Vérification formelle des systèmes multi-agents auto-adaptatifs

Zeineb Graja

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE ET DE L'UNIVERSITÉ DE SFAX

Délivré par
Université Toulouse III - Paul Sabatier **et**
Faculté des Sciences Economiques et de Gestion - Sfax

Discipline ou spécialité :
Informatique

Présentée et soutenue par
Zaineb Graja
Le
15 septembre 2015
Titre :
Vérification formelle des systèmes multi-agents auto-adaptatifs

---

JURY
Leila Jemni Ben Ayed, Maître de conférences (Hdr), Université de la Manouba (rapporteur)
Marie-Pierre Gleizes, Professeur, Université Paul Sabatier – Toulouse III (directrice)
Ahmed Hadj Kacem, Professeur, Université de Sfax (directeur)
Dominique Méry, Professeur, Université de Lorraine (examinateur)
Yamine Ait Ameur, Professeur, INPT-ENSEEIHT - Université de Toulouse (examinateur)
Frédéric Migeon, Maître de conférences, Université Paul Sabatier – Toulouse III (encadrant)
Abderrafiaa Koukam, Professeur, Université de Technologie de Belfort-Montbéliard (rapporteur)

Ecoles doctorales : Mathématiques Informatique Télécommunications (MITT) et Sciences Economiques, Gestion et Informatique
Unités de recherche : Institut de Recherche en Informatique de Toulouse (IRIT) et Laboratoire de Recherche en Développement et Contrôle d'applications Distribuées (ReDCAD)
Directeur(s) de Thèse : Marie-Pierre Gleizes, Ahmed Hadj Kacem et Frédéric Migeon
Rapporteurs : Leila Jemni Ben Ayed et Abderrafiaa Koukam

Zaineb Graja

VERIFICATION FORMELLE DES SYSTEMES MULTI-AGENTS AUTO-ADAPTATIFS

Directeurs de thèse :
Marie-Pierre Gleizes, Ahmed Hadj Kacem et Frédéric Migeon

Résumé

Un Système Multi-Agents (SMA) est formé par un grand nombre d'éléments appelés agents capables d'interagir ensemble afin de réaliser une tâche commune. Ces agents n'ont aucune information sur la fonction globale qu'ils doivent réaliser et fonctionnent généralement selon des règles comportementales simples. Pour décider des actions à entreprendre, ils reposent uniquement sur des perceptions partielles de leur voisinage. On dit que la fonction globale du système émerge des interactions entre les agents.
Un des défis majeurs pour le développement des SMA auto-organisateurs est de garantir la convergence du système vers la fonction globale attendue par un observateur externe au système et de garantir que les agents sont capables de s'adapter face aux perturbations. Dans la littérature, plusieurs travaux se sont basés sur la simulation et le model-checking pour analyser les SMA auto-organisateurs. La simulation permet aux concepteurs d'expérimenter plusieurs paramètres et de créer certaines heuristiques pour faciliter la conception du système. Le model-checking fournit un support pour découvrir les blocages et les violations de propriétés.
Cependant, pour faire face à la complexité de la conception des SMA auto-organisateurs, le concepteur a également besoin de techniques qui prennent en charge non seulement la vérification, mais aussi le processus de développement lui-même. En outre, ces techniques doivent permettre un développement méthodique et faciliter le raisonnement sur divers aspects du comportement du système à différents niveaux d'abstraction. Dans ce cadre, nous jugeons que le langage formel B-événementiel et la technique de raffinement associée fournissent une base appropriée pour le développement formel des SMA auto-organisateurs.
Dans cette thèse, trois contributions essentielles ont été apportées dans le cadre du développement et la vérification formelle des SMA auto-organisateurs : une formalisation à l'aide du langage B-événementiel des concepts clés de ces systèmes en trois niveaux d'abstraction, une expérimentation d'une stratégie de raffinement descendante pour le développement des SMA auto-organisateurs et la proposition d'un processus de raffinement ascendant basé sur des patrons de raffinement.
La première contribution est axée sur la formalisation de ces systèmes sur trois niveaux d'abstraction : (1) le niveau micro correspond aux comportements locaux des entités du système –les agents (le SMA) et l'environnement– considérés unilatéralement ; (2) le niveau méso correspond à la description des interactions du SMA dans l'environnement ; (3) le niveau macro décrit le comportement du système dans sa globalité observé par un observateur externe. Cette formalisation est réalisée à l'aide du langage B-événementiel et nous a permis de prouver des propriétés locales. Afin de prouver des propriétés globales liées à la convergence et la résilience, nous avons eu recours à la logique temporelle des actions (TLA).
La deuxième contribution nous a permis d'expérimenter un processus descendant et de motiver par conséquent le choix d'adopter un processus ascendant pour la vérification des SMA auto-organisateurs.

La troisième contribution réside dans la définition d'un processus ascendant pour le développement de SMA auto-organisateurs basé sur des patrons de raffinement. Ces patrons apportent une assistance au concepteur pour développer un comportement correct des agents aux niveaux micro et méso. Ils offrent également des guides pour prouver les propriétés de convergence et de résilience observées à l'aide des simulations au niveau macro du système.

Zaineb Graja

FORMAL VERIFICATION OF SELF-ADAPTIVE MULTI-AGENTS SYSTEMS

Supervisors :
Marie-Pierre Gleizes, Ahmed Hadj Kacem and Frédéric Migeon

Abstract

A Multi-Agent System (MAS) is composed by a large number of elements called agents interacting together in order to perform a common task. These agents have no information about the global function they have to perform and operate according to simple behavioral rules. Their decisions are based only on partial representations of their environment. The overall function of the system is said to emerge from the interactions between agents.
A major challenge for the development of self-organizing MAS is to guarantee the convergence of the system to the overall function expected by an external observer and to ensure that agents are able to adapt to changes. In the literature, several works were based on simulation and model-checking to study self-organizing MAS. The simulation allows designers to experiment various settings and create some heuristics to facilitate the system design. Model checking provides support to discover deadlocks and properties violations.
However, to cope with the complexity of self-organizing MAS, the designer also needs techniques that support not only verification, but also the development process itself. Moreover, such techniques should support disciplined development and facilitate reasoning about various aspects of the system behavior at different levels of abstraction. In this context, we believe that the Event-B language and associated technical refinement provide an appropriate basis for the formal development of self-organizing MAS.
In this thesis, three essential contributions were made in the field of formal development and verification of self-organizing MAS : a formalization with the Event-B language of self-organizing MAS key concepts into three levels of abstraction, an experimentation of a top-down refinement strategy for the development of self-organizing MAS and the definition of a bottom-up refinement process based on refinement patterns.
The first contribution focuses on the formalization of these systems on three levels of abstraction : (1) the micro level corresponds to the local behavior of the entities –the MAS and the environment– considered unilaterally ; (2) the meso level is a description of the interactions of the MAS and its environment ; (3) the macro level describes the behavior of the overall system observed by an external observer. This formalization is performed using the Event-B language and allows us to prove the local properties. In order to prove the global properties related to convergence and resilience, we use the temporal logic of actions (TLA).
The second contribution has enabled us to experiment a top-down process and therefore to motivate the choice to adopt a bottom-up process for the verification of self-organizing MAS. The third contribution is a definition of a bottom-up process for the development of self-organizing MAS based on refinement patterns. These patterns provide assistance to the designer to develop a correct behavior of the agents at the micro and meso levels. They also offer guidelines to prove the convergence and resilience properties observed using simulations at the system macro level.

Je dédie cette thèse
à la mémoire de Baba Hadj Salah,
mon grand père.

Remerciements

« Si j'ai vu si loin, c'est que j'étais monté sur des épaules de géants »
Isaac Newton

VIENS enfin le sacré moment de la rédaction des remerciements. J'avoue que c'est un moment plein d'émotions où les souvenirs se bousculent dans ma tête et où larmes, joie et fierté viennent se mêler. Ce moment marque pour moi la fin d'une aventure exceptionnelle de la quelle je sort grandie et qui n'a pas pu avoir tant de charme sans la présence de ces personnes qui m'ont épaulé durant toutes ces années et auxquelles j'ai le plaisir d'adresser mes sincères remerciements.

Je tiens à remercier ...

... Ahmed Hadj Kacem, Professeur à la Faculté des Sciences Economiques et de Gestion de Sfax. Pour la confiance que tu m'as accordée depuis que j'ai été étudiante en mastère en me mettant en contact avec Marie-Pierre et en me recommandant à elle pour faire cette thèse en cotutelle. Tes encouragements et tes conseils m'ont été d'une grande utilité. J'espère que je ne t'ai pas déçu.

... Marie-Pierre Gleizes, Professeur à l'Université Paul Sabatier et chef de l'équipe SMAC. Pour m'avoir accueilli chaleureusement dans ton équipe et m'avoir permis de "grandir" dans une ambiance si active et dynamique. Ton dévouement à la recherche, à la science, aux SMA m'inspire. Pour moi tu es beaucoup plus qu'une directrice de thèse,... tu es une idole.

... Frédéric Migeon et Christine Maurel, Maîtres de conférences à l'Université Paul Sabatier. Pour avoir été là chaque fois que j'avais besoin de vous. Vous avez toujours su, grâce à votre optimisme, votre bonne humeur et votre patience, me motiver, rebooster ma confiance en moi et m'aider à m'accrocher. Frédéric, si j'ai fait mon inscription à l'université Paul Sabatier, c'est en grande partie grâce à toi. Je n'oublierai jamais comment tu t'es battu pour cela. Cette inscription m'a ouvert beaucoup de portes.

... Eléna Troubitsyna et Linas Laibinis, Professeurs à l'Université d'Åbo (Turku). Pour m'avoir accueilli dans votre équipe durant trois mois. Les discussions et les échanges que j'ai eus avec vous m'ont permis de déchiffrer plusieurs secrets autour de la modélisation avec B-événementiel. Mon stage en Finlande n'a pas marqué mon travail de thèse uniquement

mais aussi ma vie et c'est toujours un sentiment de fierté qui m'envahit en regardant mon nom inscrit á côté des vôtres sur notre papier.

... Mme. Leila Jemni et Mr. Abderrafiaa koukam, respectivement Maître de conférences Habilitée à l'Université de la Manouba et Professeur à l'Université de Technologie de Belfort-Montbéliard. Pour avoir accepter de rapporter ce manuscrit. Les remarques que vous m'avez faites ont contribué à améliorer sa qualité. Vous me faites honneur d'être parmi le jury de ma thèse.

... Mr. Dominique Méry et Mr. Yamine Ait Ameur, respectivement Professeur à l'Université de Lorraine et Professeur à l'Université de Toulouse. Il fût un grand honneur pour moi lorsque je vous ai rencontré pour la première fois à Turku il y deux années. Je vous remercie pour avoir répondu présents chaque fois que j'ai sollicité votre aide, pour avoir accepté de collaborer à évaluer mon travail et d'être parmi le jury de ma thèse.

... Mohamed Tounsi. Pour avoir partagé avec moi les plus beaux souvenirs à la faculté aussi bien les meilleurs que les pires. Tout au long de ces années de thèse, tu ne m'as jamais épargné de tes conseils et de tes recommandations tant au niveau scientifique qu'au niveau personnel. Que notre amitié dure encore et encore.

... Les SMACiens (comme vous aimez qu'on vous appelle) permanents et doctorants. Pour m'avoir accepté parmi vous. L'ambiance qui règne au troisième étage chez SMAC m'a trop aidé à mener au mieux ce travail. Je pense particulièrement aux doctorants avec qui j'ai partagé le bureau : Jérémy, Charles, Julien, Valérien, Sylvain et tous ceux que j'ai côtoyés durant mes séjours : Luc, Simon, Raja, Sameh,... J'ai eu la chance de faire partie d'une équipe qui incarne si merveilleusement la coopération. Vous allez me manquer.

... Imen, Fatma, Ilhem, Sirine, Faten, Fairouz et Wael, mes amis du laboratoire ReDCAD. Pour les beaux moments que nous avons partagés au laboratoire, nos discussions variées et l'ambiance qu'à chaque fois nous créons durant les séminaires du laboratoire. Je garde de très beaux souvenirs avec vous.

... Hilal, Paola et Demir. Vous êtes des cadeaux du ciel. Débarquée en Finlande, j'ai pu retrouver la chaleur du sud dans votre accueil, vos mots et votre attitude. Grâce à vous, je me suis sentie chez moi dès mes premiers jours au Retrodorm.

... Amira. Tu es plus qu'une amie pour moi, tu es ma sœur. Cette dernière année de thèse a été très dure pour moi et toi tu as toujours su comment me réconforter et m'aider par ta générosité, tes conseils et tes recommandations souvent échangés autour d'un repas ou d'un café turc.

... Asma, Amina et Soumaya, mes sœurs ainsi que mes parents, le "Nour" de mes yeux. Merci pour vos encouragements, votre soutient, votre amour sans égal. Maman et Papa, ma réussite c'est la votre. Je vous remercie pour la confiance que vous avez faite en moi. J'espère que vous êtes fiers de moi.

... Radhouane, mon autre. Ton apparition, très surprenante, a été pendant les moments les plus chaotiques de ma thèse. Mais depuis, tu traces les meilleurs jours de ma vie. Tu me combles d'affection, d'amour et de bonheur. Merci pour ton grand cœur.

Toulouse le 12 septembre 2015

Table des matières

Introduction

La complexité des applications informatiques actuelles connait une croissance importante. Ceci est dû à plusieurs facteurs : l'hétérogénéité des composants qui forment ces systèmes, la nature instable et dynamique de l'environnement dans lequel ils sont plongés, les interactions imprévisibles pouvant survenir entre les composants, la distribution des données et la décentralisation du contrôle. Pour faire face à cette complexité, ces systèmes doivent être conçus de manière à ce qu'ils soient capables de gérer par eux-mêmes ces imprévisibilités et cette dynamique en s'adaptant aux changements qui viennent perturber leur fonctionnement.

Les systèmes multi-agents auto-organisateurs, dont le principe de fonctionnement est inspiré principalement des systèmes naturels, offrent un cadre idéal pour le développement de systèmes exigeant des facultés d'auto-adaptation. En effet, les observations des sociétés d'insectes ont révélé que ces minuscules entités dotées de règles comportementales simples sont capables de faire émerger, via leurs interactions, des tâches ou des structures collectives extrêmement complexes. L'objectif principal des recherches dans ce domaine est de définir les mécanismes adéquats pour guider le comportement des agents au niveau micro, les aider à s'auto-organiser pour faire émerger au niveau macro, le comportement du système attendu. Les propriétés au niveau macro sont dites des propriétés émergentes.

La conception des systèmes auto-organisateurs se fait selon un processus ascendant. Ainsi, les efforts du concepteur sont centrés autour du développement des comportements locaux des agents. La fonctionnalité globale émerge des interactions entre les agents et des interactions des agents avec leur environnement. Le défi majeur lors de ce processus est de fournir les garanties nécessaires que la fonction qui émerge correspond à une fonction attendue par un observateur externe au système (la convergence) et que le système est capable de s'adapter aux perturbations (la résilience).

Dans la littérature, plusieurs travaux se sont basés sur la simulation et le model-checking pour étudier les propriétés des SMA auto-organisateurs. La simulation permet aux concepteurs d'expérimenter plusieurs paramètres et de créer certaines heuristiques pour faciliter la conception du système. Le model-checking fournit un support pour découvrir les blocages et les violations de propriétés.

Cependant, pour faire face à la difficulté de la conception des SMA auto-organisateurs, le concepteur a également besoin de techniques qui prennent en charge non seulement la vérification, mais aussi le processus de développement lui-même. En outre, ces techniques doivent permettre un développement méthodique et faciliter le raisonnement sur divers aspects du comportement du système à différents niveaux d'abstraction.

Ce travail a consisté à proposer un cadre formel pour assurer une vérification formelle des SMA auto-organisateurs. Nous offrons à des concepteurs de SMA auto-organisateurs non spécialistes des méthodes formelles, des outils relativement intuitifs pour prouver des propriétés de convergence et de résilience sur les systèmes conçus.
Nos contributions s'inscrivent sur deux plans :

- *sur le plan théorique* :
  Nous avons proposé une formalisation des SMA auto-organisateurs à l'aide du langage B-événementiel ainsi qu'une formalisation des propriétés du niveau local (propres aux agents) et des propriétés de convergence et de résilience du niveau global (relatives au système) à l'aide de la logique temporelle. Nous considérons que ces propriétés émergentes sont déjà observées (par simulation par exemple) et nous souhaitons fournir des garanties formelles qu'elles seront vérifiées.

  Cette formalisation a servi de base pour prouver ces propriétés. Les propriétés locales ont été assurées à l'aide de la logique des prédicats de premier ordre en prouvant des théorèmes de non blocage. Pour montrer les propriétés globales, nous avons eu besoin de faire des hypothèses d'équité sur les activités locales des agents. Ainsi, nous avons eu recours à la logique temporelle des actions (TLA, Temporal Logic of Actions) qui offre un cadre de raisonnement formel plus intuitif dans notre cas.

- *sur le plan méthodologique* :
  Nos travaux se situent au carrefour de deux disciplines : les SMA auto-organisateurs et la vérification formelle. Chacune de ces disciplines possède ses propres méthodes et techniques qui ne sont pas nécessairement compatibles entre elles. Du côté des méthodes formelles, la plupart des travaux proposent des méthodes de raffinement descendantes. Ces méthodes descendantes préconisent de commencer par spécifier l'objectif global du système à concevoir puis de procéder à des étapes de raffinements permettant de décomposer la tâche globale en des tâches élémentaires attribuées ensuite aux différents éléments du système. Du côté des SMA auto-organisateurs, les concepteurs adoptent les méthodes ascendantes pour développer et vérifier leurs systèmes. Tout l'effort de conception est alors orienté vers le comportement local des agents.
  Vu cet écart entre les techniques adoptées par chacun de ces deux domaines, la première réflexion de notre travail a été de trouver la bonne méthode pour assurer une vérification formelle des SMA auto-organisateurs.
  Ainsi, nous avons expérimenté dans un premier temps la méthode descendante pour tester sa compatibilité avec le développement et la vérification des SMA auto-organisateurs. A l'issue de ce test [Laibinis *et al.*, 2014], nous avons réussi à comprendre la logique de preuve utilisée avec l'approche correcte par construction et l'adapter par

conséquent aux exigences des développeurs de SMA auto-organisateurs. Dans un second temps, nous avons proposé le processus ascendant *RefProSOMAS* composé par des étapes de raffinements incrémentales. Ce processus repose sur des patrons de raffinement et a pour objectif de guider le concepteur (grâce à ces patrons notamment) pour obtenir des spécifications de comportements locaux corrects. Ces patrons offrent aussi des guides pour prouver des propriétés de convergence et de résilience attendues par un observateur externe au système.

Ce manuscrit est organisé en trois parties. La première est constituée de trois chapitres qui présentent le contexte et l'état de l'art du domaine dans lequel s'inscrit notre travail. Le premier chapitre décrit les caractéristiques des systèmes informatiques complexes et présente les systèmes multi-agents auto-organisateurs comme approche pour faire face à cette complexité. Dans le deuxième chapitre, nous dressons un état de l'art sur les différentes techniques utilisées dans le domaine de la vérification et la validation des systèmes auto-adaptatifs de manière générale et les systèmes auto-organisateurs en particulier. Au terme de cet état de l'art, nous exposons les motivations de cette thèse et nous énonçons nos objectifs de recherche. Le dernier chapitre de cette première partie présente les outils formels utilisés en décrivant les notions relatives au langage B-événementiel et la logique TLA.
La description de notre contribution dans le cadre de la vérification formelle des SMA auto-organisateurs se fait dans la seconde partie en deux chapitres. Dans le premier, nous présentons une formalisation de ces systèmes selon trois niveaux d'abstraction. Cette formalisation est ensuite traduite au langage B-événementiel dans l'objectif de définir un processus incrémental de vérification. Le deuxième chapitre décrit un processus incrémental ascendant pour la vérification de SMA auto-organisateurs basé sur des patrons de raffinement. Les motivations de ce choix ascendant sont également exprimées dans ce chapitre. Ce processus garantit les propriétés locales du niveau micro ainsi que les propriétés de convergence et de résilience du niveau macro.
Dans la troisième partie, une illustration de ce processus est donnée en se servant du cas d'étude des fourmis fourrageuses. Nous présentons comment la plateforme Rodin a été utilisée pour instancier les patrons proposés et comment nous nous sommes servis de la logique TLA pour mener la preuve des propriétés du niveau macro.

Finalement, ce document est conclu par une synthèse de nos travaux et une ouverture sur des perspectives futures.

Première partie

Contexte et état de l'art

1 Les systèmes informatiques complexes

« L'intelligence, c'est la faculté d'adaptation »

André Gide

EN Octobre 2001, IBM a publié un manifeste ([Horn, 2001]) mentionnant que le principal obstacle au progrès dans l'industrie des technologies de l'information est la crise de la complexité logicielle. En effet, grâce à l'évolution technologique et des télécommunications, les systèmes informatiques sont de plus en plus distribués. Les données sont réparties sur des composants hétérogènes et le contrôle ne peut plus être assurée par une seule entité. La dynamique et l'imprévisibilité de l'environnement dans lequel sont plongés ces gigantesques systèmes rajoutent un autre niveau de complexité et leurs exigent de s'adapter constamment pour continuer à assurer leurs fonctions. Désormais, l'intelligence des systèmes informatiques ne réside plus dans des capacités cognitives exceptionnelles mais dans leur capacité à s'auto-adapter.

