal specification and prototyping of multi-agent systems. In *Engineering Societies in the Agents World*, volume 1972 of *LNAI*, pages 114–128. Springer-Verlag, December 2000. 1st International Workshop (ESAW'00), Berlin (Germany), 21 August 2000, Revised Papers.

[Hoare, 1969] C. A. R. HOARE. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. http://doi.acm.org/10.1145/363235.363259.

[Inverno *et al.*, 1997] M. INVERNO, D. KINNY, M. LUCK, and M. WOOLDRIDGE. A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176, 1997. citeseer.nj.nec.com/dinverno97formal.html.

[Inverno and Luck, 2001] M. INVERNO and M. LUCK. *Understanding Agent Systems.* Springer Series on Agent Technology, 2001.

[Jarras, 1995] I. JARRAS. Vérification et synthèse d'un réseau de petri relativement à une spécification logique temporelle. Master's thesis, Université de Laval, Juillet 1995. http://www.ift.ulaval.ca/ desharnais/Recherche/Theses/memoire.Imed.pdf.

[Jarras and Chaib-draa, 2002] I. JARRAS and B. CHAIB-DRAA. Appercu sur les systèmes multiagents. http://econpapers.hhs.se/paper/circirwor/2002s-67.htm visité le 22 janvier 2003. Cirano, Juillet 2002.

[Jennings, 1996] N. R. JENNINGS. Coordination techniques for distributed artificial intelligence. *Foundations of Distributed Artificial Intelligence*, pages 187–210, 1996. http://www.ecs.soton.ac.uk/ñrj/pubs.html visité le 22 janvier 2003.

[Jennings, 1999] Nicholas R. JENNINGS. Agent-Oriented Software Engineering. In Francisco J. GARIJO and Magnus BOMAN, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Germany, 30– 2 1999. citeseer.nj.nec.com/article/jennings00agentoriented.html.

[Jennings *et al.*, 1998] N. R. JENNINGS, K. SYCARA, and M. WOOLDRIDGE. A roadmap of agent research and development. *Int Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998. http://www.ecs.soton.ac.uk/ñrj/pubs.html visité le 22 janvier 2003.

[Katoen, 2002] J. P. KATOEN. Principles of model checking. http://www.cs.auc.dk/%7Ekgl/DAT4F02/KatoenIntro.ps, décembre 2002.

[Koning *et al.*, 1995] J. KONING, Y. DEMAZEAU, B. ESFANDIARI, and J. QUINQUETON. Quelques perspectives d'utilisation des langages et protocoles d'interaction dans le contexte de télécommunications. In *Actes des Journées Francophones IAD-SMA, AFCET-AFIA*, Chambéry, 1995.

[Lamport, 1978] L. LAMPORT. Time, clocks and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, pages 558–565, Juilet 1978. Reprinted in several collections, including Distributed Computing: Concepts and Implementations, McEntire et al., ed. IEEE Press, 1984. http://research.microsoft.com/users/lamport/pubs/time-clocks.pdf.

[Magnin *et al.*, 2002] L. MAGNIN, T. V. PHAM, A. DURY, N. BESSON, and A. THIEFAINE. Our guest agents are welcome to your agent platforms. In *the ACM Symposium on Applied Computing (SAC 2002)*, pages 107–114, Madrid, Spain, 2002.

[Martial, 1992] F. Von. MARTIAL. *Coordinating Plans of Autonomous Agents.* Springer Verlag, 1992.

[Mattern, 1989] F. MATTERN. Virtual time and global states of distributed systems. In Raynal COSNARD, Quinton and Robert EDITORS, editors, *Workshop on Parallel and Distributed Algorithms*, pages 215–226, Bonas (France), 1989.

[Mazouzi, 2001] H. MAZOUZI. *Ingénierie des protocoles d'interaction: des systèmes distribués aux systèmes multi-agents.* PhD thesis, Parix IX-Dauphine, Octobre 2001.