Face aux limites de la capacité humaine à gérer cette complexité, de nouvelles méthodes de conception et d'implémentation des systèmes informatiques se sont imposées. Des réflexions sur la nature de ces méthodes ont abouti à la proposition de systèmes dits *auto-réparateurs*, *auto-configurés*, *auto-optimisés*, *auto-adaptatifs*, *auto-organisateurs*, et bien plus encore. Ces systèmes sont connus aussi sous le nom de systèmes *auto*-* (ou *self*-*). Les *systèmes autonomiques*, initiative lancée par IBM en 2001 [Kephart et Chess, 2003], représentent une vision pour implémenter des systèmes dits *auto-gérés* regroupant une partie des propriétés *auto*-*. D'autres propositions inspirées des organismes vivants ont donné lieu aux systèmes auto-organisateurs tels que les robots à essaims ([Aleksandar et Diego, 2007], [Shi *et al.*, 2012]). Dans ce chapitre, nous introduisons d'abord cette notion de systèmes autonomiques en focalisant sur leurs propriétés d'*auto-gestion* (*self-management*). Nous évoquons ensuite les systèmes auto-organisateurs à fonctionnalités émergentes qui constituent le cadre de ce travail de thèse. Enfin, nous présentons les *systèmes multi-agent* auto-organisateurs, paradigme clé pour réaliser des systèmes complexes.

1.1 Systèmes autonomiques

Le concept de *systèmes autonomiques* a été proposé par IBM en 2001 pour désigner l'ensemble des systèmes informatiques possédant la capacité d'assurer leur auto-gestion en se basant sur les objectifs des administrateurs. L'auto-gestion a pour but de décharger les administrateurs des détails de fonctionnement et de maintenance du système et fournir aux utilisateurs une machine qui fonctionne convenablement. Tel un cerveau humain, un système autonomique doit maintenir et ajuster ses opérations en fonction du changement de ses composants, de sa charge de travail, des requêtes qui lui surviennent ainsi que les conditions externes et les défaillances qui pourront avoir lieu aussi bien pour le matériel que pour le logiciel. Un système autonomique est caractérisé par les quatre propriétés citées ci-dessous.

- Auto-configuration (Self-configuration) : le système est capable de modifier lui-même sa configuration automatiquement conformément à des politiques de configuration de haut niveau. Lorsqu'un nouveau composant est introduit dans le système, il va s'incorporer de façon transparente et le reste du système va s'adapter à sa présence.
- Auto-optimisation (Self-optimization) : le système cherche en permanence les opportunités pour améliorer sa performance et son efficacité.
- Auto-réparation (Self-healing) : le système détecte automatiquement les problèmes au niveau du matériel ou du logiciel et procède au diagnostic puis à la réparation.
- Auto-protection (Self-protection) : le système se défend automatiquement contre les défaillances et les attaques malveillantes. Il utilise les alertes précoces pour anticiper et prévenir les défaillances du système entier.

Pour atteindre ces propriétés *auto-** souhaitées, l'architecture observateur/contrôleur (observer/controller) a été proposée. Comme le montre la figure 1.1, cette architecture a pour objectif d'évaluer le comportement du système décentralisé (appelé *SuOC* : System under Observation/ Control) et de produire régulièrement des rétroactions (*feedbacks*) pour contrôler sa dynamique.

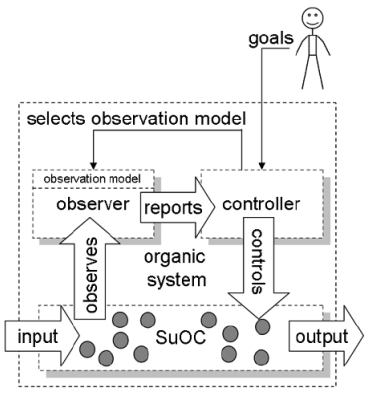

Figure 1.1 — Architecture observateur/contrôleur (adoptée de [Richter *et al.*, 2006])

Le contrôleur et l'observateur représentent la partie intelligente du système (appelée aussi la partie auto-*) et doivent assurer une boucle de surveillance et de contrôle appropriée. Cette boucle est formée de trois étapes :

- observer le comportement du système via les capteurs,
- comparer les résultats obtenus avec ceux attendus pour décider de l'action nécessaire et
- contrôler le système *SuOC* avec la meilleure action via les actionneurs.

L'action du contrôleur sur le système peut influencer le comportement de l'observateur en tirant son attention sur certaines observations intéressantes.

L'intelligence de la partie observateur/contrôleur ressort de sa capacité à sélectionner les informations pertinentes et à les utiliser de la meilleure façon. Ainsi, l'observateur doit être capable d'agréger les informations sur le système à contrôler sous forme d'indicateurs qui renseignent sur la globalité du système. Ces indicateurs vont lui servir pour prédire le comportement qui va émerger. Le contrôleur quant à lui doit trouver la bonne action pour guider le système à faire émerger la fonction désirée. Cette action peut consister à influencer le comportement local d'un élément du système *SuOC*, à modifier sa structure : les liens entre ses éléments ou encore à influencer l'environnement.

D'après la description de l'architecture observateur/contrôleur, il est clair que l'adaptation du système se fait d'une manière centralisée. En effet, toute l'intelligence du système est encapsulée dans la partie observateur/contrôleur qui se charge d'adapter le système. Les éléments du système sont des éléments passifs qui doivent subir les actions décidées par le contrôleur.

Pour concevoir des systèmes informatiques complexes, nous jugeons qu'il est primordial de décentraliser le contrôle de l'adaptabilité en dotant les éléments du système de beaucoup plus d'autonomie et de leur permettre de participer à l'adaptation du système. C'est bien cette vision qui est appliquée dans le cadre des systèmes auto-organisateurs présentés dans la section suivante.

1.2 Systèmes auto-organisateurs

Les sociétés d'insectes furent les premiers systèmes auto-organisateurs observés. Les études de Grassé [Grassé, 1959] ont révélé que les termites sont capables de construire des nids avec des architectures complexes formées par des ponts et des arches sans être supervisées par un contrôleur centralisé et en adoptant des comportements très simples. D'autres travaux ont dévoilé l'existence de plusieurs systèmes naturels exhibant des facultés d'auto-organisation.

Les colonies de fourmis Grâce à la stigmergie [1], les fourmis sont capables de coordonner leurs activités et faire apparaître des structures complexes non planifiées à l'avance telles que le recrutement alimentaire. En se servant du même principe de stigmergie, les fourmis sont capables de faire des choix collectifs efficaces tels que choisir la meilleure source de nourriture (la plus sucrée par exemple) ou encore de choisir le plus court chemin reliant leur nid et une source de nourriture (l'expérience des deux ponts [Bonabeau *et al.*, 1999]).

Le cerveau humain La science cognitive a révélé que le fonctionnement du cerveau se base essentiellement sur des cellules simples appelées neurones. Malgré leur simplicité, ces neurones sont responsables (grâce à leurs actions et aux connexions entre eux) de réaliser des activités très complexes telles que la perception, la pensée, les sentiments, la conscience, et bien d'autres importantes activités.

Le système immunitaire Le comportement du système immunitaire résulte des actions d'une multitude de cellules simples : les lymphocytes et les cellules T. Les actions de ces simples cellules peuvent être considérés comme une sorte de réseau chimique de traitement de signaux dans lequel la détection d'une bactérie par une cellule déclenche une cascade de signaux entre les cellules élaborant ainsi une réponse complexe.

Ces exemples d'auto-organisation naturelle ont fortement inspiré les chercheurs pour créer des modèles et des méthodes afin de résoudre des problèmes complexes tels que le problème de routage dans les réseaux, les problèmes d'optimisation, l'apprentissage des robots, la planification, etc. Ces méthodes de résolution considèrent que la solution au problème est une fonctionnalité émergente du processus d'auto-organisation des entités qui composent le système. Ainsi, un système auto-organisateur est défini comme étant un système formé de plusieurs *entités autonomes* ayant une *connaissance limitée* de leur environnement et qui *interagissent localement* en vue de produire un résultat. Ces entités autonomes travaillent de manière *décentralisée* et leur fonction globale *émerge* des interactions entre les différentes entités [Serugendo, 2009].

La figure 1.2 explique le principe d'auto-organisation comme un moyen pour adapter la fonctionnalité globale émergente du système à la dynamique de son environnement. Nous supposons que chaque partie P_i du système assure une fonction f_i. La fonction globale f_s du système S est obtenue par la combinaison des fonctions partielles f_i. Ceci est noté par l'expression $f_s = f_1 \ominus f_2 \ominus ... \ominus f_n$. L'opérateur $\ominus$ désigne l'opérateur de combinaison (illustré par les doubles flèches courbés sur la figure 1.2) et vérifie la propriété $f_1 \ominus f_2 \neq f_2 \ominus f_1$. Le changement de l'organisation (les liens d'interactions entre les parties du système) implique un changement dans la combinaison des fonctions partielles ce qui entraine le changement de la fonction globale du système.

La conception des systèmes auto-organisateurs passe alors par la définition des règles comportementales des entités du système et la manière avec laquelle ils interagissent localement afin de faire émerger et maintenir une organisation produisant la fonctionnalité adé-

1. Ce terme introduit par Grassé désigne une classe de mécanismes qui permet aux insectes de coordonner leurs activités au moyen d'interactions indirectes [Théraulaz, 2010]

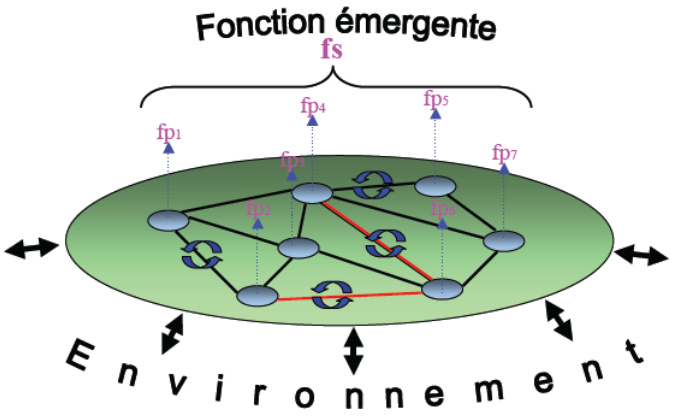

Figure 1.2 — Le principe d'auto-organisation (adoptée de [Georgé, 2004])

quate. Ce principe est utilisé pour développer des systèmes particuliers qui montrent des résultats prometteurs pour la simulation des systèmes complexes, appelés *systèmes multi-agents auto-organisateurs*.

1.3 Les SMA pour la simulation des systèmes complexes

Le paradigme de "Système Multi-Agents" (SMA) est apparu dans les années 80 comme une évolution du domaine de l'intelligence artificielle distribuée pour permettre une résolution collective et distribuée de problèmes. Depuis leur apparition, les SMA n'ont pas cessé de progresser et leur évolution peut être résumée en quatre générations [Di Marzo Serugendo *et al.*, 2011]. La première génération (1970-1980) des travaux s'est intéressée principalement à la résolution distribuée de problèmes avec un contrôle centralisé. La seconde génération des travaux (1980-1990) a accordé plus d'importance à la création de systèmes décentralisés et génériques pour favoriser la réutilisabilité. Durant le troisième génération (1990-2000), les interactions se sont mises au cœur des travaux sur les SMA. La génération actuelle focalise sur l'environnement et les manières avec lesquelles les SMA sont conçus pour s'adapter à la dynamique de l'environnement. Les SMA sont suffisamment matures pour simuler des systèmes complexes.

L'objectif de cette section est d'introduire des notions étroitement liées au domaine des SMA. Nous commençons par définir ce qu'est un SMA puis ce qu'est un agent. Nous décrivons ensuite qu'est ce qu'on entend dire par auto-adaptation et les moyens qui permettent de l'atteindre avec les SMA.

1.3.1 Système Multi-Agent

La diversité des travaux de recherche sur les SMA a donné lieu à plusieurs définitions de ce paradigme. Dans le cadre de ce travail, nous considérons qu'un SMA est un ensemble d'agents *autonomes* qui interagissent dans un *environnement commun* en vue de réaliser une

tâche commune.Un SMA est caractérisés par les propriétés suivantes.

l'autonomie Le SMA est autonome lorsqu'aucune entité externe ne contrôle ce système. L'autonomie du système dérive de l'autonomie de ses agents.

La distribution/la décentralisation Dans un SMA, les données, que ce soit relatives à l'environnement ou relatives aux compétences des agents, sont distribuées entre les différents agents. En plus, dans un SMA il n'existe aucune entité centrale qui peut superviser le système. Le contrôle est décentralisé.

la synchronisation/le parallélisme Dans un SMA, les agents s'exécutent de manière asynchrone et parallèle. L'exécution des agents est asynchrone car, chaque agent émettant une requête n'arrête pas son exécution en attendant une réponse. Les agents agissent simultanément, leur exécution est alors dite parallèle.

Système ouvert/système fermé Le SMA est qualifié d'ouvert lorsqu'il est possible d'en rajouter/retirer des agents. Dans le cas contraire, il est dit fermé. Un agent est capable de se reproduire ou de se suicider.

Système homogène/système hétérogène Le système est homogène lorsque les agents qui le composent ont le même type de perception et d'action ainsi que les mêmes compétences.

1.3.2 Agent

Jacques Ferber ([Ferber, 1999]) définit un agent comme étant une entité physique ou virtuelle, *autonome*, capable de *percevoir* de manière limitée son *environnement* et d'agir sur celui-ci selon ses propres objectifs. Un agent est capable de communiquer avec d'autres agents directement ou indirectement (via l'environnement). Il possède des compétences et offre des services.
L'autonomie est la principale propriété d'un agent. D'ailleurs, c'est la propriété qui le différencie de la notion d'objet. Un agent est autonome signifie que son comportement n'est pas régi par des commandes provenant de l'extérieur, mais par ses propres objectifs ou ses propres fonctions de satisfaction qu'il cherche à optimiser.
Un agent fonctionne selon le cycle *percevoir-décider-agir* formé des trois étapes suivantes :

- la perception : cette étape permet à l'agent d'actualiser les représentations qu'il possède sur son environnement ou d'acquérir de nouvelles informations sur celui-ci,
- la décision : durant cette étape, l'agent décide de l'action qu'il doit entreprendre en se basant sur ses représentations partielles,
- l'action : c'est l'étape durant laquelle l'agent réalise effectivement l'action choisie à l'étape de décision.

L'environnement représente un élément clé dans la définition d'un agent. En effet, d'après [Giunchiglia *et al.*, 2003] comme cité dans [Di Marzo Serugendo *et al.*, 2011], « *Without an environment, an agent is effectively useless. Cut off from the rest of its world, the agent can*

neither sense nor act ». Dans la littérature, il n'existe pas un consensus sur la définition d'un environnement [Weyns *et al.*, 2004]. Dans le cadre de ce mémoire, nous adoptons une définition générique donnée dans le premier chapitre de [Di Marzo Serugendo *et al.*, 2011]. Selon cette définition, on distingue l'environnement du système et l'environnement de l'agent.

L'environnement du système est composé de tous les éléments qui se trouvent à l'extérieur du système. Ces éléments sont dits situés (l'emplacement de chacun peut être déterminé) et peuvent être passifs ou actifs. Le système et son environnement sont fortement couplés et ont une influence sur leur comportement respectif (figure 1.3). Ainsi, nous distinguons entre l'activité du système et l'activité de l'environnement. Plus précisément, le système perçoit localement son environnement et agit dans celui-ci. Son comportement modifie l'état de l'environnement qui réagit en appliquant une pression sur le système. Pour s'adapter à ces contraintes, le système fait une nouvelle action dans l'environnement qui applique à son tour une nouvelle pression. Cette nouvelle pression peut être considérée comme une rétroaction (feedback), par le système en référence à ses précédentes actions. L'environnement dans lequel un système est plongé, exerce des contraintes à la quelles le système doit s'adapter. Ce couplage par les interactions conduit à des influences réciproques qui permettent des ajustements mutuels entre le système et son environnement.

De manière analogique, nous définissons l'environnement d'un agent comme l'ensemble des éléments qui lui sont extérieurs y compris les autres agents du système. Il existe également un couplage par les interactions entre l'agent et son environnement. En effet, l'agent perçoit son environnement de manière partielle et doit être capable d'agir dans cet environnement. Cette relation d'interdépendance entre le système et son environnement peut être

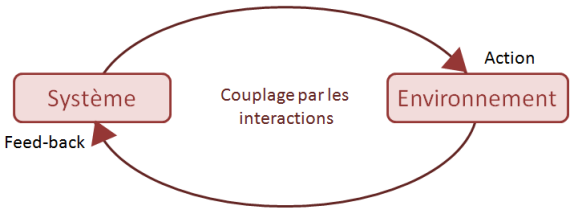

Figure 1.3 — Couplage par les interactions entre un système et son environnement (adoptée de [Di Marzo Serugendo *et al.*, 2011]

exploitée pour définir des agents pouvant s'auto-adapter aux changements de leur environnement. La *théorie des AMAS* [Glize, 2001], présentée à la section suivante, étant un exemple.

1.3.3 La théorie des AMAS : la coopération pour l'auto-organisation

La théorie des AMAS (Adaptive Multi-Agent Systems) a été développée par l'équipe SMAC pour permettre de concevoir des systèmes multi-agents capables de s'auto-organiser et faire émerger ainsi une fonction globale adaptée à l'environnement dans lequel le système est plongé. Afin de guider le processus d'auto-organisation, les comportements des agents doivent être définis en fonction d'une heuristique indépendante de la fonction glo-

bale du système. Dans la théorie des AMAS, la *coopération* représente cette heuristique pour l'auto-organisation. La validité de cette heuristique est démontrée grâce au théorème de l'adéquation fonctionnelle [Camps, 1998] énoncé ci-dessous. Ce théorème décrit la relation entre la coopération et l'adéquation fonctionnelle du système. Le système est fonctionnellement adéquat lorsque la fonction qu'il produit est jugée "bonne" soit par un observateur externe soit par l'environnement.

1.3.3.1 Théorème de l'adéquation fonctionnelle

L'interaction entre le système et son environnement joue un rôle primordial dés lors que l'on désire concevoir des systèmes auto-adaptatifs. Cette idée représente la base du théorème de l'adéquation fonctionnelle. Selon [Glize, 2001], une interaction peut être :

coopérative La transformation opérée par le système favorise celle de l'environnement et réciproquement.

antinomique La transformation opérée par le système ou l'environnement empêche l'autre de réaliser son action.

indifférente La transformation opérée par le système ou l'environnement n'a aucune influence sur l'activité de l'autre.

Un interaction est dite *non coopérative* lorsqu'elle n'est *antinomique* et *indifférente*.

Théorème de l'adéquation fonctionnelle

> *Pour tout système fonctionnellement adéquat, il existe au moins un système à milieu intérieur coopératif qui réalise une fonction équivalente dans le même environnement.*

Un système à milieu intérieur coopératif est un système dans lequel toutes ses interactions avec son environnement sont coopératives.

Conformément à ce théorème, concevoir des systèmes fonctionnellement adéquat revient à concevoir des agents qui sachent rester *coopératifs* malgré les perturbations qui peuvent se produire. Ces agents sont dits *coopératifs* et ils sont dotés d'un comportement *nominal* leur permettant d'atteindre leurs buts et d'un comportement *coopératif* leur permettant de détecter les *situations non coopératives* (SNC) et de les résoudre.

1.3.3.2 Les situations non coopératives

Le comportement d'un agent coopératif au sein d'un AMAS est orienté par son attitude coopérative. Ainsi, il doit âtre capable de détecter les SNC et de les résoudre. Ces SNC sont définies de manière générale à chaque étape du cycle de vie de l'agent.

- A l'étape de perception :

- *L'incompréhension* : l'agent n'arrive pas à comprendre le signal qu'il perçoit.
- *L'ambiguïté* : l'agent attribut plusieurs interprétations pour le signal perçu.

▷ A l'étape de décision :

- *L'incompétence* : le signal perçu n'a pas d'intérêt pour le processus de décision de l'agent.
- *L'improductivité* : l'agent n'arrive pas à décider de l'action à entreprendre.

▷ A l'étape d'action

- *La concurrence* : l'action de l'agent aura les mêmes conséquences que d'autres agents.
- *Le conflit* : l'action de l'agent sera antinomique avec les actions d'autres agents.
- *l'inutilité* : l'action de l'agent ne lui permet pas de se rapprocher de son but et ne sert pas à aider d'autres agents.

1.4 Conclusion

L'objectif de ce chapitre est de décrire le contexte dans lequel se situe nos travaux de recherche et de clarifier les notions utilisées dans la suite de ce manuscrit. Les SMA auto-adaptatifs auxquels nous nous intéressons emploient l'auto-organisation comme moyen pour s'auto-adapter appelés SMA auto-organisateurs.

Notre problématique étant autour de la vérification formelle des SMA auto-organisateurs, le chapitre suivant présente un survol sur les différents travaux traitant cette problématique.

2 État de l'art sur la vérification/validation des systèmes auto-adaptatifs

« Dans tous les arts, le plaisir croît avec la connaissance que l'on a d'eux. »
Ernest Hemingway

AVEC l'évolution croissante des systèmes informatiques se manifestant par l'apparition des systèmes à large échelle et les systèmes plongés dans des environnement extrêmement dynamiques, le développement de systèmes auto-adaptatifs est devenu une piste incontournable.

L'auto-adaptation permet au système d'ajuster son comportement en fonction des changements auxquels il peut se heurter et offre ainsi un moyen de concevoir des systèmes complexes de manière flexible et naturelle. Le développement de ces systèmes est souvent confronté à la problématique de vérification et de validation.

Dans la littérature, cette problématique a été traitée de trois manières différentes :

- à travers l'évaluation qui consiste à définir des métriques bien particulières permettant de déterminer jusqu'à quelle mesure le système assure sa fonctionnalité et s'adapte à son environnement,
- à travers la simulation qui permet d'exécuter le système selon plusieurs scénarios et d'ajuster ces paramètres,
- à travers la vérification formelle qui se base sur des méthodes formelles. Ces méthodes offrent des techniques, des langages et des outils basés sur les mathématiques pour spécifier et vérifier les systèmes.

Dans ce chapitre, nous dressons un état de l'art couvrant les différents travaux qui ont traité la vérification des systèmes auto-adaptatifs. D'abord, une revue sur les travaux por-

tant sur l'évaluation des systèmes auto-adaptatifs nous a permis de cerner les propriétés pertinentes à cette vérification. Ensuite, un panorama des travaux traitant la simulation et la vérification formelle est dressé.

2.1 Évaluation des systèmes auto-adaptatifs

L'évaluation d'un système repose sur l'identification des propriétés à évaluer et la définition des métriques permettant l'évaluation de ces propriétés. La revue de littérature que nous avons réalisée montre une diversité dans les classifications ainsi que les définitions accordées à ces propriétés. Dans cette section, nous présentons un survol des propriétés des systèmes auto-adaptatifs de manière générale ainsi que certaines métriques utilisées pour les évaluer. Nous concluons par citer les propriétés auxquelles nous nous intéressons dans le cadre de ce travail ainsi que les définitions adoptées.

2.1.1 Identification/classification des propriétés

Plusieurs classifications ont été proposées pour les propriétés des systèmes auto-adaptatifs. Dans [Kaddoum *et al.*, 2010], les propriétés ont été identifiées selon le niveau considéré dans le développement du système. Ainsi, Kaddoum et *al* distinguent des propriétés d'ordre méthodologique, celles d'ordre architectural et les propriétés relatives à l'exécution. Dans [Villegas *et al.*, 2011], les propriétés ont été classées selon qu'elles se réfèrent au gestionnaire de l'adaptation (Observateur/controleur) ou bien au système géré (système sous contrôle) (voir la section 1.1 du chapitre 1). Les principales propriétés identifiées sont citées dans la suite de cette section.

La performance du système ([Kaddoum *et al.*, 2010]) La performance du système renseigne sur sa capacité à atteindre son objectif. Elle est mesurée en calculant la variation entre le temps de réponse du système sans utiliser ses mécanismes d'auto-adaptation et son temps de réponse en utilisant ses mécanismes d'auto-adaptation pour assurer la même fonctionnalité. La performance est aussi mesurée en fonction du degré de progression du système vers son but et comment cette progression est influencée par les mécanismes d'auto-adaptation.

La robustesse ([Kaddoum *et al.*, 2010], [Serugendo, 2009], [Villegas *et al.*, 2011])
Kaddoum et *al* définissent la robustesse comme la capacité du système à maintenir son fonctionnement malgré les perturbations. Dans [Villegas *et al.*, 2011], cette propriété est liée au gestionnaire d'adaptation et elle renseigne sur sa capacité à opérer même dans des conditions imprévues à l'avance. Serugendo ([Serugendo, 2009]) donne une définition plus précise dans le cadre des systèmes auto-organisateurs en définissant quatre différents attributs pour qualifier la robustesse.

- ▷ la convergence : renseigne sur la capacité du système à atteindre l'objectif pour lequel il a été conçu,
- ▷ la stabilité : montre la capacité du système à maintenir son but une fois atteint,

- ▷ la rapidité de convergence,
- ▷ le passage à l'échelle : montre l'impact du nombre d'agent sur le système.

La résilience ([Bankes, 2010] [Serugendo, 2009] Comme définie dans [Bankes, 2010], la résilience indique la capacité du système à se remettre des perturbations extérieures lorsque la robustesse n'est plus garantie. Les mécanismes de résilience fournissent la capacité d'adaptation et permettent de progresser vers la robustesse en mode dégradé.

L'homéostasie [Kaddoum *et al.*, 2010] ou stabilité [Villegas *et al.*, 2011] L'homéostasie est la capacité du système à regagner son comportement nominal après une perturbation. Cette capacité est évaluée par la durée nécessaire pour s'auto-adapter aussi bien au niveau d'un agent qu'au niveau du système. Dans [Villegas *et al.*, 2011], la stabilité désigne la capacité du processus d'adaptation à converger vers les objectifs du contrôle.

La consistance [Villegas *et al.*, 2011] La consistance a pour but de garantir l'intégralité, en l'occurence l'intégralité structurelle du système géré après une opération d'adaptation.

Le passage à l'échelle [Villegas *et al.*, 2011] Cette propriété indique la capacité du gestionnaire d'adaptation à maintenir sa performance face à la croissance des demandes pour lesquelles il est sollicité.

L'objectif de l'identification et la définition des propriétés des systèmes auto-adaptatifs est de trouver les métriques adéquates permettant de les mesurer. Dans la section suivante, nous citons quelques une de ces métriques.

2.1.2 Les métriques pour évaluer les propriétés

Dans cette section, nous présentons quelques métriques utilisées pour évaluer les propriétés des systèmes auto-adaptatifs.

La mesure d'entropie Cette métrique prend ses origines de l'entropie de Shannon de la théorie de l'information [Shannon, 1948].

$$E = \frac{-\sum_{n}^{N}(p_n) * logp_n}{logN} \tag{2.1}$$

Dans cette équation, p_n dénote la probabilité que l'état n parmi N états soit atteint. Le quotient par $log\ N$ sert à normaliser la valeur de E dans l'intervalle $[0, 1]$. Une entropie proche de 1 indique des probabilités égales. Cette mesure est utile lorsqu'il est possible de décrire les propriétés à vérifier en termes d'états avec leurs probabilités correspondantes et lorsque la mesure d'égalité ou d'inégalité de probabilité est signifiante.

Dans [De Wolf et Holvoet, 2007], cette métrique a été utilisée pour mesurer le degré de distribution spatiale de robots dans un bâtiment contenant des objets dispersés dans plusieurs zones. Le but a été de vérifier que les robots sont capables d'atteindre toutes les zones pour collecter tous les objets. Dans ce cas, chaque zone est caractérisée par un état dans lequel elle contient un robot ou pas et la probabilité p_n représente la

probabilité de trouver un robot sur une zone.

Dans [Guerin et Kunkle, 2004], la mesure d'entropie a été utilisée pour calculer "l'ignorance" d'une fourmi se déplaçant sur une grille à la recherche de nourriture. Dans ce contexte l'ignorance représente le degré de liberté de déplacement. Ainsi, si la fourmi ne perçoit aucun gradient de phéromone, elle est complètement ignorante et son entropie vaut 1. Par contre, si la fourmi a une seule alternative pour se déplacer, son ignorance est nulle. Dans ce cas, p_n est la probabilité de déplacement vers la position n et N est le nombre total de positions vers lesquelles la fourmi est susceptible de se déplacer. L'ignorance de la colonie de fourmis est mesurée en faisant la moyenne des ignorances de toutes les fourmis.

L'exposant de Lyapunov Cette métrique est originaire de la théorie du chaos [cha] et mesure le degré de sensibilité du comportement du système aux changements effectués dans les conditions initiales de son exécution [De Wolf et Holvoet, 2007]. Elle sert à renseigner sur la stabilité et la robustesse du système.

La moyenne En faisant la moyenne de la durée nécessaire au système de s'auto-adapter, il est possible de mesurer la robustesse du gestionnaire d'adaptation [Villegas *et al.*, 2011].

La distance Cette métrique est utilisée dans [Kaddoum *et al.*, 2010] pour mesurer la variation entre l'état du système avant et après une perturbation et déterminer sa robustesse.

2.1.3 Bilan sur l'évaluation à l'aide des métriques

A partir de ces travaux sur l'évaluation des systèmes auto-adaptatifs, nous avons pu identifier les propriétés pertinentes de ces systèmes. Dans le cadre de ce travail nous considérons deux propriétés à vérifier formellement pour les SMA auto-organisateurs :

- la convergence définie par la capacité du système à atteindre un état fonctionnellement adéquat et
- la résilience qui détermine la capacité des mécanismes d'auto-organisation à ramener le système dans un état fonctionnellement adéquat après perturbations.

2.2 La simulation : un moyen incontournable pour la validation

Cette section est composée de deux parties. La première montre l'utilisation de la simulation "classique" (parfois couplée avec l'analyse numérique) en vue de valider les SMA conçus. La deuxième partie est consacrée aux travaux qui appliquent la simulation stochastique qui se base sur les statistiques afin de prédire le comportement du système réalisé.

2.2.1 Simulation pour les SMA

Durant la dernière décennie, la simulation des systèmes complexes est devenue l'application par excellence des SMA ([Niwa *et al.*, 2014], [Picascia *et al.*, 2014]). Ceci s'est traduit par l'émergence d'une panoplie de simulateurs de SMA (Netlogo [1], Repast [2], GAMA [3] et SeSAm [4]), ainsi que par l'intégration de la simulation dans le processus de développement de SMA comme dans *PASSIM* [Cossentino *et al.*, 2008], *INGENIAS* [Gómez-Sanz *et al.*, 2010] et *easyABMS* [Garro et Russo, 2010]. Dans cette section, nous nous intéressons à la simulation en tant qu'un moyen incontournable (comme indiqué dans [Cossentino *et al.*, 2010] [De Wolf *et al.*, 2005]) pour valider les SMA conçus. La validation par simulation consiste à évaluer plusieurs stratégies du fonctionnement du système en faisant varier ses paramètres pour choisir le meilleur paramétrage [Shannon, 1975].

Orfanus ([Orfanus *et al.*, 2011]) propose un processus basé sur la simulation pour valider les systèmes embarqués massivement distribués (composé de centaines voire de milliers d'éléments autonomes) ayant un comportement émergent . Ce processus s'appuie sur deux modèles : le modèle du niveau microscopique et celui du niveau macroscopique. Le premier modèle décrit les activités des entités qui composent le système, ainsi que les informations locales permettant de déclencher ces activités. Le second décrit l'état global du système par un ensemble de variables macroscopiques. Ces variables contiennent des agrégats de valeurs obtenus à partir des variables des éléments du niveau micro. La simulation est appliquée pour la réalisation de la transition entre le niveau micro et le niveau macro. Chaque simulation prend en entrée le modèle du niveau micro et un scénario de simulation composé de trois aspects :

- l'environnement et les changements qui peuvent y avoir lieu,
- la dynamique des éléments dans l'environnement,
- le déploiement initial du système.

La validation du modèle du niveau micro se fait en comparant le résultat de la simulation par rapport au comportement macro désiré et retourne une "déviation" qui indique jusqu'à quelle mesure le comportement macro obtenu par simulation est conforme au comportement macro désiré. Cette déviation est utilisée pour ajuster le comportement des agents au niveau micro.
L'avantage de la méthode proposée est que la simulation est réalisée sur des modèles microscopiques de haut niveau d'abstraction ce qui permet d'identifier les activités locales et les mécanismes de coordination distribués appropriés avant d'intégrer des détails de l'implémentation. Son inconvénient majeur réside dans le temps nécessaire pour la réalisation de ces simulations surtout lorsque le système est formé d'un nombre très importants d'agent ce qui est le cas des systèmes embarqués massivement distribués.

1. http ://ccl.northwestern.edu/netlogo/
2. http ://repast.sourceforge.net/repast_3
3. https ://code.google.com/p/gama-platform/wiki/GAMA ?tm=6)
4. http ://www.simsesam.de/

Pour pallier ce problème, Tom de Wolf [De Wolf *et al.*, 2005] suggère de coupler la simulation avec des algorithmes d'analyse numérique dans une même approche. Habituellement, les analyses numériques sont appliquées aux modèles $equation - based$. Le comportement global du système est décrit par une équation d'évolution macroscopique et l'algorithme d'analyse numérique permet de quantifier les propriétés souhaitées. Devant l'impossibilité d'avoir une description de l'équation macroscopique pour un système dynamique et complexe à partir des modèles des entités autonomes le composant, Tom De Wolf propose d'utiliser une approche d'analyse $equation\text{-}free$ [Kevrekidis *et al.*, 2004], en remplaçant l'équation macroscopique (qui n'est pas connue) par la simulation des modèles des agents.

L'analyse du comportement collectif des agents (le comportement macroscopique) se base sur la sélection des propriétés macroscopiques pertinentes quantifiées avec des variables mesurables appelées des "*variables macroscopiques*". Cette analyse permet de fournir des "*garanties macroscopiques*" étant donné un ensemble de conditions initiales bien déterminées. Ces garanties représentent le moyen pour vérifier si le système évolue conformément à la *tendance* requise. Une *tendance* est définie comme l'évolution du comportement macroscopique du système obtenue en considérant la moyenne de plusieurs exécutions. L'analyse complète du système au niveau macroscopique est obtenue via plusieurs *garanties macroscopiques*.

Contrairement aux approches de simulation classique, qui se contentent d'une simple observation des résultats, l'utilisation de l'algorithme d'analyse numérique permet d'orienter le processus de simulation vers un but bien déterminé. Le but de l'analyse peut être par exemple, la recherche d'un éventuel état durant lequel le comportement du système se stabilise (le but de l'algorithme *projective integration algorithm*) ou prévoir (extrapoler) le comportement du système aussi loin que possible dans le temps (le but de l'algorithme de Newton). L'avantage dans ce cas est que le temps nécessaire pour la simulation sera réduit considérablement.
L'approche proposée peut se résumer par les étapes suivantes devant être répétées jusqu'à ce que l'algorithme d'analyse atteigne son but. :

- Initialisation : les variables macroscopiques à étudier sont initialisées. Un certain nombre de simulations du niveau micro sont initialisées conformément aux valeurs données pour les variables au niveau macro.
- Simulation : des simulations sont réalisées durant une certaine durée de temps en se basant sur le code de simulation des éléments du système.
- Mesure : les valeurs des variables macroscopiques sont évaluées et fournies à l'algorithme d'analyse pour qu'il génère les initialisations de la prochaine simulation.

L'utilisation de cette approche requiert certaines informations identifiées au préalable telles que les propriétés macroscopiques à vérifier, les variables macroscopiques qui permettent de les mesurer ou qui ont une influence sur leur évolution ainsi que les correspondances

entre chaque variable macroscopique et l'ensemble des variables microscopiques qui l'influencent.

Il est nécessaire également de choisir l'opérateur de mesure qui permet d'évaluer les variables macroscopiques à partir des variables microscopiques au cours d'une simulation, l'opérateur d'initialisation qui initialise les variables microscopiques conformément aux variables macroscopiques fixées ainsi que l'algorithme d'analyse en fonction du but de l'analyse. Ces choix ne sont pas triviaux et nécessitent une grande expertise de la part du concepteur. En effet, passer du niveau macro au niveau micro et inversement au cours des étapes d'initialisation et de mesure supposent une connaissance complète et suffisante de la relation micro-macro. L'application de l'approche *equation-free* est soumise à une autre condition pour garantir des résultats fiables ; l'évolution des états des entités du système doit être plus rapide que celle du comportement du système à l'échelle macroscopique.

Dans [Bernon *et al.*, 2006], une phase de simulation a été intégrée à la méthode *ADELFE*, conçue pour le développement des SMA adaptatifs appelés aussi *AMAS*. Avec les *AMAS*, l'adaptation se fait par auto-organisation et l'émergence de la fonctionnalité adéquate est assurée en dotant les agents par une attitude coopérative (comme définie dans [Capera *et al.*, 2003]). Ainsi, une bonne partie de la charge du concepteur consiste à énumérer toutes les Situations Non Coopératives (SNC), auxquelles un agent peut être confronté, et à décrire le comportement lui permettant de les neutraliser. Le but de la simulation dans [Bernon *et al.*, 2006] a été de découvrir ces SNC lors de la conception des agents.

Dans le but d'automatiser la simulation et de réduire la charge du concepteur, les travaux de S. Lemouzy ont été proposés dans [Lemouzy *et al.*, 2007]. Les agents ont la capacité d'auto-ajuster leurs comportements en faisant de leurs règles comportementales des agents capables de s'auto-organiser. Ces deux approches ont été appliquées pour simuler des agents utilisant l'environnement pour communiquer. Les simulations ont été réalisée à l'aide de la plateforme de simulation *SeSAm* [5].

Pour bien guider les simulations lors du développement du comportement coopératif des agents d'un *AMAS*, N. Bonjean a proposé, dans [Bonjean *et al.*, 2010], des heuristiques permettant de mieux ajuster les paramètres des agents lorsque le comportement global du système n'est pas conforme à celui désiré. Ces heuristiques peuvent être résumées par les étapes suivantes :

- ▷ établir les liens entre les paramètres et les actions,
- ▷ déterminer la manière avec laquelle l'influence des actions sur les paramètres se propage,
- ▷ ajuster les paramètres,
- ▷ vérifier l'impact de l'ajustement réalisé sur le comportement du système,

5. http ://www.simsesam.de/

- ▷ réaliser d'autres ajustement (tout en préservant l'effet des précédents) si le comportement global n'est pas toujours celui désiré,

- ▷ observer le comportement collectif obtenu.

Récemment, la processus de la méthode ADELFE a été enrichi par une étape basée sur le *living design*. Cette étape se base sur le modèle *S-DLCAM* (Self-Design and Learning Cooperative Agent Model) [Mefteh *et al.*, 2013] qui étend celui de l'agent coopératif (proposé pour concevoir des *AMAS* [Georgé *et al.*, 2003]) par des mécanismes d'auto-conception (*self-design*) et d'apprentissage. Avec ce nouveau modèle, l'agent possède des compétences lui permettant de détecter lui même les SNC, de les corriger et de les éviter une prochaine fois. Ce modèle a été également développé au moyen de la plateforme *SeSAm*.

En conclusion, la simulation permet de visualiser le comportement du système sous plusieurs scénarios pour ajuster au mieux ses paramètres ou bien pour aider à prévoir les perturbations auxquelles il peut faire face en vue de les anticiper et empêcher leur production ou apporter les corrections nécessaires dans le cas où elles se produisent (tel est le cas des SNC avec les *AMAS*).

2.2.2 Simulation stochastique

La simulation stochastique, connue aussi sous le nom de la simulation par la méthode de Monte Carlo, permet de prédire le comportement d'un système dont l'évolution peut contenir des incertitudes. Elle se base sur la génération d'événements de manière aléatoire et utilise les inférences statistiques pour l'estimation de l'intervalle de confiance de la performance du système modélisé.

La simulation stochastique a été utilisée dans divers domaines scientifiques : en chimie pour la modélisation des réactions chimiques ([Erban *et al.*, 2007], [Scappin et Canu, 2001]), en finance pour la modélisation des modèles financiers [Huynh *et al.*, 2009] ou encore en biologie moléculaire pour l'analyse et la compréhension des interactions complexes dans le système cellulaire [Meng *et al.*, 2004], [Regev *et al.*, 2001].

Elle a servi également pour les informaticiens comme un moyen pour modéliser et analyser les systèmes auto-adaptatifs. Cette utilisation s'est manifestée par la création de simulateurs stochastiques comme SPiM [Phillips et Cardelli, 2007], PRISM[6] [Hinton *et al.*, 2006], [Kwiatkowska *et al.*, 2011] (ces outils sont basés sur l'algorithme de Gillespie [Gillespie, 1977] de simulation stochastique) ainsi que la réalisation de plusieurs extensions stochastiques sur les algèbres de processus et notamment le langage π-*Calculus* [Milner *et al.*, 1992]. Parmi ces extensions, on cite *Stochastic* π-*Calculus* [Priami, 1995] et *Performance Evaluation Process Algebra* (*PEPA*) [Hillston, 2005c], [Hillston, 2005b].

6. http ://www.prismmodelchecker.org/

Le langage π-*Calculus* a été proposé par Robin Milner au début des années 90s [Milner *et al.*, 1992] comme une extension de l'algèbre de processus *CCS* pour la modélisation de systèmes mobiles et le raisonnement sur leurs comportements. Deux concepts clés sont utilisés dans ce formalisme ; *nom* et *processus*. Un nom est défini comme un canal ou une valeur transmise via un canal. Les processus servent à modéliser les agents du système. L'évolution du système est décrite par un ensemble de règles de transition ayant la forme $P \xrightarrow{\alpha} P'$ et signifiant que le processus P évolue vers le processus P' après avoir exécuté l'action α. Cette action peut être une action d'envoi ou de réception d'un nom sur un canal ou encore l'action *silent* qui représente une communication interne au processus.

Afin de prendre en compte des aspects quantitatifs permettant de mesurer la performance d'un système, Priami introduit une extension stochastique au langage π-*Calculus* appelée π-*Calculus stochastique* [Priami, 1995]. Avec cette extension, chaque transition est enrichie par un taux (r) représentant le paramètre d'une distribution exponentielle. Cette distribution caractérise la durée de temps nécessaire pour qu'une transition s'active. Ainsi, pour un taux r, la probabilité qu'une transition s'active dans t unités de temps est égale à $1 - e^{(-r*t)}$.