[Odell *et al.*, 2002] J. ODELL, H. Van. Dyke PARUNAK, M. FLEISCHER, and S. BRUECKNER. Modeling agents and their environment. In *AOSE Workshop at AAMAS 2002*, 2002. http://www.jamesodell.com visité le 22 janvier 2003.

[Polat and Güvenir, 1991] F. POLAT and H. Altay GÜVENIR. A unification-based approach for knowledge base verification. *Expert Systems: The International Journal of Knowledge Engineering*, 8(4):251–259, 1991.

[Reed *et al.*, 2002] C. REED, T. J. NORMAN, and N. R. JENNINGS. Negotiating the semantics of agent communication languages. *Computational Intelligence*, 18(2):229–252, 2002. http://www.ecs.soton.ac.uk/ñrj/pubs.html visité le 22 janvier 2003.

[SA, 1999] Verilog SA. *ObjectGEODE Tutorial Version 4.0.* Verilog SA, Avril 1999. www.control.auc.dk/ henrik/undervisning/ embedd/tutorial.pdf.

[Schmidt, 1990] K. SCHMIDT. *Distributed Decision Making: Cognitive Models for Cooperative Work*, Chapitre Cooperative Work: A Conceptual Framework. Wiley & Sons, 1990.

[Shoham, 1994] Y. SHOHAM. Agent oriented programming: An overview of the framework and summary of recent research. In M. MASUCH and L. POLOS, editors, *Knowledge Representation and Reasoning under Uncertainty: Logic at Work*, pages 123–129. Springer, Berlin, Heidelberg, 1994.

[Sommerville, 1996] I. SOMMERVILLE. *Software Engineering*, Chapitre 22,23,24, pages 445–501. International Computer Sciences Series. Addison-Wesley, 5th edition, 1996.

[URL1,] URL1. Cmu model cheking group. http://www-2.cs.cmu.edu/m̃odelcheck/smv.html.

[URL2,] URL2. Aglets. http://www.alphaworks.ibm.com/tech/aglets, visité en avril 2003.

[URL3,] URL3. Jade. http://jade.cselt.it, visité en avril 2003.

[URL4,] URL4. ObjectGeode. http://www.telelogic.com/, visité en mars 2003.

[Wen and Mizoguchi, 1999] W. Wen and F. Mizoguchi. Analysis and verification of multi-agent interaction protocols. In *Proceedings of the third annual conference on Autonomous Agents*, pages 372–373. ACM Press, Avril 1999. http://doi.acm.org/10.1145/301136.301240.

[Wooldridge, 1992] M. Wooldridge. *On the Logical Modelling of Computational Multi-Agent Systems.* PhD thesis, Faculty of Technology in the University of Manchester, Manchester, UK, october 1992. citeseer.nj.nec.com/wooldridge92logical.html visité le 25 Février 2003.

[Wooldridge and Ciancarini, 2000] M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag, Berlin, 2000. citeseer.nj.nec.com/wooldridge00agentoriented.html visité le 01 Février 2003.

[Wuu and Bernstein, 1984] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 233–242, 1984.

[Zlatareva, 1992] N. Zlatareva. A framework for knowledge-based system validation, verification and refinement: the vvr system. In *FLAIRS*, pages 10–14, Ft. Lauderdale, FL, 1992.