Parmi les travaux qui ont utilisé la simulation stochastique et le langage π-*Calculus stochastique*, on cite celui de Gardelli dans [Gardelli *et al.*, 2006] et [Gardelli *et al.*, 2005] Cette technique de simulation a été appliquée au problème de découverte d'intrusions qui focalise sur la détection des anomalies dans le comportement des agents. Le système est basé sur l'architecture *TuCSon* ([Omicini et Zambonelli, 1999]). Selon cette architecture, un SMA est composé d'agents, d'artéfacts de coordination représentant les ressources de l'environnement et de contextes de coordination dont le rôle est de spécifier les politiques d'accès des agents aux ressources. Le système est protégé contre les intrusions par une couche permettant de détecter les agents malicieux et inspirée du système immunitaire humain. Cette couche est formée par des agents lymphocytes qui se chargent de détecter les comportements suspects des autres agents et qui doivent, par auto-organisation, s'adapter aux changements de leur environnement, soit en augmentant leur nombre en cas d'attaque soit en libérant des ressources non utilisées.

Des simulations ont été effectuées à l'aide de l'outil *SPiM* (Stochastic Pi-Machine) afin d'évaluer la capacité du système à détecter les intrusions et éliminer les agents malicieux en variant plusieurs paramètres tels que le nombre d'agents lymphocytes (chargés de l'inspection) et la fréquence des inspections sur le comportement du système. L'évaluation du système se fait en mesurant le temps nécessaire pour la détection et l'élimination des agents malicieux. Trois scénarios ont été simulés.

Dans le premier scénario, les agents légitimes et malicieux peuvent entrer et quitter le système. Le nombre d'agents inspecteurs (ou agents lymphocytes) ainsi que la fréquence d'inspections sont fixés. La probabilité qu'un agent lymphocyte détecte l'intrusion d'un agent malicieux est aussi fixe. Cette probabilité peut être considérée comme un contrat entre

le système et les agents inspecteurs. Si un agent inspecteur ne remplit pas le contrat, il sera remplacé par le système.

Dans le deuxième scénario, les agents inspecteurs disposent d'une durée de vie limitée et ils sont classés en deux catégories A et B selon leurs performances. La catégorie A regroupe des agents plus performants que ceux de la catégorie B. Les agents qui montrent de meilleures performances seront récompensés en leur augmentant leurs durées de vie. Ainsi, quatre paramètres se rajoutent au modèle : (1) la probabilité qu'un agent appartienne à la classe A plutôt que la classe B, (2) la durée de vie des deux classes, (3) la probabilité qu'un agent de la catégorie A détecte une intrusion et (4) la probabilité qu'un agent de la catégorie B détecte une intrusion.
Dans le troisième scénario, on suppose que les comportements d'intrusion ne sont pas uniformément distribués dans l'espace des comportements possibles. Les agents les moins performants doivent alors apprendre des agents les plus performants.

Le langage *Bio-PEPA* [Ciocchetta et Hillston], basé sur le langage *PEPA*, a été proposé originalement pour modéliser les systèmes biochimiques et les systèmes biologiques mais il a été également adopté pour analyser tout système regroupant plusieurs individus et se caractérisant par une absence totale de contrôle centralisé. Ainsi, ce formalisme a servi dans [Massink *et al.*, 2013] pour l'analyse des systèmes robotiques à essaims (swarm robotics systems).

Avec *Bio-PEPA*, les processus représentent des groupes d'entités ayant un comportement similaire appelés *Populations* ou *Espèce*. Pour chaque espèce, on définit les actions qu'elle peut accomplir en précisant la manière avec laquelle chaque action affecte le nombre d'entités dans une population (soit une augmentation soit une diminution). La coopération entre deux populations se traduit par une composition parallèle en synchronisant les actions partagées. Chaque action, définie pour une population, est caractérisée par une durée modélisée par une variable aléatoire continue suivant une distribution exponentielle négative et un taux exprimé en fonction des entités qui forment la population. Dans une spécification en *Bio-PEPA*, il est possible d'exprimer une représentation symbolique de l'espace physique. Ainsi, en plus de son taux, une action est définie en fonction d'un endroit.

Dans [Massink *et al.*, 2013], *Bio-PEPA* a été utilisé pour modéliser et analyser un groupe de robots chargé de collecter un ensemble d'objets et les transporter d'une localisation source vers une localisation cible par le chemin le plus court. Ces deux localisations sont reliées par deux chemins de longueurs différentes. Pour emporter les objets, les robots doivent se mettre en trois et coopérer ensemble. Avant d'emprunter un chemin, tous les robots d'un groupe donné votent leurs choix et le chemin préféré par la majorité sera choisi. Ce vote se fait uniquement à l'emplacement d'origine, ceci signifie que pour le retour les robots vont emprunter le même chemin. La spécification de ce système à l'aide de *Bio-PEPA* définit huit localisations parmi lesquelles, on cite les localisations origine et cible et deux emplacements où les robots effectuent leurs choix. Cette spécification comporte également

un ensemble de populations dont la population des robots empruntant le chemin le plus court, la population des robots empruntant le chemin le plus long et les populations des robots partant d'une localisation vers une autre.

L'analyse de cette spécification a été réalisée à l'aide de la simulation stochastique pour visualiser l'évolution du nombre moyen de groupes de robots actifs dans le temps.

2.2.3 Bilan sur la simulation

La simulation représente sans doute le moyen le plus répandu pour la validation des systèmes auto-adaptatifs vu la complexité des interactions entre les entités du système. Pendant plus de deux décennies, le domaine de la simulation et celui des SMA ont été combinés dans un axe de recherche très actif [Uhrmacher et Weyns, 2009]. Combiner la simulation avec des algorithmes numériques ou utiliser la simulation stochastique a permis d'améliorer le temps nécessaire pour les simulations des systèmes de grande taille. Cependant, le fait se s'appuyer sur des résultats expérimentaux ne fournit pas des garanties formelles que la solution qui émerge des interactions des différentes entités est la fonction souhaitée du système. Les méthodes formelles, que nous détaillons dans la prochaine section, sont capables de donner de telles garanties.

2.3 Application des techniques formelles

Dans cette section, nous présentons les différents travaux basés sur les techniques formelles pour la vérification de systèmes auto-adaptatifs. Il existe principalement deux techniques de vérification formelles : le *model-cheking* et la *preuve par théorème*.

Le model-cheking permet de vérifier si un modèle du système satisfait une spécification formulée généralement avec la logique temporelle. Un *model-checker* est un outils qui prend en entrée un automate à états finis modélisant le système plus une propriété temporelle et retourne en résultat une valeur booléenne. Si le résultat est "faux", le système ne satisfait pas la propriété et le "*model checker*" renvoie un contre exemple indiquant un état du système ne satisfaisant pas la propriété.

La technique du model-checking offre un moyen automatique pour la vérification. Cependant elle exige qu'on fournisse un modèle à états finis, il faut ainsi procéder à une abstraction du système (lorsqu'il ne vérifie pas cette contrainte) pour pouvoir l'utiliser. Un autre inconvénient de cette technique est l'explosion combinatoire qui se manifeste lorsque le système considéré est formé d'un nombre important d'états.

La preuve par théorème est une technique où le système et ses propriétés désirées sont exprimés par des formules dans une certaine logique mathématique. La logique est présenté par un système formel, qui définit un ensemble d'axiomes et un ensemble de règles d'inférence. Cette technique est un processus permettant de prouver une propriété

à partir des axiomes du système et en se servant des axiomes, des règles d'inférence et des lemmes intermédiaires dérivés. Le "*theorem prover*" est un outils qui apporte une assistance pour construire des preuves. L'avantage de la preuve par théorème par rapport au model-checking est qu'elle permet de vérifier directement des systèmes dont l'espace des états est infini.

Dans cette section, nous commençons par la présentation des travaux qui ont utilisé la technique de la preuve par théorème. Nous citons ensuite ceux qui ont appliqué la technique du model-checking. Les travaux cités dans les deux parties suivantes sont classés selon le formalisme de modélisation utilisé. Nous concluons en énonçant nos choix en terme de langage et technique formels.

2.3.1 La preuve par théorème

2.3.1.1 Les grammaires de graphe

Ehrig et al [Ehrig *et al.*, 2010] proposent un cadre théorique basé sur les grammaires de graphes et les transformations de graphes (Algebraic Graph Transformation) [Ehrig *et al.*, 2006] pour la spécification, l'analyse et la vérification formelle des systèmes auto-réparateurs (self-healing systems) [Kephart et Chess, 2003; Rodosek *et al.*, 2009]. Un système auto-réparateur (SHS) est défini comme suit :

$$SHS = (GG, C_{sys}) \tag{2.2}$$

Dans cette définition, C_{sys} est un ensemble de contraintes de graphes typés regroupant les contraintes de consistance ($C_{consist}$) ainsi que les contraintes de défaillances (C_{fail}). GG est une grammaire de graphe typée définie comme suit :

$$GG = (TG, G_{init}, Rules) \tag{2.3}$$

Dans cette définition, TG est un graphe typé modélisant les configurations du système (les états du systèmes), G_{init} est la configuration initiale et $Rules$ est un ensemble de règles de transformation de graphes modélisant la dynamique (le comportement) du système. Ces règles peuvent être de trois types : des règles normales, des règles de l'environnement et des règles de réparation. Les règles normales définissent le comportement normal du système. Les règles de l'environnement modélisent toutes les défaillances possibles. Des règles de réparation sont définies pour chaque défaillance.

En se basant sur ces définitions, une formalisation des différents états d'un système auto-réparateur a été donnée. Ainsi, le système SHS (équation 2.2) peut se trouver dans un état parmi les trois états suivants atteints par l'application des règles de transformations définies : consistant, défaillant ou normal. Le système est dans un état consistant lorsqu'il vérifie les contraintes de consistance ($C_{consist}$). Il est dans un état de défaillance lorsqu'il

satisfait certaines contraintes de défaillance parmi l'ensemble C_{fail}. Le système est dans un état normal lorsqu'il ne satisfait aucune contrainte de défaillance.

La formalisation de ces états a permis de spécifier de manière formelle des propriétés de consistance et des propriétés d'auto-réparation des systèmes auto-réparateurs. Un système SHS est dit consistant lorsque tous ses états accessibles sont consistants. SHS est dit *à état normal consistant* si son état initial est normal et l'application de toutes les règles normales préserve un état normal. Du point de vue auto-réparation, le système peut être soit auto-réparateur soit fortement auto-réparateur. Il est *auto-réparateur* si chaque défaillance peut être réparée par l'application d'un ensemble de règles de réparation, le ramenant ainsi à un état normal. Le système est dit *fortement auto-réparateur* s'il est auto-réparateur et en plus il est capable de revenir à un état normal après une défaillance par l'application de ses règles normales sans faire recours aux règles de réparation.

Dans ce cadre théorique, les propriétés de non blocage et de vivacité ont été formalisées. Le *non blocage* signifie que le système ne peut pas se bloquer dans aucun des états atteints. La *vivacité*, qualifiée de "forte" et appelée aussi "forte cyclicité", signifie que tout état accessible peut être souvent atteint. La preuve formelle de ces propriétés est assurée par l'outil *AGG* (*Attributed Graph Grammars*).

La formalisation et la preuve de l'adaptabilité ont bien été prises en compte à l'aide des grammaires de graphes. Cependant, pour appliquer le modèle formel proposé sur une étude de cas (système de feu de circulation), il a fallu l'instancier en énumérant les différents éléments du système et décrire toutes les configurations possibles. Dans le cas de systèmes composés d'un grand nombre d'entités, cette instanciation ainsi que la prévision de toutes les configurations ne sont pas possibles. Nous avons constaté une autre limite de ces travaux, il s'agit de l'emploi d'une formalisation et d'un raisonnement global sur le système. En effet, la formalisation proposée concerne le comportement du système dans sa globalité et non en considérant le comportement des entités qui le composent. Nous jugeons qu'une formalisation modulaire est nécessaire dans le cas des SMA auto-organisateurs. Les travaux cités dans la suite adoptent cette formalisation modulaire.

2.3.1.2 Paradigme *rely/guarantees*

Dans [Nafz *et al.*, 2013], les auteurs considèrent les systèmes "self-*" dont l'adaptation se fait par le changement des configurations de leurs composants. L'approche de spécification formelle proposée se base essentiellement sur l'intégration de deux éléments : l'architecture *Observateur/Contrôleur* et l'approche *Restaurer l'invariant*. L'architecture *Observateur/Contrôleur*, comme indiqué dans [Richter *et al.*, 2006] se base sur la séparation de la partie fonctionnelle du système de sa partie self-*. Cette dernière est composée d'un observateur dont le rôle est de superviser le fonctionnement du système et d'un contrôleur qui se charge de son adaptation. Quant à l'approche *Restaurer l'invariant*, son idée principale est de restreindre le comportement du système par la spécification d'un invariant –une formule de la logique des prédicats– décrivant le comportement correct et devant être évalué à chaque

état du système. Ainsi, l'objectif du système est de maintenir cet invariant et de rester dans un état fonctionnel. En cas de violation de l'invariant, l'observateur notifie le contrôleur qui fait basculer le système vers un état de reconfiguration. Le contrôleur se charge alors de calculer une nouvelle configuration qui doit restaurer l'invariant. Après la reconfiguration, le système assure à nouveau un comportement correct.
De manière plus formelle, le système $self^*SYS$ est défini comme indiqué ci-dessous. L'opérateur $\|$ désigne une exécution parallèle.

$$self^*SYS = o/c \| SYS_{func} \tag{2.4}$$

Dans cette définition, o/c représente la partie *Observateur/Contrôleur* du système et SYS_{func} dénote la partie fonctionnelle composée d'un ensemble de n agents s'exécutant en parallèle et définie formellement comme suit :

$$SYS_{func} = Ag_1||...||Ag_n$$

Une exécution du système est vue comme une séquence d'états appelée *trace*. Un état étant défini par la valuation de l'ensemble des variables V du système. Une étape dans la trace est formée de deux transitions : une transition système au cours de laquelle un composant du système fait une action et une transition de l'environnement au cours de laquelle se passent les changements de l'environnement. Formellement, une transition du système est modélisée par une formule de la logique des prédicats décrivant la relation entre l'état du système avant et après l'exécution d'une action d'un agent. Une transition de l'environnement est aussi une formule de la logique des prédicats décrivant le changement de l'état du système avant et après cette transition.

La spécification du comportement de chaque agent est basée sur le paradigme *hypothèse/garanties* (*rely/guarantees*) [Jones, 1983]. Avec ce paradigme, les garanties (le comportement) qu'un agent peut fournir sont modélisées en fonction de ses hypothèses sur le comportement de son environnement. L'environnement du point de vue d'un agent englobe aussi les autres agents. Ainsi le comportement d'un agent Ag_i est défini formellement par la formule suivante :

$$R_i(V', V'') \xrightarrow{+} G_i(V, V') \tag{2.5}$$

De manière informelle, cette formule indique que l'agent Ag_i garantit le comportement $G_i(V, V')$ tant que l'environnement change conformément à la propriété $R_i(V', V'')$. $G_i(V, V')$ est une formule de la logique des prédicats modélisant une transition du système suite à une action de l'agent Ag_i. V représente l'état du système avant cette transition. V' représente le nouvel état du système après la transition. $R_i(V', V'')$ est aussi une formule de la logique des prédicats permettant de spécifier une transition de l'environnement. V' est l'état du système après la transition système et avant la transition de l'environnement. V'' est l'état du système après la transition de l'environnement.

Le paradigme *hypothèse/garanties* a été utilisé également pour modéliser le comportement de la partie o/c du système. Ainsi, la définition $self^*SYS$ donnée dans la formule (2.4) peut être reformulée comme suit :

$$self^*SYS = R_{o/c}(V', V'') \xrightarrow{+} G_{o/c}(V, V') \| R_{sys}(V', V'') \xrightarrow{+} G_{sys}(V, V') \tag{2.6}$$

avec $R_{o/c}(V', V'') \xrightarrow{+} G_{o/c}(V, V'$ modélise la partie "self-*" du système et $R_{sys}(V', V'') \xrightarrow{+} G_sys(V, V')$ est définie par la formule suivante :

$$R_1(V', V'') \xrightarrow{+} G_1(V, V') \| ... \| R_n(V', V'') \xrightarrow{+} G_n(V, V') \tag{2.7}$$

Afin de mener un raisonnement formel sur le système dans sa globalité, les auteurs proposent d'utiliser le théorème de compositionnalité développé dans [Bäumler *et al.*, 2011] et prouvé à l'aide du prouveur *KIV* [Balser *et al.*, 1998]. Ce théorème donne les relations nécessaires entre les comportements des différents composants du système et les obligations de preuves garantissant le comportement global. La principale obligation est de prouver que chaque composant maintient ses garanties tant que les hypothèses qui lui sont associées sont vérifiées. Les autres obligations assurent la compatibilité et la consistance entre les comportements des composants, tel que par exemple, garantir qu'un composant ne va pas violer les hypothèses d'un autre composant.

En plus de ces obligations de preuve, il faut s'assurer que les implémentations particulières des agents et de la partie self-* du système sont conformes aux spécifications données par leurs formules *hypothèse/garanties* respectives.

2.3.1.3 SLABS

Dans [Zhu, 2005], la spécification formelle d'agents a été réalisée à l'aide du langage *SLABS* (Specification Language for Agent-Based Systems) [Zhu, 2003]. Ce langage offre une structuration modulaire pour spécifier formellement des SMA. Il permet de définir le comportement des agents à l'aide d'un mécanisme de description basé sur la notion de *Caste*.

La notion de caste peut être assimilée à celle de classe dans la modélisation basée objet. Un caste sert à décrire les caractéristiques structurelles et comportementales d'un type d'agents. Il est composé par quatre parties : *ENVIRONMEMNT*, *VAR*, *ACTION* et *RULE*, comme le montre la figure 2.1.

- *ENVIRONMENT* : un agent est capable d'observer les actions et les états externes d'autres agents. Ces agents constituent son environnement.
- *VAR* : l'ensemble des variables décrivant l'état d'un agent. Les variables peuvent être internes (marquées par un astérisque) ou visibles pour l'environnement de l'agent.
- *ACTION* : les différentes actions qu'un agent peut accomplir. Lorsqu'une action n'est pas visible pour l'environnement de l'agent, elle est dite *interne* et se marque par un astérisque.

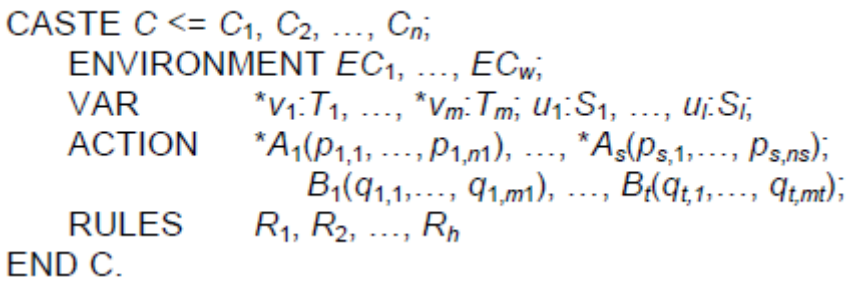

Figure 2.1 — Description d'un *Caste* ([Zhu, 2005])

- *RULE* : l'ensemble des règles comportementales permettant à un agent de décider de l'action à exécuter. Une règle comportementale est définie en fonction d'un *patron* et d'un *scénario*. Le patron décrit le comportement d'un agent par une séquence d'états et d'actions observables. Il sert à formaliser les états et les actions antérieurs d'un agent. Le scénario décrit la situation globale de l'environnement reflétée par les états de l'ensemble des agents du système ainsi que leurs actions.

Pour assurer un raisonnement formel sur le comportement dynamique du SMA, le calcul des scénarios (scénario calcul) a été défini sur la base du langage $SLABS$. Le calcul des scénarios définit trois relations entre les scénarios :

- la relation d'inclusion : un scénario Sc_1 inclut un scénario Sc_2 signifie que si le système est dans le scénario Sc_1, il est alors dans le scénario Sc_2 aussi. De manière formelle, on considère deux scénarios Sc_1 et Sc_2 d'un SMA M. Un scénario Sc_1 inclut Sc_2 dans M, noté $M \models Sc_1 \Rightarrow Sc_2$ si et seulement si pour toute exécution r et pour tout instant $t \in T$, $r \downarrow t \models Sc_1$ implique que $r \downarrow t \models Sc_2$.
  T est un sous ensemble de l'ensemble des réels et joue le rôle d'un indice temps. La notation $r \downarrow t \models Sc$ signifie que le système est dans le scénario Sc à l'instant t pour l'exécution r.

- la relation de transition : le scénario Sc_1 transite au scénario Sc_2 signifie que le système peut passer d'un état du scénario Sc_1 vers un état du scénario Sc_2. Formellement, un scénario Sc_1 transite à un scénario Sc_2 dans M, noté $M \models Sc_1 \rightarrow Sc_2$ si et seulement si il existe une exécution r de M et deux instants $t_1 < t_2 \in T$ tel que $r \downarrow t_1 \models Sc_1$ et $r \downarrow t_2 \models Sc_2$.

- la relation d'orthogonalité : le scénario Sc_1 est orthogonal au scénario Sc_2 signifie que le système ne peut jamais être dans les deux scénarios simultanément. La relation d'orthogonalité est utilisée pour montrer la consistance du modèle en $SLABS$.

Ces relations d'inclusion et de transition ont servi pour montrer trois propriétés sur les scénarios et par conséquent sur l'état global du SMA : l'atteignabilité d'un scénario ("reachability"), la stabilité et la convergence.

L'atteignabilité d'un scénario particulier *Sc* signifie que le système sera à un instant donné durant son exécution dans ce scénario. Cette propriété est exprimée formellement par la définition suivante :

Définition 1. *Un système M atteint toujours un scénario Sc, noté* $M \rightarrowtail Sc$*, si pour toute exécution r, il existe un instant t tel que* $r \downarrow t \models Sc$.

La stabilité signifie qu'une fois le système atteint un scénario, il y reste. Formellement, cette propriété est définie par la définition suivante :

Définition 2. *Un scénario Sc est stable dans un système M, noté M@Sc, si pour toute exécution r et pour tout instant t avec* $t \in T$, $r \downarrow t \models Sc \Rightarrow \forall t' > t \in T.(r \downarrow t' \models Sc)$.

La convergence vers un scénario *Sc* indique la capacité du système à revenir sur le scénario *Sc*. Formellement, cette propriété est définie comme suit :

Définition 3. *Un système M converge toujours vers un scénario Sc, noté* $M \looparrowright Sc$*, si pour toute exécution r, il existe un instant t tel que* $\forall t' \in T.(t' > t \Rightarrow r \downarrow t' \models Sc)$.

Ces travaux de [Zhu, 2005] sont particulièrement intéressants car ils s'inscrivent dans le cadre des systèmes à fonctionnalités émergentes et ont réussi à prouver des propriétés pertinentes de ces systèmes à savoir la stabilité, l'atteignabilité et la convergence[7]. Leur inconvénient majeur réside dans $SLABS$, le langage formel utilisé. Ce langage n'est pas supporté par des outils et la formalisation ainsi que la preuve doivent être faites à la main.

2.3.2 La technique du model-checking

Dans cette partie, nous citons des travaux utilisant le model-checking comme technique de vérification formelle pour les systèmes auto-adaptatifs. Ces travaux sont classés selon les formalismes utilisés.

2.3.2.1 Systèmes synchrones adaptatifs

Dans ([Adler *et al.*, 2007]), Adler et al proposent une formalisation des systèmes embarqués adaptatifs basée sur les systèmes synchrones adaptatifs SAS (Synchronous Adaptive Systems) et permettant la modélisation modulaire et la vérification du comportement d'adaptation. Un système, comme le montre la figure 2.2, consiste en un ensemble de modules *M* opérant de manière synchrone. Chaque module *m* possède un ensemble de configurations prédéfinies correspondant aux différentes variantes comportementales du module. Afin de spécifier le comportement d'adaptation dans un module de façon modulaire et séparée de sa fonctionnalité, chaque module comporte deux ensembles disjoints de variables d'entrée, de variables locales et de variables de sortie :

- ▷ les variables fonctionnelles *var* avec leurs valeurs initiales *init* et

7. La définition de convergence est différente de celle donnée dans [Serugendo, 2009]

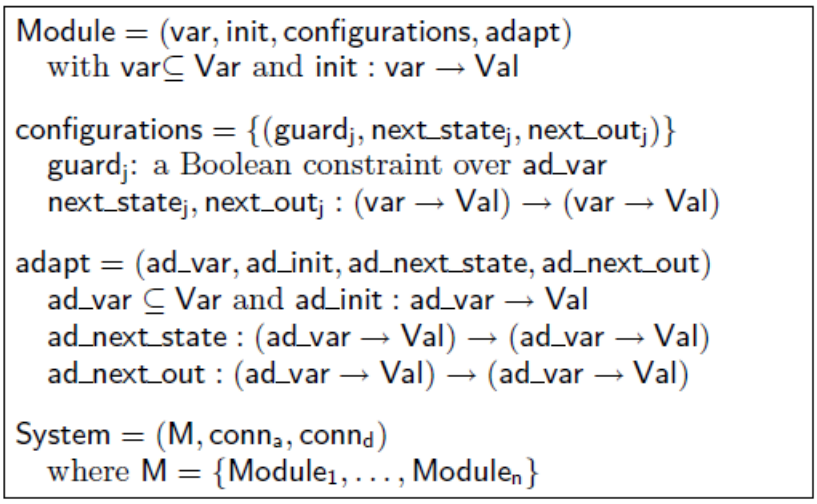

Figure 2.2 — Description d'un système SAS ([Schaefer et Poetzsch-Heffter, 2009])

- ▷ les variables d'adaptation *ad_var* avec leurs valeurs initiales *ad_init* encapsulées dans un aspect d'adaptation *adapt*.

La condition d'activation d'une configuration est formulée par la garde $guard_j$ et dépend uniquement des variables d'adaptation. Les fonctions de transition $next\_state_j$ et $next\_out_j$ évaluent respectivement les variables locales fonctionnelles et les variables fonctionnelles de sortie. Les variables d'adaptation de sortie sont connectées aux variables d'adaptation d'entrée par les relations d'adaptation $conn_a$. Ces relations transmettent des informations concernant la configuration choisie et permettent ainsi aux modules connectés de réagir aux adaptations des autres. Les variables fonctionnelles de sortie sont connectées aux variables fonctionnelles d'entrée par les connections fonctionnelles $conn_d$. Ces connexions fonctionnelles communiquent les valeurs fonctionnelles calculées par les modules.

L'évolution du système est décrite par un système à état/transition $T = (\sigma, I, \rightsquigarrow)$. L'ensemble des états σ est décrit par la valuation des variables des modules. I est l'évaluation des variables d'entrée et de sortie. La transition $\rightsquigarrow$ est une transition synchrone se faisant en deux étapes : premièrement, les modules lisent les informations en entrée et ensuite ils effectuent tous une transition locale de manière synchrone. Une transition locale d'un module se fait comme suit : les nouvelles valeurs des variables d'adaptation sont calculées, ensuite la configuration valide est choisie et finalement, la configuration choisie calcule le nouvel état local et la nouvelle sortie.

Cette formalisation avec le modèle SAS a permis la vérification des propriétés suivantes :

- ▷ des propriétés génériques liées à l'application,
- ▷ des propriétés de non blocage qui permettent de vérifier :
  - qu'aucun module ne reste bloqué dans la configuration par défaut,

  - que chaque module peut atteindre toutes les configurations à n'importe quel instant.

- ▷ la consistance qui signifie que tout module doit être toujours dans une configuration prédéfinie, ainsi aucun état inconsistant ne sera atteint.

- ▷ la stabilité du système. Cette propriété signifie qu'il n'existe pas de séquence d'adaptabilité infinie.

Les propriétés de non blocage et de consistance sont exprimées en CTL (Computation Tree Logic) et vérifiées en utilisant les techniques du model-checking standards. Tandis que la propriété de stabilité est exprimée en *μ-Calculs* et est vérifiée à l'aide du model checker *Averest*. Le formalisme *μ-Calculs* permet de raisonner de manière plus efficace que la logique temporelle sur la propriété de stabilité.

2.3.2.2 Object-Z

G. Smith et ses collègues proposent dans [Smith *et al.*, 2012] une formalisation de l'adaptation. Cette formalisation repose sur la définition de modèles formels d'agents et de SMA en se basant sur les systèmes de transitions étiquetées (appelés aussi *LTS* (Labelled Transition Systems)). Un *LTS* est composé d'un ensemble d'états, d'un ensemble d'états initiaux et d'une collection d'actions permettant les transitions entre les états. Ainsi, un agent A_i est modélisé formellement par un *LTS* particulier appelé *automate de composants* décrit comme suit :

$$A_i = (Q_i, I_i, \Sigma_i, \delta_i) \tag{2.8}$$

Dans cette définition, Q dénote l'ensemble des états de l'agent. $I_i \subseteq Q_i$ est l'ensemble des états initiaux devant être non vide. $\Sigma_i = \Sigma_{i,int} \cup \Sigma_{i,inp} \cup \Sigma_{i,out}$ est l'ensemble des actions. Σ_i est l'union des actions internes de l'agent ($\Sigma_{i,int}$), de ses actions d'input ($\Sigma_{i,inp}$) et de ses actions d'output ($\Sigma_{i,out}$). Les actions internes sont contrôlées par l'agent et ne sont pas observables à l'extérieur. Les actions d'input sont observables à l'extérieur et contrôlées par l'environnement de l'agent. Finalement, les actions d'output sont observables à l'extérieur et contrôlées par l'agent. δ_i dénote l'ensemble des transitions entre les différents états de l'agent A_i.

Un SMA est obtenu à travers la composition d'automates en synchronisant les actions des agents. Formellement, un SMA composé de n agents $A_1, ...A_n$ est un *LTS*, $M = (Q, I, \Sigma, \delta)$, tels que :

- ▷ $Q = \prod_{i:1..n} Q_i$: l'ensemble des états possibles de M est le produit cartésien des états des différents agents qui le composent.

- ▷ $I = \prod_{i:1..n} I_i$

- $\Sigma = \Sigma_{int} \cup \Sigma_{inp} \cup \Sigma_{out}$ tels que :
  - $\Sigma_{int} = \bigcup_{i=0..n} \Sigma_{i,int}$: les actions internes de M sont les actions internes des agents.
  - $\Sigma_{out} = \bigcup_{i=0..n} \Sigma_{i,out}$: les actions d'output de M sont les actions contrôlées par les agents et observables à l'extérieur.
  - $\Sigma_{inp} = (\bigcup_{i=0..n} \Sigma_{i,inp}) \setminus \Sigma_{out}$: les actions d'input de M sont les actions contrôlées par l'environnement et observables à l'extérieur mais qui ne sont pas aussi des actions contrôlées par les agents.
- $\delta \subseteq Q \times \Sigma \times Q$; les transitions décrivent l'évolution des états d'un ensemble d'agents impliqués dans la même action. Les états des agents qui ne sont pas concernés restent inchangés. Chaque transition interne d'un agent correspond à une transition dans le SMA. Afin d'assurer la consistance de la transition impliquant plusieurs agents, toute action d'output dans le système doit être une action d'output pour au moins un agent impliqué dans la transition.

La formalisation de l'adaptabilité permet d'évaluer la capacité du système à revenir sur un état *légitime* [Dijkstra, 1974]) face à une action externe Z le plaçant dans un état *non légitime* en supposant qu'une seule occurrence de Z a eu lieu. Le SMA est dit *Z-adaptatif* si les deux conditions suivantes sont vérifiées :

- le système M' est capable d'atteindre un état légitime après avoir passé par un état non légitime suite à une occurrence de Z. Cette condition est formalisée comme suit :
  $\forall b' \in B(M', q') \cdot \exists j > 0 \cdot st(b, j) \in Q(M')$

- avoir au moins une séquence d'états et d'actions lui permettant d'atteindre un état légitime sans changer les variables auxiliaires. Formellement, cette condition est exprimée comme suit :
  $\forall b' \in B(M', q') \cdot \exists j \geq 0 \cdot st(b, j) \in Q(M') \wedge (\forall k \leqslant j \cdot st(b, k)|_E = q'|_E)$

Dans ces deux dernières expressions, $B(M', q')$ décrit l'ensemble de tous les comportements possibles de M à partir de l'état q'. Le comportement du système est décrit à l'aide d'une séquence, pouvant être infinie, d'état et d'action alternés. $st(b, j)$ représente le $j^{\text{ème}}$ état de b. $q'|_E$ restreint l'état q' aux variables E. Ces deux conditions doivent être vérifiées pour toute séquence de comportement $b \in B(A')$ pouvant contenir un état non légitime $st(b, i)$ tel que $st(b, i) = q$ et $(q, Z, q') \in \zeta$, ζ étant l'ensemble des transitions définies par l'action Z.

Cette formalisation est assez générique mais elle n'est pas convenable pour modéliser des systèmes non triviaux. Pour pallier cette limite, les auteurs proposent de traduire ces définitions dans le langage *Object-Z* ([Smith, 2000]). Ainsi, les différents états du SMA sont modélisés par des schémas Z et ses actions par des opérations. La propriété d'adaptabilité est exprimée dans la logique *CTL* et elle est vérifiée par le model-checker *SAL* [Smith et Wildman, 2005].

2.3.2.3 Automates temporisés

Dans [Iftikhar et Weyns, 2012], la technique du model-checking a été utilisée pour la vérification d'un système auto-adaptatif décentralisée pour la surveillance du trafic routier. Dans le cadre de ce travail, le système est modélisé par un ensemble de processus interconnectés. Chaque processus est modélisé par un automate temporisé (timed automata) : une machine à états finis avec des variables horloges. Les horloges sont des valeurs réelles qui servent à mesurer le temps écoulé entre deux événements. Ces automates servent à créer des modèles génériques des éléments du systèmes qui seront instanciés pour assurer la vérification.
Les propriétés considérées sont relatives à *la robustesse* et *la flexibilité* du système. Ces propriétés sont décrites par la logique $TCTL$ (Timed Computaion Tree Logic) et leur vérification formelle a été assurée par le model checker $Uppaal$.

Dans ces travaux, la dynamique de l'environnement a bien été prise en compte dans la modélisation et dans la vérification. Cependant, certaines limites sont à noter. La première est relative à l'instanciation des modèles génériques pour effectuer la vérification. Cette instanciation n'est pas toujours évidente dans le cas de systèmes à large échelle. La deuxième limite concerne l'utilisation des horloges. Les horloges ont servi uniquement pour modéliser la progression du système dans le temps et exprimer les contraintes sur les transitions entre les différents états. Cette utilisation a pu être étendue pour raisonner de manière quantitative sur la robustesse et la résilience du système en calculant par exemple le temps nécessaire pour la réparation du système en cas de panne.

D'après les travaux cités ci-dessous, la technique du model-checking a montré son efficacité pour la vérification des propriétés relatives à l'adaptabilité. Ces travaux se sont intéressés aux systèmes auto-adaptatifs dont l'ensemble des configurations possibles est généralement connu à l'avance. Cette condition ne peut pas être appliquée sur les systèmes auto-organisateurs dont la fonctionnalité est émergente.

Le model-checking stochastique est une technique de model-checking particulière dont l'utilisation s'est répondue pour la vérification de systèmes stochastiques. Dans la section suivante, nous décrivons les processus stochastiques utilisés pour modéliser ce type de systèmes et nous citons des travaux ayant utilisé le model-checking stocastique pour la vérification de leurs propriétés.

2.3.2.4 Processus stochastiques

Un processus stochastique permet de modéliser l'évolution de systèmes aléatoires en prenant en compte la survenance d'événements imprévisibles. Un processus stochastique $\{X_t, t \in T\}$ est une collection de variables aléatoires indexées par un paramètre t et définies sur un même espace de probabilité. t dénote le temps et X_t représente l'état du processus à l'instant t.

Dans la plupart des travaux, la modélisation stochastique des systèmes auto-organisateurs se fait à l'aide d'un type bien particulier des processus stochastiques appelé *chaînes de Markov*. La particularité des chaînes de Markov réside dans la propriété d'"absence de mémoire" appelé propriété de Markov. Un processus stochastique vérifie cette propriété si et seulement si la loi de probabilité des états futurs du processus dépend uniquement de son état présent. Ceci se traduit par l'équation suivante :

$$Pr(X_{n+1} = x|X_0, X_1, X_2, ...X_n) = Pr(X_{n+1} = x|X_n).$$

Une chaîne de Markov peut être à temps discret (Discrete-Time Markov Chain ou DTMC) ou à temps continu (Continuous-Time Markov Chain ou CTMC).

Pour les DTMC, si le processus est dans un état y à l'instant n, la probabilité qu'il passe à l'état x à l'instant $n + 1$ est indépendant de n et dépend uniquement de son état courant y. Ainsi, à tout instant n, une chaîne de Markov peut être représentée par une matrice de probabilité dont l'élément (x, y) est indépendant de n et est donnée par l'équation :

$$Pr(X_n = x|X_n = y).$$

Pour les CTMC, le temps passé dans chacun des états est une variable aléatoire réelle positive suivant une loi exponentielle. Les valeurs contenues dans une matrice de transition d'une CTMC représentent les taux de transition d'un état à un autre.

Récemment, plusieurs langages et outils pour la modélisation et la vérification de systèmes stochastiques ont été proposés. Parmi lesquels on peut citer les langages *Stochasticπ-Calculus* [Priami, 1995] (extension du langage π *Calculus* [Milner *et al.*, 1992]) et *Bio-PEPA* [Hillston, 2005c] (extension du langage *PEPA* (Performance Evaluation Process Algebra) [Hillston, 2005a]), [Hillston, 2005b] et les outils SPiM [Phillips et Cardelli, 2007] et PRISM [8] [Hinton *et al.*, 2006], [Kwiatkowska *et al.*, 2011].

Ces outils offrent le moyen de modéliser des systèmes stochastiques et de les analyser au moyen de techniques numériques tel que le model-checking probabiliste ou au moyen de techniques statistiques tel que le *model-checking probabiliste approximatif* appelé aussi *Statistical model-checking*.

Plus récemment, la technique du model-checking a été étendue pour prendre en compte les aspects quantitatifs liés à la performance des systèmes donnant lieu à la technique du *model-checking probabiliste* (appelée aussi *model-checking stochastique* dans [Ciocchetta et Hillston]). Par conséquent, des versions probabilistes de la logique temporelle ont été proposées pour la formulation de propriétés quantitatives.

Parmi ces extensions, on cite la logique stochastique continue (Continuous Stochastic Logic) [Aziz *et al.*, 2000], la logique PCTL (Probabilistic Computation Tree Logic) [Hansson

8. http ://www.prismmodelchecker.org/

et Jonsson, 1994] et PLTL (Probabilistic Linear Temporal Logic).

La méthode de vérification avec cette technique est basée sur des algorithmes numériques pour calculer de manière exacte les probabilités sur le passage entre les différents états ou les états stables. L'application de cette technique nécessite la construction de tout l'espace d'états que le système peut atteindre ce qui limite son utilisation aux modèles dont le nombre d'états est de l'ordre de 10^7 [Ciocchetta et Hillston], [Nimal, 2010].

Pour pallier cet inconvénient, une solution possible consiste à utiliser des techniques d'approximation pour estimer la probabilité à calculer. Cette technique se compose de trois étapes :

- ▷ générer plusieurs exécutions aléatoires du système,
- ▷ vérifier la propriété considérée sur ces exécutions générées et
- ▷ calculer la probabilité en faisant la moyenne des exécutions vérifiant la propriété considérée.

Dans [Casadei et Viroli, 2009], une approche hybride pour la modélisation et la vérification de systèmes auto-organisateurs a été proposée. Cette approche utilise la simulation stochastique pour la modélisation du système dont l'évolution est capturée à l'aide des chaînes de Markov et la technique de model-checking probabiliste pour la vérification.

Pour éviter le problème d'explosion d'états, rencontré avec les model-checkers, l'approche propose de faire recours au model-checking approximatif qui repose sur les simulations. En effet, pour vérifier une certaine propriété, le modèle du système sera exécuté plusieurs fois. Pour chaque exécution, le résultat est évalué. Le résultat final est la moyenne des résultats obtenus.

L'approche a été testée pour le problème de tri collectif qui consiste à ranger les éléments de même type dans le même groupe. Le tri collectif a été modélisé l'aide de l'outil *PRISM* ([Kwiatkowska *et al.*, 2009]) en tant qu'une chaîne de Markov à temps discret en modélisant explicitement toutes les transitions possibles entre les différents groupes d'éléments.

Konur et al. [Konur *et al.*, 2012] utilisent également l'outil *PRISM* et le model-checking probabiliste pour la vérification des comportements des essaims de robots et en particulier les robots fourrageurs. Pour éviter le problème d'explosion d'états, les auteurs optent pour une approche macroscopique au lieu d'une approche microscopique. Dans l'approche microscopique, chaque robot est représenté par une machine à états finis probabiliste et le système est généré en faisant le produit de toutes ces machines. Dans l'approche macroscopique, on considère uniquement le comportement de la population en sa globalité.

Le système est alors modélisé par une seule machine à états dont chaque état contient une information sur le nombre total d'agents dans cet état ainsi qu'une description de l'évo-

lution de ce nombre dans le temps. Grâce à leur approche macroscopique, les auteurs ont réussi à vérifier certaines propriétés en utilisant aussi la logique *PCTL* et la simulation pour différents scénarios. Ces propriétés renseignent en particulier sur la probabilité que l'essaim acquière une certaine quantité d'énergie dans un délai et pour un nombre d'agents bien déterminé. Les simulations montrent la corrélation entre la densité des robots fourrageurs dans l'arène et la quantité d'énergie acquise.

2.3.3 Bilan sur les méthodes formelles

L'état de l'art concernant l'application des techniques formelles pour la vérification des systèmes auto-adaptatifs est très riche et varié. En effet, on remarque l'utilisation de différents formalismes et de différentes techniques : la vérification par la preuve, le model-checking ou encore le model-checking stochastique. Néanmoins, l'application de ces techniques pour des systèmes auto-organisateurs souffrent de plusieurs limites. En effet, il faut ajouter des hypothèses sur la dynamique du système pour pouvoir donner des garanties formelles par la preuve ce qui n'est pas adapté pour des systèmes auto-organisateurs. Ce problème a été résolu au moyen des techniques du model-checking stochastique qui prennent en compte le non déterminisme dans la modélisation du système. Une autre limite notée dans ces approches est que les travaux cités ne fournissent aucun guide ou heuristique au niveau de la modélisation des comportements des agents et la laissent entièrement à la charge de l'expertise du concepteur.