Annexe A

La grammaire BNF du langage GOAL

La grammaire du langage GOAL est la suivante:

```
-- "observation" level
<observation> ::=
     OBSERVATION OF <system NAME> end
          { <observation body> } +
     ENDOBSERVATION [<system NAME>] end

<observation body> ::=
      <synonym definition part>
     | <type definition part>
     | <probe declaration>
     | <observer definition part>

-- observer level
<observer definition part> ::=
     [ EVENT ] OBSERVER <observer NAME> <end>
          { <obs declaration part> }  *
          <obs body>
     ENDOBSERVER [<observer NAME>] end

-- declaration in an observer
<obs declaration part> ::=
      <synonym definition part>
     | <type definition part>
     | <procedure definition part>
     | <variable definition part>
     | <probe declaration>
     | <state set definition part>

-- probe declaration
<probe declaration> ::=
     PROBE <probe NAME> <model entity path> <end>
<model entity path> ::=
```

```
        SYSTEM
        | <path item>
        | <model entity path> ! <path item>
    <path item> ::=
        <block NAME>
        | <process NAME> [ (<path instance number>) ]
        | <procedure NAME>
    <path instance number> ::=
        <integer ground expression>

    -- definition of constants
    <synonym definition part> ::=
         <synonym definition>

    -- type definition
    <type definition part> ::=
         <partial type definition>

    -- procedure definition
    <procedure definition part> ::=
         <procedure definition>

    -- observer variables
    <variable definition part> ::=
         DCL <name list> <sort> [ ":=" <expression> ]
<end>

    -- observer states classification
    <state set definition part> ::=
        ERROR STATE <state name list> <end>
        | SUCCESS STATE <state name list> <end>

    <state name list ::=
         <name list>

    -- observer body
    <obs body> ::=
         <start> {<state> } *

    <start> ::=
         START <end> <transition>

    <state> ::=
         STATE
              <state list> <end>
              { <transition header> <transition> } *
         [ENDSTATE [ state NAME ] <end>]

    <state list> ::=
         <state list> Z.100-C2-22

    -- transition clauses
    <transition header> ::=
         WHEN <event observation predicate> [<enabling
condition>]
```

```
          | <continuous signal>

     -- WHEN clause
     <event observation predicate> ::=
          <interaction event>
          | <transition event>
          | <call event>
          | <create event>
          | <stop event>

     -- observation of a transition firing
     <transition event> ::=
         TRANS <probe reference>
         | [ <event naming> ] TRANS <probe reference> [ :
<transition NAME>  ]
         | [ <event naming> ] TRANS TIME

     -- observation of communication events
     <interaction event> ::=
         [ <event naming> ] { <input event> | <output
event> }

     <input event> ::=
         [ TRUE | FALSE ] INPUT [ <signal reference> ]
             [ <from part> ]
             [ <to part> [ <via part> ] ]

     <output event> ::=
         [ TRUE | FALSE ] OUTPUT [ <signal reference> ]
             [ <from part> [ <via part> ] ]
             [ <to part> ]

     <from part> ::=
         FROM { <probe reference> | ENV }

     <to part> ::=
         TO { <probe reference> | ENV }

     <via part> ::=

         VIA <route or channel NAME>

     <event naming> ::=
         <variable NAME> :=

     -- observation of a procedure call
     <call event> ::=
          CALL [<procedure probe reference>]

     -- observation of a process creation
     <create event> ::=
     CREATE [<process probe reference>] [ FROM <process
probe reference> ]

     -- observation of a process stop
```

```
     <stop event> ::=
          STOP [ <process probe reference> ]

     <enabling condition> ::=
          <enabling condition>

     -- continuous signal clause
     <continuous signal ::=
          <enabling condition>

     -- transition
     <transition> ::=
          <transition string> [<terminator statement>]
          | <terminator statement>

     <transition string> ::=
          { [<label>] <transition action> <end> } +

     <transition action> ::=
          <task>
          | <decision>
          | <procedure call>
          | <write statement>
          | <loop statement>
          | <if statement>
          | <assign statement>


     <terminator statement> ::=
          <terminator statement>

     -- task
     <task> ::=
          <task>

     -- decision
     <decision> ::=
          <decision>

     -- procedure call
     <procedure call> ::=
          CALL <procedure reference> [<actual
parameters>]

     <actual parameters> ::=
          <actual parameters> Z.100-C2-45

     -- displays
     <write statement> ::=
          WRITE "(" <write arguments> ")"
          | WRITELN [ "(" <write arguments> ")" ]

     <write arguments> ::=
          <expression> { , <expression> } *
```

```
     -- loop statements
     <loop statement> ::=
          FOR NAME '=' <expression> TO <expression> <end>
               <transition actions> ENDFOR
          | FOR ALL <iterator NAME> IN <sort> <end>
               <transition action> ENDFOR
          | FOR ALL <iterator NAME> IN PROBE <probe array
NAME> <end>
               <transition action> ENDFOR
          | WHILE <expression> <end>
               <transition action> ENDWHILE

     -- if statement
     <if statement> ::=
          IF <expression> THEN <transition> ELSE
<transition> FI

     -- assign statement
     <assign statement> ::=
          <expression> ':=' <expression>

     -- sort
     <sort> ::=
          <sort identifier>
          | <syntype identifier>
          | <model sort reference>

     <model sort reference> ::=
          <probe reference> ! <type identifier>

     -- extended primary expression
     <primary> ::=
          <primary>
          | <event observation predicate>
          | <model variable reference>

     -- variable and constant observation
     <model variable reference> ::=
          <probe reference> ! [ <type qualifier> ]
<object name>

     <type qualifier> ::= TYPE <sort>

     -- reference to a probe
     <probe reference> ::=
          <probe NAME>
          | <probe array NAME> (<instance number>)

     <instance number> ::=
          <integer expression>

     <object name> ::=
          NAME
          | START
          | STATE
```

```
-- identifier
<identifier> ::=
     <identifier>

-- label
<label> ::=
     <label>

-- end
<end> ::=
     <end>

<comment> ::=
     [ COMMENT <character string> ]

<name list> ::=
     NAME { "," NAME } *

-- keywords
<keyword> ::=
     <keyword>
     | OBSERVATION
     | OBSERVER
     | EVENT OBSERVER
     | OF
     | ENDOBSERVATION
     | ENDOBSERVER
     | PROBE
     | ERROR STATE
     | SUCCESS STATE
     | WHEN
     | TRUE INPUT
     | FALSE INPUT
     | TRUE OUTPUT
     | FALSE OUTPUT
     | TRANS
     | TIME
     | WRITE
     | WRITELN
```