Tenant compte de ces limites, nous nous sommes fixés les objectifs suivants :

- décrire une méthode guidée par des heuristiques et basée sur des raffinements pour la modélisation du comportement des agents du systèmes. A partir de cette modélisation, on peut ensuite dériver le comportement du système dans sa globalité et raisonner par conséquent sur ses propriétés globales,
- prouver des propriétés locales qui concernent les agents,
- prouver les propriétés globales de convergence et de résilience. La convergence désigne la capacité du système à atteindre son objectif en l'absence de perturbations. La résilience renseigne sur la capacité du système à s'adapter face aux changements de son environnement.

Pour atteindre ces objectifs, nous choisissons d'utiliser le formalisme *B-événementiel* pour modéliser le comportement local des agents. Ce choix est dû pour trois raisons :

1. B-évènementiel est un langage bien supporté par des outils (en l'occurrence la plateforme *RODIN*) permettant la modélisation et la vérification de systèmes distribués.
2. Il s'agit d'un langage assez expressif pour modéliser les aspects relatifs aux SMA auto-organisateurs.
3. Finalement, *B-événementiel* repose sur le principe de raffinement qui consiste en le développement de systèmes de manière incrémentale et sûre ce qui permet de réduire fortement la difficulté de modélisation et de preuve des SMA auto-organisateurs.

B-événementiel ne permet pas la modélisation des propriétés relatives à la progression d'un système (des propriétés de vivacité). Ainsi, nous avons choisi d'utiliser la logique temporelle pour exprimer les propriétés globales du systèmes. Finalement, nous avons eu recours à la logique temporelle des actions pour pouvoir raisonner sur ces propriétés globales.

Dans le chapitre suivant, ces éléments de modélisation formelle sont présentés.

3 Outils formels utilisés pour la vérification

« C'est avec la logique que nous prouvons et avec l'intuition que nous trouvons. »

Henri Poincaré

3.1 Le langage B-Evénementiel

3.1.1 Concepts clés de modélisation

Le formalisme *B-événementiel* a été proposé par Jean Raymond Abrial [Abrial, 2010] comme une évolution du langage *B* [Abrial, 2005] mais aussi du formalisme des systèmes d'action (Action systems) [Back et Kurki-Suonio, 1983] [Back et Sere, 1991] auquel il a emprunté plusieurs concepts. Le langage *B-événementiel* permet la modélisation des systèmes distribués et réactifs à événements discrets en se basant sur la logique des prédicats de premier ordre et la théorie des ensembles.
Une spécification formelle écrite en *B-événementiel* contient deux types de composants : les contextes et les machines. Un contexte représente la partie statique de la spécification et comprend un ensemble de constantes ainsi que leurs propriétés. Une machine sert à modéliser la dynamique du système au moyen d'un ensemble de variables définissant l'état du système et d'un ensemble d'événements faisant évoluer ces variables. La figure 3.1 illustre de manière sommaire les différents éléments formant une machine et un contexte ainsi que la relation "*Regarde*" qui permet à une machine d'utiliser toutes les constantes définies dans un contexte. Dans les sections suivantes, nous détaillons davantage ces éléments et nous illustrons nos propos par un exemple simple d'un contrôleur des feux de circulation dont l'objectif est la régulation de la circulation [1].

1. cet exemple est tiré du site http ://handbook.event-b.org/current/html/

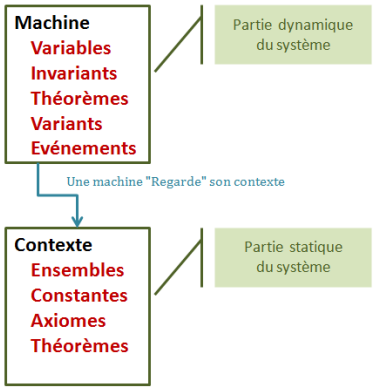

Figure 3.1 — Machine et contexte : les concepts clés dans une spécification en *B-événementiel*

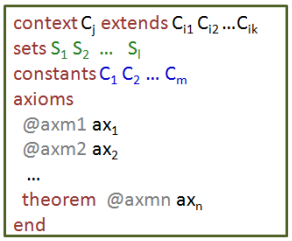

Figure 3.2 — Structure d'un contexte en *B-événementiel*

Description d'un contexte en *B-événementiel* En *B-événementiel*, un contexte regroupe les éléments statiques du système et se présente comme le montre la figure 3.2. Les clauses *sets* et *constants* renferment respectivement les ensembles et les constantes que l'utilisateur peut définir. Les ensembles définis par l'utilisateur doivent être non vides et vont servir pour typer d'autres éléments de la spécification. Dans la clause *axioms* sont définies, sous forme de prédicats, toutes les propriétés relatives aux constantes (leurs types entre autres) et aux ensembles déclarés. Certains axiomes peuvent être marqués comme *theorem*. Dans ce cas, ils doivent être inférés (prouvés) à l'aide des axiomes déjà définis. Un contexte peut utiliser des éléments déjà définis dans d'autres contextes. Dans ce cas, il s'agit d'une extension et les noms des contextes à étendre doivent figurer dans la clause *extends*.

Dans le contexte *ctx*1 (figure 3.3), l'ensemble *COLOURS* et les deux constantes *red* et *green* sont définis. L'axiome *@type* partitionne l'ensemble *COLOURS* en deux sous ensemble dont le premier est formé de la couleur *red* et le deuxième est formé de la couleur *green*.

```
context ctx1
sets COLOURS
constants red  green
axioms
 @type partition(COLOURS,{red},{green})
end
```

Figure 3.3 — Exemple : le contexte *ctx*1

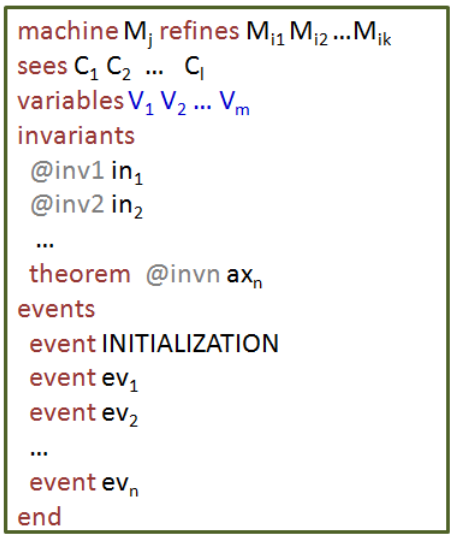

Figure 3.4 — Structure d'une machine en *B-événementiel*

Description d'une machine en *B-événementiel* Une machine décrit le comportement du système en modélisant son état par des variables dont les valeurs changent au moyen d'événements. La figure 3.4 illustre la structure d'une machine en *B-événementiel*. La clause *refines* est optionnelle et renferme les noms des machines à raffiner (le principe de raffinement sera abordé dans ce qui suit). La clause *sees* définit les contextes qu'une machine peut "*regarder*" ce qui lui permettra d'utiliser les constantes, les ensembles ainsi que les axiomes et les théorèmes définis dans ces contextes. Les valeurs des variables déclarées dans la clause *variables* et définissant l'état de la machine peuvent vérifier des propriétés d'invariance (appelées *invariants*) définies dans la clause *invariants*. Les invariants sont spécifiés sous forme de prédicats et doivent indiquer les types des variables de la machine. Les types utilisés peuvent être déclarés dans les contextes "regardés" par la machine. Les propriétés d'invariance pouvant être inférées à partir des autres invariants sont marquées comme *theorem*. Les variables sont initialisées par un événement spécifique (*event INITIALIZATION*) et évoluent grâce à l'activation des événements définis dans la clause *events*. Nous illustrons cette notion de machine par la machine *mac* donnée dans la figure 3.5.

Dans la machine *mac*, les signaux sont modélisés par les variables *peds_go* (pour les piétons) et *cars_go* (pour les voitures). Ces variables sont booléennes : la valeur FALSE indique l'obligation de s'arrêter et la valeur TRUE autorise la circulation. Les invariants *inv*1 et *inv*2 définissent le type des ces deux variables. La circulation simultanée des piétons et des voitures est interdite. Cette condition est exprimée par l'invariant *inv*3. Les événements *set_peds_go* et *set_peds_stop* sont responsables de l'évolution de la variable *peds_go*. L'événement *set_cars* se charge de mettre à jour la variable *cars_go*. Cette notion d'événement est présentée dans le paragraphe suivant.

Le concept d'événement en *B-événementiel* Un événement E est défini généralement par trois parties (comme le montre la figure 3.6) : un ensemble de paramètres p_E, une garde $g_E(C, V, p_E)$ qui donne les conditions nécessaires à l'activation de l'événement et une action $a_E(C, V, p_E)$ qui décrit le changement des valeurs des variables considérées dans cette

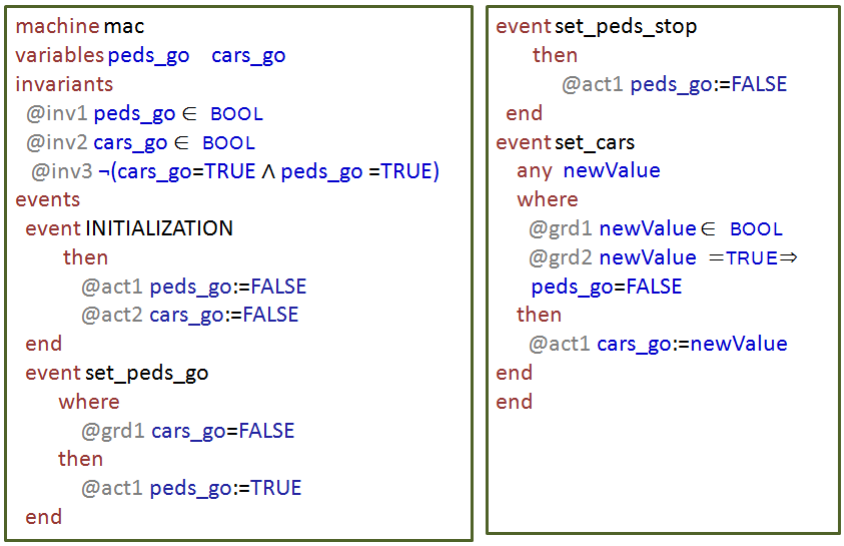

Figure 3.5 — Exemple : la machine *mac*

action. Les variables non considérées dans l'action gardent leurs valeurs inchangées. Les ensembles C et V représentent respectivement des constantes et des variables déclarées dans le modèle où est défini l'événement E.

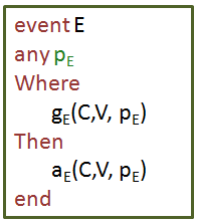

Figure 3.6 — Structure d'un événement en *B-événementiel*

L'événement *set_peds_go* de la machine *mac* (figure 3.5) permet d'autoriser la circulation des piétons en mettant la valeur de la variable *peds_go* à TRUE via son action *act*1. Il est déclenché lorsque la circulation des voitures est arrêtée. Cette condition est donnée par sa garde *grd*1. L'événement *set_cars* est défini en fonction du paramètre *newValue*. La première garde de cet événement définit le type de ce paramètre. La deuxième vérifie que la valeur de la variable *peds_go* et la nouvelle valeur de *cars_go* ne seront pas toutes les deux à TRUE et interdit ainsi aux piétons et aux voitures de circuler en même temps.

Il est à noter que lorsque la garde d'un événement est satisfaite, l'action de cet événement n'est pas nécessairement exécutée. En effet, les gardes de plusieurs événements peuvent être satisfaites en même temps et dans ce cas, un événement est choisi de manière arbitraire pour que son action soit déclenchée.

Une action peut se présenter sous trois formats possibles :

- une affectation $x := Exp(C, V, p_E)$ qui remplace la valeur de la variable x par la valeur de l'expression $Exp(C, V, p_E)$. La substitution $x := x + 1$ est une affectation.

- ▷ une substitution non déterministe ensembliste $x :\in S(C, V, p_E)$ qui choisit arbitrairement une valeur de l'ensemble $S(C, V, p_E)$ pour l'affecter à x. Par exemple, la substitution $x :\in \{1, 2, 3\}$ affecte à la variable x une valeur parmi les trois valeurs de l'ensemble $\{1, 2, 3\}$.
- ▷ une substitution non déterministe avec un prédicat $x : |Q(C, V, p_E, x')$ qui préserve le prédicat Q vérifié en modifiant la valeur de x. Dans le prédicat Q, x' dénote la valeur de la variable x après l'exécution de la substitution. Par exemple, la substitution $x : |x' = x + 1$ affecte à la variable x une valeur telle que sa nouvelle valeur est une incrémentation de l'ancienne.

Ce dernier format de substitution est plus général que les deux premiers. Ainsi, l'affectation et la substitution ensembliste peuvent être exprimées en fonction d'une substitution avec un prédicat. L'affectation $x := Exp(C, V, p_E)$ est équivalente à la substitution suivante $x : |(x' = Exp(C, V, p_E))$ et la substitution ensembliste $x :\in S(C, V, p_E)$ est équivalente à $x : |x' \in S(C, V, p_E)$. Par exemple, les substitutions $x := x + 1$ et $x :\in \{1, 2, 3\}$ sont équivalentes respectivement aux substitutions $x : |x' = x + 1$ et $x : |x' \in \{1, 2, 3\}$.

La forme généralisée permet aussi d'écrire des substitutions parallèles permettant de modifier les valeurs de plusieurs variables distinctes simultanément. Son utilisation devient obligatoire lorsque l'action modifie plusieurs variables dont la nouvelle valeur de l'une d'entre elles dépend de la nouvelle valeur d'une autre variable.
Chaque événement peut être reformulé sous la forme d'un prédicat appelé prédicat *avant-après* (*before-after predicate*) qui détermine la relation entre l'état de la machine avant et après l'exécution de l'événement. Le prédicat *avant-après* de l'événement E représenté dans la figure 3.6 est $\exists p_E.(g_E(C, V, p_E) \wedge a_E(C, V, p_E))$. On notera le prédicat *avant-après* d'un événement E par BA_E.

3.1.2 Le raffinement : garantie de la correction en *B-événementiel*

Le raffinement est le principe clé dans le développement de systèmes à l'aide de *B-événementiel*. Cette technique trouve ses origines dans les travaux de Dijkstra [Dijkstra, 1970] où elle a été proposée comme un moyen de construction systématique de programmes en faisant des raffinements successifs. Cette notion de raffinement a été définie comme une relation permettant la préservation de la **correction** de programmes et formulée en termes de transformations de prédicats par Back dans [Back, 1978] et [Back, 1980]. Un programme est dit **correct** s'il est conforme à sa spécification (la spécification peut être par exemple une formulation des besoins des utilisateurs). Ces travaux ont été ultérieurement développés pour donner lieu au *calcul de raffinement* (refinement calculus) [Back et von Wright, 1998] représentant une base mathématique permettant de raisonner rigoureusement sur la correction et le raffinement de programmes. Le processus de raffinement utilisé par *B-événementiel* est basé sur ce cadre théorique du *calcul de raffinement*.

Le raffinement d'un modèle abstrait M_a par un modèle concret M_c noté $M_a \sqsubseteq M_c$ consiste à ajouter des détails au modèle abstrait tout en conservant la correction du modèle.

Le processus de raffinement commence par un modèle initial M_0 qui satisfait les critères de correction et consiste à faire des étapes de raffinement successives :

$$M_0 \sqsubseteq M_1 \sqsubseteq ... \sqsubseteq M_n.$$

Il est garanti par transitivité que $M_0 \sqsubseteq M_n$. Il est garanti également que le modèle M_n satisfait les contraintes de correction initiales car la relation de raffinement préserve la correction. Le processus de raffinement permet ainsi une conception *correcte par construction*.

Avec *B-événementiel*, le raffinement relie deux machines : une machine *abstraite* (à raffiner) et une machine *concrète* (résultat du raffinement) et est réalisé en raffinant les variables ainsi que les événements de la machine abstraite. On appelle événement abstrait (respectivement concret) un événement d'une machine abstraite (respectivement concrète) et de même pour les paramètres, les gardes et les variables.

Le raffinement d'une variable abstraite se fait en spécifiant sa relation avec la variable concrète qui la raffine. Cette relation est exprimée à l'aide d'un invariant appelé *invariant de collage* (*gluing invariant*). Pour notre exemple du contrôleur de circulation, le raffinement de la machine *mac* par l'introduction des couleurs des feux de signalisation pour les piétons entraîne le remplacement de la variable *peds_go* par la variable *peds_colour* et l'ajout de l'invariant de collage $peds\_go = TRUE \Leftrightarrow peds\_colour = green$.

Un événement abstrait peut être raffiné dans la machine concrète par un ou plusieurs événements concrets. Généralement, ce raffinement est effectué en renforçant la garde abstraite et en réduisant le non-déterminisme dans les actions abstraites. Pour garantir que le comportement de la machine concrète correspond au comportement de la machine abstraite deux conditions doivent être vérifiées. La première condition assure que les événements concrets ne peuvent être déclenchés que si leurs événements abstraits le sont aussi, c'est ce qu'on appelle le renforcement de la garde. La deuxième condition garantit que la survenance d'un événement concret doit se faire conformément à l'événement abstrait de manière à préserver les invariants de collage.

Dans la partie gauche de la figure 3.7, nous présentons l'exemple de la machine *mac*1 qui raffine la machine *mac* (figure 3.5). Dans la partie droite, les événements de la machine concrète sont détaillés. Les actions des événements *set_peds_go* et *set_peds_stop* ainsi que la garde *grd*2 de l'événement abstrait *set_cars* ont été raffinées pour prendre en compte l'introduction de la variable *peds_colour*.

Dans le processus de raffinement, de nouveaux événements peuvent être introduits. Dans ce cas, on dit qu'ils raffinent l'événement *Skip* qui est un événement qui ne fait rien et qui peut survenir à n'importe quel moment. Les paramètres abstraits peuvent également être raffinés. Dans ce cas, il faut utiliser des *témoins* (*Witness*) qui décrivent la relation entre les paramètres abstraits et les paramètres concrets.

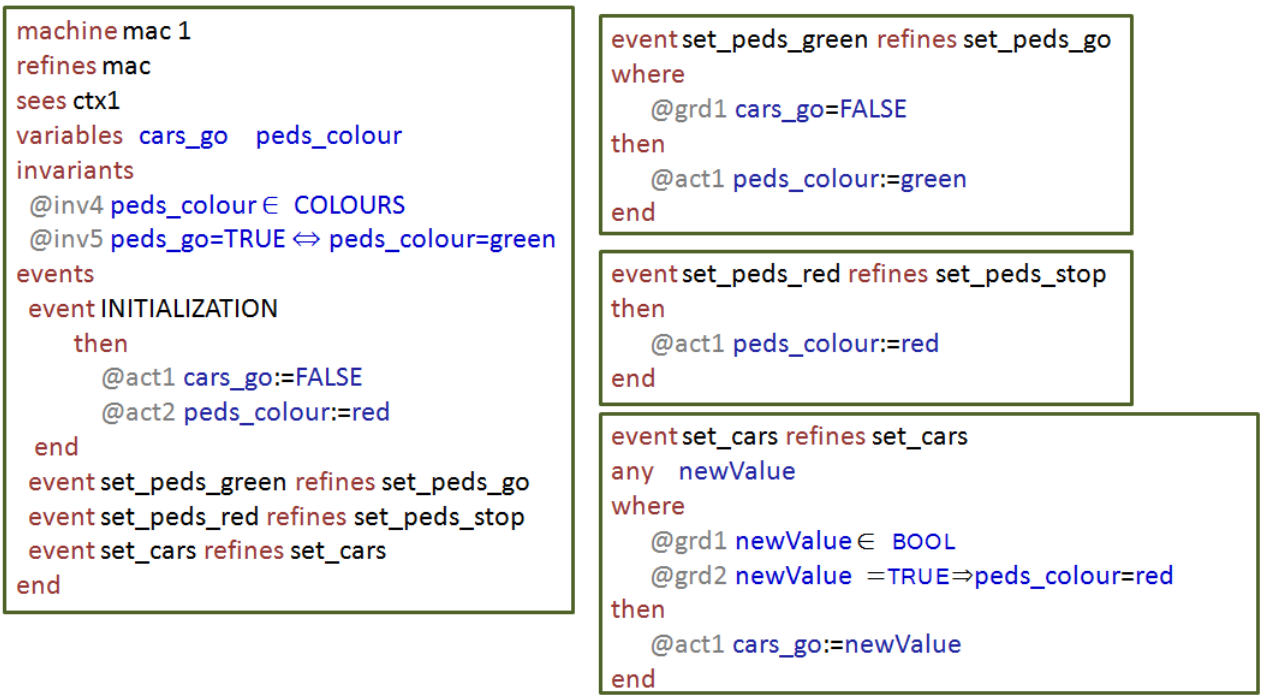

Figure 3.7 — La machine *mac*1 qui raffine la machine *mac*

La **correction** au sens de *B-événementiel* est définie par un ensemble de propriétés logiques telles que la faisabilité des actions des événements, la préservation des invariants et la correction des raffinements. La préservation de ces propriétés détermine la correction de la spécification et est garantie en définissant un ensemble d'obligations de preuves. Une spécification en *B-événementiel* est dite **correcte** lorsque toutes les obligations de preuve sont prouvées (ou déchargées). Elles sont décrites dans la section 3.1.4.

Le processus de raffinement réduit considérablement les efforts dédiés aux tests du système conçu et permet d'établir un lien de traçabilité entre les propriétés du système à différents niveaux d'abstraction. Néanmoins l'utilisation de ce principe de modélisation nécessite une expertise en terme d'abstraction et un savoir-faire pour maîtriser la complexité des preuves qui lui sont associées.

Afin de faciliter l'intégration du raffinement dans le processus de développement de systèmes, il est essentiel d'automatiser le processus de raffinement et de privilégier la réutilisation des modèles et des preuves. L'objectif est d'automatiser certaines étapes de transformation de modèles par l'instanciation et l'utilisation de modèles de solutions déjà établis et prouvés appelés des *patrons de raffinement*.

3.1.3 La modélisation basée sur les patrons de raffinement

Alexei Iliasov définit dans [Iliasov *et al.*, 2009] un **patron de raffinement** comme une transformation générique de modèles qui prend en entrée un modèle paramétré (le

modèle p_0 dans la figure 3.8) et fournit comme résultat un nouveau modèle représentant le raffinement du modèle en entrée (le modèle p_1 dans la figure 3.8). Un patron de raffinement est composé de trois parties.
La première partie regroupe les conditions d'applicabilité du patron ; les conditions syntaxiques et sémantiques que le modèle doit vérifier pour être éligible à l'application du patron.
La seconde partie contient la définition des transformations (les raffinements) que le patron va faire subir au modèle initial.
La dernière partie comporte les obligations de preuve qui garantissent que les transformations effectuées correspondent bien à des étapes de raffinement.
Une modélisation basée sur les patrons de raffinement est illustrée par la figure 3.8. Le processus de développement formel commence par le modèle initial m_0. A l'étape de raffinement n, le modèle obtenu m_n remplit les conditions d'applicabilité du patron p et par conséquent une correspondance entre m_n et p_0 est réalisée. Les raffinements définis dans le modèle p_1 sont incorporés dans le processus de développement pour créer le modèle m_{n+1}.

Lors de l'instanciation, les paramètres du modèle p_0 sont remplacés par les données concrètes du modèle m_n. Les contraintes définies dans le patron pour ces paramètres deviennent des théorèmes à prouver pour montrer que le patron est applicable au modèle m_n. Une fois l'incorporation achevée, les propriétés d'invariance ainsi que les preuves qui leur correspondent sont retrouvées dans le modèle m_{n+1}. Ces preuves ainsi que les preuves relatives au raffinement ne doivent pas être prouvées car elles le sont déjà dans le patron.

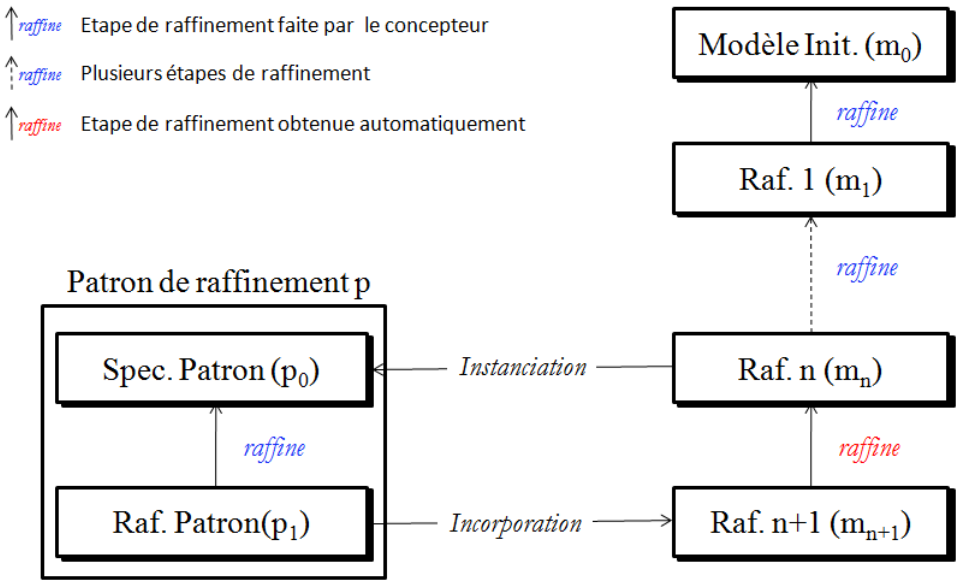

Figure 3.8 — Une modélisation basée sur les patrons de raffinement

3.1.4 Les obligations de preuve

Les obligations de preuve relatives à une spécification peuvent être classées en quatre catégories :

- ▷ les obligations de preuve générées dans un contexte,
- ▷ les obligations de preuve définies pour prouver la consistance d'une machine,
- ▷ les obligations de preuve générées pour les raffinements,
- ▷ les obligations de preuve qui servent pour la preuve de la terminaison des événements.

3.1.4.1 Les obligations de preuve spécifiques aux contextes et la consistance des machines

Dans le tableau 3.1, on mentionne les principaux types d'obligations de preuve qu'on peut trouver dans les deux premières catégories. Les obligations de preuve de bonne

	Contexte	**Consistance d'une machine**
WD	bonne définition des axiomes	bonne définition des invariants, des actions ou des gardes
THM	preuve des axiomes marqués comme théorèmes	preuve des invariants et gardes marqués comme théorèmes
INV		preuve de la préservation des invariants
FIS		preuve de la faisabilité des actions non déterministes

Tableau 3.1 — Les principaux types d'obligations de preuve générées dans un contexte et celles générées pour prouver la consistance d'une machine

définition notées WD ($Well\ Definedness$) peuvent être appliquées aux axiomes, aux invariants ou encore aux gardes et aux actions d'un événement. Elles assurent qu'un prédicat ou une affectation est bien défini. Par exemple, pour effectuer une division de A par B, le dénominateur B doit être différent de zéro. Cette restriction représente une contrainte de bonne définition de l'expression A/B. Un autre exemple consiste en l'application de l'opérateur $Card$ sur un ensemble E. La contrainte de bonne définition de l'expression $Card(E)$ est que E doit être fini.

Les obligations de preuve THM ($THeoreM$) assurent qu'un axiome, un invariant ou une garde marqué comme théorème, sera effectivement prouvé. Les obligations de preuve d'in-

variance notées INV (*INVariant*) et celles de faisabilité notées FIS (*Feasibility*) garantissent la consistance d'une machine. Les obligations de preuve INV assurent que les propriétés d'invariance, exprimées par des invariants, ne seront jamais violées durant l'évolution du système. Si on considère $I(v)$ un invariant d'une machine, alors pour tout événement E dont le prédicat *avant-après* est $BA_E(v, v')$ l'obligation de preuve correspondante à la préservation de l'invariant I est donnée par l'implication suivante :

$$I(v) \wedge BA_E(v, v') \Rightarrow I(v')$$

Les obligations de preuves de faisabilité notées FIS assurent la faisabilité de tout événement défini dans une machine. La faisabilité d'un événement signifie qu'à chaque activation, il doit y avoir pour toute action $v : |Q(v, v')$ une valeur v' à attribuer à la variable v satisfaisant le prédicat $Q(v, v')$. Si on considère un événement E ayant la garde $g_E(C, V, p_E)$, $I(v)$ étant l'invariant défini pour la variable v, l'obligation de preuve qui correspond à la faisabilité de l'événement E est donnée par l'implication suivante :

$$I(v) \wedge g_E(C, V, p_E) \Rightarrow \exists v'.Q(v, v')$$

3.1.4.2 Les obligations de preuve spécifiques à la correction du raffinement

La correction d'un raffinement est assurée grâce à un ensemble d'obligations de preuve dont les plus importantes sont mentionnées dans le tableau 3.2.

	Correction du raffinement
GRD	renforcement des gardes
SIM	simulation des actions
EQL	equality of preserved variable

Tableau 3.2 — Les principaux types d'OP assurant la correction d'un raffinement

Les obligations de preuve de renforcement des gardes (notées GRD) et de simulation des actions (notées SIM) concernent la preuve du raffinement des événements. L'obligation de preuve d'égalité des variables conservées (figurant aussi bien dans la machine concrète et la machine abstraite) notée EQL (*EQuaLity*) concerne le raffinement des données. Un événement raffiné ne doit être activé que lorsque l'événement abstrait est activé. Ainsi, la garde d'un événement concret doit renforcer celle de l'événement abstrait. Cette contrainte est assurée par l'obligation de preuve GRD formulée comme suit :

$$I(c, v) \wedge J(c, v, w) \wedge g_{Ec}(c, w, p_{Ec}) \Rightarrow g_{Ea}(c, v, p_{Ea})$$

Dans cette formule, $I(c,v)$ représente l'invariant abstrait, $J(c,v,w)$ l'invariant de collage, $g_{Ec}(c,w,p_{Ec})$ la garde de l'événement concret Ec et $g_{Ea}(c,v,p_{Ea})$ la garde de l'événement abstrait Ea.
L'action d'un événement concret peut modifier les valeurs de certaines variables déclarées dans la machine abstraite. Cette modification doit être faite de manière à ce que le comportement concret correspond au comportement abstrait. En considérant $I(c,v)$ l'invariant abstrait, $J(c,v,w)$ l'invariant de collage, $BA_{Ea}(v,v')$ le prédicat *avant-après* de l'événement abstrait et $BA_{Ec}(w,w')$ le prédicat *avant-après* de l'événement concret, l'obligation de preuve montrant la correction du raffinement de Ea par Ec est définie comme suit :

$$I(c,v) \wedge J(c,v,w) \wedge BA_{Ec}(c,w,w') \Rightarrow \exists v'.(BA_{Ea}(c,v,v') \wedge J(c,v',w'))$$

Lorsqu'une variable notée x est utilisée par la machine abstraite et la machine concrète, les événements de la machine concrète qui modifient la variable x doivent garder sa valeur inchangée. Cette contrainte est vérifiée par l'obligation de preuve EQL définie comme suit :

$$I(c,v) \wedge J(c,v,w) \wedge BA_{Ec}(c,w,w') \Rightarrow x' = x$$

Le prédicat $BA_{Ec}(c,w,w')$ est le prédicat *avant-après* de l'événement concret qui change la variable x.

Outre ces obligations de preuve, il est essentiel dans certains cas de prouver le non blocage d'une machine en garantissant qu'à chaque instant il existe au moins un événement déclenchable. Ceci peut être formulé en montrant que la disjonction des gardes des actions est toujours satisfaite sous l'hypothèse de la préservation des invariants. Cette contrainte de non blocage (Deadlock freedom) est formalisée comme suit :

$$I(c,v) \Rightarrow g_{E1}(c,w,p_{E1}) \vee ... \vee g_{Ek}(c,w,p_{Ek})$$

3.1.4.3 Les obligations de preuve spécifiques à la terminaison des événements

Dans un raffinement, de nouveaux événements concrets peuvent être ajoutés raffinant l'événement *Skip* dont l'action ne fait rien. A fin de préserver la propriété de non blocage par le raffinement, il faut prouver que les nouveaux événements sont convergents signifiant qu'ils ne seront pas déclenchables infiniment empêchant ainsi les autres événements de se déclencher. La preuve de la convergence [2] (appelée aussi la terminaison) des nouveaux événements se fait en définissant un variant –une expression numérique ou un ensemble fini– pour tous les événements convergents et en montrant que chaque exécution de l'un de ces événements le fait décroître. Lorsque la preuve de la terminaison d'un événement ne peut pas être prouvée à un niveau de raffinement, cet événement est marqué *anticip*

2. La notion de convergence évoquée avec B-événementiel est synonyme de terminaison et elle est différente de la propriété de convergence évoquée avec les systèmes auto-organisateurs

(*anticipated*) ce qui signifie que sa terminaison doit être prouvée dans les prochaines étapes de raffinement.

Les obligations de preuve citées dans cette section sont relatives à la preuve de la terminaison et sont résumées dans le tableau 3.3.
Les obligations de preuve FIN et NAT concernent le variant. FIN est utilisée lorsque le

	Terminaison des événements
FIN	le variant est un ensemble fini
NAT	le variant est un entier naturel
VAR	le variant décroît

Tableau 3.3 — Les principaux types d'obligations de preuve assurant la terminaison d'un événement convergent

variant est un ensemble et elle assure que cet ensemble est fini. Tandis que NAT est utilisée lorsque le variant est une expression numérique et permet de prouver que cette expression est entière. L'obligation de preuve FIN est donnée par la formule suivante dans laquelle $Finite(V(c,w))$ est un prédicat qui indique que l'ensemble $V(c,w)$ (le variant) est fini.

$$I(c,v) \wedge J(c,v,w) \Rightarrow Finite(V(c,w))$$

L'obligation de preuve NAT est formulée comme suit :

$$I(c,v) \wedge J(c,v,w) \Rightarrow V(c,w) \in \mathbb{N}$$

L'obligation de preuve VAR garantit que l'exécution de chaque événement convergent décroît le variant. Elle est définie comme suit :

$$I(c,v) \wedge J(c,v,w) \wedge g_{Ec}(c,w,p_{Ec}) \Rightarrow V(c,w') < V(c,w)$$

3.1.5 Une interprétation plus opérationnelle d'une machine *B-événementiel*

Dans les sections précédentes, nous avons vu qu'une spécification en *B-événementiel* est formée d'une séquence de machines reliées par des relations de raffinement. Les variables abstraites v sont reliées aux variables concrètes par un *invariant de collage* $J(v,w)$. Le comportement de la machine concrète doit être *simulé* par le comportement de la machine abstraite conformément à l'invariant de collage $J(v,w)$.
Dans cette section, nous présentons une interprétation plus opérationnelle des modèles en *B-événementiel* pour la formalisation des systèmes muli-agents auto-organisateurs. Cette interprétation a été proposée par Thai Son Hoang et Jean-Raymond Abrial dans [Hoang

et Abrial, 2011] et a pour objectif de permettre de prouver des propriétés temporelles de *vivacité* sur un modèle en *B-événementiel*.
Une machine en *B-événementiel* correspond à un automate à état/transition : les états s sont définis par des vecteurs (tuples) $\langle \overline{v} \rangle$ représentant les valeurs des variables v, les transitions entre les états sont définies par les événements. Un événement *evt* est dit *activable* (enabled) dans un état s lorsqu'il existe un paramètre x tel que la garde G de l'événement *evt* soit satisfaite à l'état s. Dans le cas contraire, l'événement est dit *non activable* (disabled). Une machine M est dite *bloquée* dans un état s si tous ses événements sont non activables dans cet état.

Pour un événement *evt*, un état t est un *evt-successeur* de l'état s s'il peut être atteint par l'exécution de l'événement *evt* à partir de l'état s. Pour une machine M, un état t est un *M-successeur* de l'état s s'il existe un événement *evt* de M tel que t est un *evt-successeur* de l'état s.

Une trace σ d'une machine M est une séquence d'états pouvant être finie ou infinie $s_0, s_1, ...$ vérifiant les trois conditions suivantes :

- s_0 est l'état initial vérifiant le prédicat *avant-après* de l'événement d'initialisation $BA_{INITIALIZATION}$.
- Pour chaque paire d'états consécutifs s_i et s_{i+1}, s_{i+1} est un état *M-successeur* de s_i.
- Si cette séquence est finie et se termine par un état s_{final}, alors la machine est bloquée dans s_{final}.

On note par $\mathcal{T}(M)$ l'ensemble des chemins d'exécution (ou traces d'exécution) possibles d'une machine M.
En se basant sur la représentation d'une machine sous forme d'une trace, il est possible d'utiliser la Logique Temporelle Linéaire (*LTL*) [Manna et Pnueli, 1984] pour spécifier les propriétés à vérifier sur cette trace. La notation $\sigma \models \phi$ dénote que la trace σ satisfait la formule ϕ. Une formule *LTL* est une formule de la logique de premier ordre à laquelle on peut appliquer les opérateurs temporels suivant : *toujours* ($\Box$), *fatalement* ($\Diamond$) et *jusqu'à* ($\mathcal{U}$). On dénote par σ une séquence non vide d'états $s_0, s_1, \ldots$ et par σ^k la séquence d'états $s_k,\ s_{k+1}, \ldots$.
La sémantique des opérateurs temporels est définie comme suit :

- $\sigma \models \Box\, \phi$ si pour tout $k = 0, 1, ...$, on a $\sigma^k \models \phi$
- $\sigma \models \Diamond\, \phi$ s'il existe $k = 0, 1, ...$, tel que $\sigma^k \models \phi$
- $\sigma \models \phi_1 \mathcal{U}\, \phi_2$ s'il existe $k = 0, 1, ...$, tel que $\sigma^k \models \phi_2$ et $\sigma^0 \models \phi_1,\ ...,\ \sigma^{k-1} \models \phi_1$ (ϕ_1 et ϕ_2 dénotent des formules *LTL*).

Une machine M satisfait une propriété ϕ (noté $M \models \phi$) si toutes ses traces satisfont la propriété ϕ. Cette relation de satisfaction est notée comme suit :

$$M \models \phi \text{ si } \forall \sigma \in \mathcal{T}(M).\sigma \models \phi$$

On note par $M \vdash \phi$ le fait que la relation $M \models \phi$ est prouvée.
Deux propriétés importantes peuvent être vérifiées sur les machines décrites en *B-événementiel*. Il s'agit de vérifier l'absence d'états de blocage (le **non blocage**) et la **vivacité** (*liveness*) du système. La machine se trouve dans un état de blocage si elle ne peut plus évoluer à partir de cet état, ce qui signifie qu'aucun de ses événements n'est activable. La vivacité exprime qu'une propriété souhaitable (une bonne propriété) finira par avoir lieu. Dans ce qui suit, nous commençons par définir l'obligation de preuve relative à la propriété de non blocage. Ensuite, nous présentons les règles de preuves permettant de prouver trois classes de propriétés de *vivacité* : l'existence, la progression et la persistance. Les preuves de ces règles sont données dans [Hoang et Abrial, 2011].

3.1.5.1 Propriété de non blocage

Le non blocage d'une machine M dans un état vérifiant la propriété P spécifie que toute trace finie de M ne se termine pas avec un état vérifiant P. Ceci est assuré en prouvant qu'à un état vérifiant la propriété P, il existe au moins un événement activable parmi les événements de la machine M. Un événement est dit activable lorsque sa garde est vraie. Formellement, cette preuve se traduit par la démonstration de la proposition suivante :

$$P(v) \Rightarrow \bigvee_i (\exists x.G_i(x,v))$$

Dans cette formule, v dénote un vecteur de variables de la machine M, x et $G_i(x,v)$ représentent respectivement les paramètres et la garde d'un événement de la machine M. Nous utilisons la notation $M \vdash\circlearrowleft P$ pour désigner le fait que la machine M est non bloquée dans un état vérifiant la propriété P.

3.1.5.2 Propriété d'existence

La propriété d'existence ($M \vdash \Box\Diamond P$) spécifie qu'une (bonne) propriété P est toujours fatale. La règle de preuve $LIVE_{\Box\Diamond}$ permet de prouver qu'une machine M vérifie la propriété d'existence en utilisant les propriétés de convergence et de non blocage.

$$\frac{\begin{array}{l} M \vdash\downarrow \neg P \\ M \vdash\circlearrowleft \neg P \end{array}}{M \vdash \Box\Diamond P} \; LIVE_{\Box\Diamond}$$

Dans cette règle de preuve, la première prémisse (celle du haut) dénote le fait que M est convergente en $\neg P$. Elle garantit qu'une trace ϕ de la machine M ne peut pas se terminer dans un état vérifiant $\neg P$ ce qui signifie qu'un état vérifiant P finira par apparaître. La deuxième prémisse garantit, dans le cas où la trace ϕ est finie, que la trace ne se termine pas dans un état vérifiant $\neg P$.
Avec *B-événementiel*, la preuve de la première prémisse nécessite la définition d'un variant $V(v)$ et la démonstration des deux conditions suivantes pour tout événement *evt* de la machine M :

- lorsque la machine est dans un état vérifiant la propriété P et si l'événement *evt* est activable, alors $V(v)$ est non nul dans le cas où le variant est un entier naturel ou un ensemble non vide dans le cas où le variant est un ensemble.
- une exécution de l'événement *evt* à partir d'un état vérifiant la propriété P doit nécessairement décrémenter le variant $V(v)$.

3.1.5.3 Propriété de persistance

La propriété de persistance spécifie que la propriété P finit par rester définitivement vraie. Formellement, elle est exprimée pour une machine M comme suit :

$$M \vdash \Diamond\Box P$$

La règle de preuve $LIVE_{\Diamond\Box}$ (donnée ci-dessous) identifie les conditions nécessaires pour prouver la propriété de persistance au moyen des propriétés de divergence et de non blocage.

$$\frac{M \vdash \nearrow P \quad M \vdash \circlearrowleft \neg P}{M \vdash \Diamond\Box P}\ LIVE_{\Diamond\Box}$$

Dans cette règle de preuve, la première prémisse dénote le fait que M est divergente en P. Elle garantit, pour une trace ϕ de la machine M, que ϕ se termine avec une séquence infinie d'états vérifiant la propriété P (la divergence). La deuxième prémisse assure que dans le cas où ϕ est finie, elle ne vas pas se bloquer sur un état où la propriété P n'est pas vraie. Ainsi, les deux prémisses garantissent que ϕ (qu'elle soit finie ou infinie) va finir par être dans un état vérifiant la propriété P.
Afin de prouver la première prémisse avec l'outil *Rodin*, il est nécessaire de définir un variant $V(v)$ et de démontrer les trois conditions suivantes pour tout événement *evt* de la machine M :

- lorsque la machine est dans un état vérifiant la propriété $\neg P$ et si l'événement *evt* est activable, alors $V(v)$ est non nul dans le cas où le variant est un entier naturel ou un ensemble non vide dans le cas où le variant est un ensemble fini.
- une exécution de l'événement *evt* à partir d'un état vérifiant la propriété $\neg P$ doit nécessairement décrémenter le variant $V(v)$.
- une exécution de l'événement *evt* à partir d'un état vérifiant la propriété P ne doit pas augmenter le variant $V(v)$.