Annexe B

La grammaire de la trace d'exécution

La grammaire de la trace d'exécution du système est aussi définie en XML. Cette dernière couvre les différents éléments qui doivent figurer dans la trace d'exécution pour garantir une bonne vérification. La trace est composée par au moins un événement qui peut être soit un événement local ou un événement de communication. Dans tous les cas, cet événement est défini par son estampille, son type (création de l'agent, destruction de l'agent ou communication) et l'opération effectuée lors de cet événement (envoi, réception de message...). Les paramètres de l'événement doivent contenir la source ou le destinaire du message inhérent à l'événement, l'identifiant du scénario du protocole *correlation-id* et l'identifiant du message. De plus, si l'événement est de type communication, alors nous pourrons définir les différents paramètres du message.

```
<?xml version="1.0" encoding="UTF-8" ?>

<!--*****************************************-->
<!--* grammar of event traces (AU 2002/05/16) *-->
<!--*****************************************-->

<!ELEMENT trace (project, events)>
<!ATTLIST trace date              CDATA #REQUIRED
                generator-name    CDATA #IMPLIED
                generator-version CDATA #IMPLIED>

<!ELEMENT project EMPTY>
<!ATTLIST project name              CDATA #REQUIRED
                  version           CDATA #IMPLIED>

<!ELEMENT events (event*)>

<!ELEMENT event (parameters, local?, message?)>
<!ATTLIST event timestamp CDATA #REQUIRED
```

```
                type      CDATA #REQUIRED
                operation CDATA #REQUIRED>

<!ELEMENT parameters EMPTY>
<!ATTLIST parameters identifier     CDATA #IMPLIED
                     rtc-frequency  CDATA #IMPLIED
                     device-id      CDATA #IMPLIED
                     child-id       CDATA #IMPLIED
                     parent-id      CDATA #IMPLIED
                     object-id      CDATA #IMPLIED
                     object-type    CDATA #IMPLIED
                     thread-id      CDATA #IMPLIED
                     correlation-id CDATA #IMPLIED
                     message-id     CDATA #IMPLIED>

<!ELEMENT message (parameter+)>

<!ELEMENT local (parameter+)>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name  CDATA #REQUIRED
                    type  CDATA #REQUIRED
                    value CDATA #REQUIRED>
```