3.1.5.4 Propriété de progression

La propriété de progression ($M \vdash \Box(P_1 \Rightarrow \Diamond P_2)$) permet d'indiquer que l'apparition d'un état vérifiant une propriété P_1 est toujours suivie par l'apparition d'un état vérifiant une propriété P_2 dans le futur. Cette propriété est exprimée également par l'opérateur *Leads*

to noté $\rightsquigarrow$. Ainsi, $\Box(P_1 \Rightarrow \Diamond P_2)$ est équivalente à $P_1 \rightsquigarrow P_2$.
La règle de preuve $LIVE_{progress}$ permet de prouver qu'une machine M vérifie la propriété de progression en utilisant la propriété d'invariance et l'opérateur *jusqu'à* ($\mathcal{U}$).

$$\frac{\begin{array}{l} M \vdash \Box(P_1 \wedge \neg P_2 \Rightarrow P_3) \\ M \vdash \Box(P_3 \Rightarrow (P_3 \,\mathcal{U}\, P_2)) \end{array}}{M \vdash \Box\,(P_1 \Rightarrow \Diamond\, P_2)}\; LIVE_{progress}$$

La première prémisse indique qu'à chaque événement, déclenché à partir d'un état vérifiant $P_1 \wedge \neg P_2$, la machine va évoluer vers un état dans lequel une propriété intermédiaire P_3 sera vraie. Cette prémisse peut être décrite directement dans la machine M à l'aide d'un invariant. La deuxième prémisse indique que la machine va se trouver dans un état vérifiant la propriété P_3 jusqu'à ce que la propriété P_2 soit vérifiée. Cette prémisse utilisant l'opérateur $\mathcal{U}$ est prouvée à l'aide de la règle $Until$ donnée ci-dessous.

$$\frac{\begin{array}{l} M \vdash \Box((P_3 \wedge \neg P_2) \curvearrowright (P_3 \vee P_2)) \\ M \vdash \Box\Diamond(\neg P_3 \vee P_2) \end{array}}{M \vdash \Box(P_3 \Rightarrow (P_3 \,\mathcal{U}\, P_2))}\; Until$$

Dans cette règle, la première prémisse exprimée à l'aide de l'opérateur *Leads from* $\curvearrowright$ indique que chaque événement amène la machine M d'un état vérifiant la propriété $P_3 \wedge \neg P_2$ à un état vérifiant la propriété $P_3 \vee P_2$. La deuxième prémisse est une propriété d'existence qui est prouvée à l'aide de la règle $LIVE_{\Box\Diamond}$ définie dans la section 3.1.5.2.

L'interprétation d'une machine à l'aide des automates à états/transitions a permis d'exprimer les propriétés désirées du système à l'aide de la logique *LTL*. Certes, la preuve de ces propriétés est possible en utilisant des concepts de *B-événementiel* tels que la *terminaison*, les *invariants* et le *non blocage* mais elle devient compliquée avec des modèles plus difficiles. Pour rendre la preuve de ces propriétés plus explicite, nous avons recours à une logique temporelle, qui étend la logique *LTL* en ajoutant la modalité des actions, appelée la logique temporelle des actions ou *TLA* (Temporal Logic of Actions). L'utilisation de *TLA* avec *B-événementiel* n'est pas nouvelle. Dans [Mosbahi et Jemni Ben Ayed, 2006] et [Méry et Poppleton, 2013], la combinaison de ces deux formalismes a servi pour prouver des propriétés de vivacité. La logique *TLA* est présentée dans la section suivante.

3.2 La logique TLA

La logique *TLA* combine la logique temporelle et la logique des actions pour la spécification et le raisonnement sur des systèmes discrets concurrents et réactifs [Lamport, 1994]. Sa syntaxe est basée sur quatre éléments :

- des **constantes**, des **fonctions** et **des prédicats**,
- des **formules d'état** définies pour le raisonnement sur les états, exprimées sur les variables et les constantes,
- des **formules d'action** pour raisonner sur les paires d'états (états avant et après) et

- des **prédicats temporels** pour raisonner sur les séquences d'états ; ces derniers sont construits à partir des éléments déjà définis et certains opérateurs temporels.

Nous terminons cette description de cette logique en détaillant le concept "*d'étape de bégaiement*" (*stuttering step*), les contraintes d'équité ainsi que quelques règles de preuve pour *TLA*.

Étape de bégaiement. Une étape de bégaiement pour une action A et un vecteur de variables f, a lieu lorsque soit l'action A se produit soit les valeurs des variables de f restent inchangées. Nous définissons l'opérateur de bégaiement noté $[A]_f$ comme : $[A]_f \triangleq A \vee (f' = f)$ [3]. De manière duale, $\langle A \rangle_f$ indique que l'action A se produit et au moins une variable dans f change de valeur. $\langle A \rangle_f \triangleq A \wedge (f' \neq f)$.

Contraintes d'équité. L'équité indique que si une action est activable, elle finira par s'exécuter. Deux types d'équité peuvent être distingués selon que l'action est constamment activable ou qu'elle est souvent activable :

- **l'équité faible** pour l'action A notée $WF_f(A)$; affirme que si A est constamment activable, il est alors garantit que toujours, dans le futur A s'exécutera.
- **l'équité forte** pour l'action A notée $SF_f(A)$; affirme que si A est souvent activable dans le futur, il est alors garantit que toujours, dans le futur A s'exécutera.

Formellement $WF_f(A)$ et $SF_f(A)$ sont définis comme le montre la figure 3.9. Le prédicat $Enabled\langle A \rangle_f$ affirme que l'action A est activable.

$$WF_f(A) \triangleq \Diamond\Box Enabled\langle A \rangle_f \Rightarrow \Box\Diamond\langle A \rangle_f$$
$$SF_f(A) \triangleq \Box\Diamond Enabled\langle A \rangle_f \Rightarrow \Box\Diamond\langle A \rangle_f$$

Figure 3.9 — Équité faible et équité forte

Les règles de preuve pour *TLA*. Nous considérons les trois règles de preuve : $WF1$ et $SF1$ données par la figure 3.10 et $LATTICE$.
Dans les règles $WF1$ et $SF1$, P et Q sont des prédicats, A une action et N dénote une disjonction d'actions. P' (respectivement Q') est le prédicat P (respectivement Q) obtenu en remplaçant les variables qu'il contient par leurs nouvelles valeurs après l'exécution d'une action parmi N. La règle $WF1$ donne les conditions dans lesquelles une hypothèse d'équité faible pour l'action A est suffisante pour prouver $P \rightsquigarrow Q$. La condition $WF1.1$ décrit une étape pouvant donner lieu, soit à un état vérifiant P, soit un état vérifiant Q. La condition $WF1.2$ décrit l'étape inductive où $\langle A \rangle_f$ produit un état vérifiant Q. La condition $WF1.3$ assure que $\langle A \rangle_f$ est toujours activable. La règle $SF1$ donne les conditions nécessaires pour prouver $P \rightsquigarrow Q$ en supposant une équité forte. Les deux premières conditions sont similaires à $WF1$. La troisième condition assure que $\langle A \rangle_f$ va finir par être activable.

3. f' représente les nouvelles valeurs du vecteur f

$$\frac{\begin{array}{ll} WF1.1 & P \wedge [N]_f \Rightarrow (P' \vee Q') \\ WF1.2 & P \wedge \langle N \wedge A \rangle_f \Rightarrow Q' \\ WF1.3 & P \Rightarrow Enabled\langle A \rangle_f \end{array}}{\Box[N]_f \wedge WF_f(A) \Rightarrow P \rightsquigarrow Q} \; WF1$$

$$\frac{\begin{array}{ll} SF1.1 & P \wedge [N]_f \Rightarrow (P' \vee Q') \\ SF1.2 & P \wedge \langle N \wedge A \rangle_f \Rightarrow Q' \\ SF1.3 & \Box P \wedge \Box[N]_f \Rightarrow \Diamond Enabled\langle A \rangle_f \end{array}}{\Box[N]_f \wedge SF_f(A) \Rightarrow P \rightsquigarrow Q} \; SF1$$

Figure 3.10 — Règles de preuve $WF1$ et $SF1$ de TLA

$$\frac{F \wedge (c \in S) \Rightarrow (H_c \rightsquigarrow (G \vee \exists d \in S.(c \succ d) \wedge H_d))}{F \Rightarrow ((\exists c \in S.H_c) \rightsquigarrow G)} \; LATTICE$$

Figure 3.11 — Règle $LATTICE$ de TLA

La règle $LATTICE$ (figure 3.11) est une règle de preuve inductive. Dans cette règle, F, G, H_c et H_d dénotent des formules TLA, S représente un ensemble donné et $\succ$ est une relation d'ordre partiel bien formée sur l'ensemble S. Informellement, cette règle signifie que pourvu qu'à partir d'un état vérifiant la formule H_c, il est possible de passer à un état vérifiant la formule G ou à un état dans lequel la formule H_d est vérifiée pour une valeur d strictement inférieure à c, il est garantie par induction que la formule G sera atteinte.

Deuxième partie

Contributions au développement formel des SMA auto-organisateurs

4 Une modélisation structurée en niveaux des SMA auto-organisateurs

AFIN de raisonner de manière rigoureuse sur l'auto-organisation, nous commençons dans la partie 4.1 par définir formellement ce qu'est un agent et ce qu'est un SMA auto-organisateur. Inspirée des niveaux d'observation identifiés par Zambonelli dans [Zambonelli et Omicini, 2004], notre formalisation est structurée en trois niveaux d'abstraction : le niveau **micro** correspond aux comportements locaux des entités du système –les agents (le SMA) et l'environnement– considérés unilatéralement (section 4.1.1), le niveau **méso** correspond à la description des interactions SMA/environnement (section 4.1.2) et le niveau **macro** décrit le comportement du système dans sa globalité observé par un observateur externe (section 4.1.3). Cette formalisation est ensuite exprimée dans la section 4.2 à l'aide du langage *B-événementiel* qui bénéficie de la plateforme *Rodin* pour automatiser le maximum de preuves.

4.1 Formalisation d'un SMA auto-organisateur

Un système multi-agent est constitué d'un ensemble d'agents plongés dans un environnement. Au niveau micro, ces deux éléments sont décrits de manière unilatérale. A un très haut niveau d'abstraction, le comportement de chaque agent se résume au cycle de vie suivant : l'agent détecte des informations de l'environnement (la perception), prend une décision en fonction de ses perceptions (la décision) et enfin exécute l'action choisie (l'action). Nous nous référons à ces étapes par le cycle *percevoir-décider-agir*.

Un agent est décrit par ses états ainsi que les transitions entre ses états. L'état d'un agent est défini par l'ensemble des représentations qu'il possède sur son environnement (A_{rep}), ses propriétés (A_{prop}) telles que la représentation qu'il a sur lui-même, l'état de ses capteurs (sensors) (A_{sens}) et l'état de ses actionneurs (A_{act}).
L'évolution de l'état d'un agent est due à trois types de transition :

- l'ensemble des opérations qui lui permettent de mettre à jour ses représentations ($A_{perceive}$),
- l'ensemble des règles pour prendre ses décisions (A_{decide}),
- l'ensemble des actions qu'il peut effectuer ($A_{perform}$). Les actions qu'un agent peut accomplir peuvent avoir comme conséquences, le changement des propriétés internes à l'agent, dans ce cas elles sont dites internes et on les note $A_{performInt}$ ou le changement de la partie locale de l'environnement que l'agent est capable de percevoir et on les note

$A_{performAlterEnv}$. Les deux ensembles $A_{performInt}$ et $A_{performAlterEnv}$ doivent être disjoints et vérifient la propriété $A_{performInt} \cap A_{performAlterEnv} = \varnothing$.

4.1.1 Formalisation du niveau micro

Au niveau micro, nous nous intéressons uniquement aux décisions que l'agent est capable de prendre. Nous faisons abstraction de la manière avec laquelle il acquiert les perceptions dont il a besoin ainsi que les actions qui résultent de ses décisions. Nous considérons que ces aspects font partie des interactions du SMA avec son environnement, ils seront pris en compte dans le niveau méso. Nous définissons dans ce qui suit ce qu'est un agent, un SMA et finalement ce qu'est un environnement de manière formelle.
Formellement, un agent est décrit par un automate comme le montre la définition 4.

Définition 4. *Un agent est un automate* $A = (SA, SA_{init}, TA, \delta A)$ *avec*

- SA *est l'ensemble des états de l'agent.*
  $SA = A_{prop} \times A_{act}$
- $SA_{init} \in SA$ *dénote l'état initial de l'agent.*
- TA *est un ensemble d'étiquettes représentant les règles de décision de l'agent.*
  $TA = A_{decide}$
- δA *est l'ensemble des transitions possibles entre les différents états de l'agent.*
  $\delta A \subseteq SA \times TA \times SA$

Formellement, chaque transition est exprimée par un prédicat de transition mettant en relation des variables primées et non primées. Les variables non primées représentent l'état de l'agent avant la transition. Tandis que les variables primées correspondent à l'état de l'agent après la transition. Le prédicat $t(a, p, p')$ signifie que la transition t fait passer l'agent a de l'état vérifiant la proposition p vers un état vérifiant la proposition p'.
Lorsqu'un agent prend une décision, ceci se traduit par une modification de l'état de ses propriétés ou l'activation de ses actionneurs afin de les préparer à exécuter l'action choisie. Les décisions d'un agent a sont formalisées par un ensemble de prédicats de transition noté $Decide(a, A_{prop} \cup A_{act}, A'_{prop} \cup A'_{act})$.
Le SMA formé de n agents $A_1, ..., A_n$ est décrit formellement par l'automate MAS résultat de la composition parallèle $A_1 \parallel A_2, ... \parallel A_n$ et défini comme suit :

Définition 5. *Un SMA est un automate* $MAS = (S\_MAS, S\_MAS_{init}, T\_MAS, \delta\_MAS)$ *avec*

- S_MAS *représente l'ensemble des états possibles du SMA donné par le produit cartésien des états des agents qui le composent.*
  $S\_MAS = \prod_{i:1..n} SA_i$ *dénote l'ensemble des états du SMA.* SA_i *représente l'ensemble des états de l'agent* A_i.
- $S\_MAS_{init} = \prod_{i:1..n} SA_{i,init}$ *est l'état initial du SMA.*
  $S\_MAS_{init} \in S\_MAS$

- *T_MAS dénote l'ensemble des étiquettes des transitions entre les états du SMA. Les étiquettes au niveau micro sont limitées aux règles décisionnelles des agents.*
  $T\_MAS = \bigcup_{i:1..n} A_{i,decide}$
- *δ_MAS est l'ensemble des transitions possibles entre les différents états du SMA.*
  $\delta\_MAS \subseteq S\_MAS \times T\_MAS \times S\_MAS$

Nous considérons que l'environnement est composé d'un ensemble de m éléments notés $l_1, ...l_m$. L'état de l'environnement est décrit par l'état de ses différents éléments. On note par E_{change} les actions de l'environnement ayant comme conséquences des changements sur ses éléments. On note par $EnvironmentChange(E, l, l')$ l'ensemble des prédicats de transition modélisant les changements qui peuvent avoir lieu dans l'environnement.
Formellement, l'environnement est décrit à l'aide d'un automate comme le montre la définition 6.

Définition 6. *Un environnement est un automate* $E = (SE, SE_{init}, TE, \delta E)$ *avec*

- *SE est l'ensemble des états de l'environnement.*
  $SE = \prod_{i:1..m} Sl_i$ *avec* Sl_i *représente l'état de l'élément* l_i
- $SE_{init} \in SE$ *dénote l'état initial de l'environnement.*
- *TE est un ensemble d'étiquettes représentant les actions de l'environnement.*
  $TE = E_{change}$
- *δE est l'ensemble des transitions possibles entre les états de l'environnement.*
  $\delta E \subseteq SE \times TE \times SE$

4.1.2 Formalisation du niveau méso

Au niveau méso, nous nous intéressons aux interactions SMA/environnement. Les agents interagissent ensemble via l'environnement. Ces interactions sont formalisées par le biais d'une composition parallèle d'automates. Généralement, la composition de deux automates implique une éventuelle synchronisation sur les actions ayant les mêmes noms [1]. Ainsi, pour éviter qu'il y ait des synchronisations sur des actions propres à l'agent ou propres à l'environnement, il faut s'assurer qu'aucune action propre à l'agent ne figure parmi les actions propres à l'environnement et inversement. Pour un ensemble d'agents $A_1, A_2, ...A_n$ et un environnement E, cette contrainte est exprimée formellement comme suit : la composition $A_1 \parallel A_2, ... \parallel A_n \parallel E$ est réalisable si $\forall i \in 1..n \cdot (A_{i,performInt} \cap TE) = \varnothing$.
La composition du SMA avec l'environnement donne lieu à un automate noté $SYSTEM$ décrivant le système et défini comme suit :

Définition 7. *Un système auto-organisateur est un automate* $SYSTEM = (S, S_{init}, T, \delta)$ *avec*

1. La synchronisation sur des actions signifie que ces actions sont exécutées simultanément

- *S dénote l'ensemble des états du système. Il est donné par le produit cartésien du SMA, de l'environnement du système, des représentations ($\prod_{i:1..n} A_{i,rep}$) et des états des capteurs des agents ($\prod_{i:1..n} A_{i,sens}$).*
  $S = S\_MAS \times \prod_{i:1..n} A_{i,rep} \times \prod_{i:1..n} A_{i,sens} \times SE$
- *S_{init} est l'état initial du système. $S_{init} = S\_MAS_{init} \times SE_{init}$ avec $S_{init} \in S$*
- *T est l'ensemble des étiquettes des transitions sur les états du système. Ces transitions sont formées par les décisions des agents (T_{MAS}), les transitions de l'environnement (TE), les opérations de perceptions des agents ($\bigcup_{i:1..n} A_{i,perceive}$) et les actions des agents ($\bigcup_{i:1..n} A_{i,performInt} \cup \bigcup_{i:1..n} A_{i,performAlterEnv}$).*
  $T = \bigcup_{i:1..n} A_{i,perceive} \cup T\_MAS \cup \bigcup_{i:1..n} A_{i,performInt} \cup \bigcup_{i:1..n} A_{i,performAlterEnv} \cup TE$
- *δ dénote les transitions possibles entre les états du système. $\delta \subseteq S \times T \times S$*

Les modifications qui peuvent avoir lieu sur l'état d'un agent *a* doivent être conformes aux règles suivantes :

- ▷ les opérations de perception sont responsables de la modification uniquement des représentations partielles de l'agent. Cette modification est décrite par le prédicat de transition $Perceive(a, A_{rep}, A'_{rep})$. A_{rep} et A'_{rep} dénotent l'état des représentations de l'agent *a* respectivement avant et après l'exécution d'une transition de l'ensemble $A_{perceive}$.
- ▷ les actions d'un agent ont comme conséquences la modification de l'état de ses propriétés et l'activation de ses capteurs afin de le préparer à la prochaine étape de perception. Les actions de chaque agent *a* sont aussi considérées comme un ensemble de prédicats de transition noté $Perform(a, A_{prop} \cup A_{sens}, A'_{prop} \cup A'_{sens})$.

Le comportement local des agents décrit précédemment est dit "correct", si les propriétés suivantes sont satisfaites.

- ▷ LocProp1 : le comportement de chaque agent est conforme au cycle *percevoir-décider-agir*.
- ▷ LocProp2 : l'agent ne doit pas être bloqué dans l'étape de décision, c'est-à-dire que la décision doit lui permettre de réaliser une action.
- ▷ LocProp3 : l'agent ne doit pas être bloqué dans l'étape de perception ; c'est-à-dire que les représentations interprétées doivent lui permettre de prendre une décision.

L'exécution du système *SYSTEM* est une séquence d'états. Chaque état est décrit par la valuation des variables des agents et de l'environnement. L'évolution de l'état du système se fait en deux temps. Premièrement, tous les agents exécutent un cycle *percevoir-décider-agir*. Cette exécution peut faire évoluer l'état des agents ou encore l'état de l'environnement du système. Une *transition des agents* désigne l'évolution des états des agents ainsi que l'environnement suite à l'accomplissement d'un cycle *percevoir-décider-agir* par tous les agents.

Suite à une *transition des agents*, l'environnement du système prend le relais et fait une transition en exécutant une action. L'évolution de l'état de l'environnement suite à cette action forme une *transition de l'environnement du système*.
Une *transition du système* est composée d'une transition des agents suivie d'une transition de l'environnement comme le montre la figure 4.1. Les transitions δi, δA et δE dénotent respectivement des transitions du système, des transitions des agents et des transitions de l'environnement. Durant la première étape, l'exécution parallèle des agents se fait par l'opé-

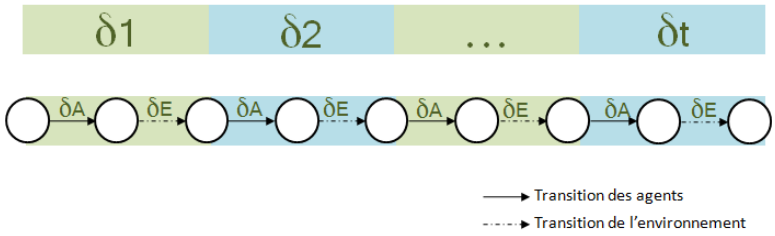

Figure 4.1 — Les transitions du système modélisées par une transition des agents suivie par une transition de l'environnement.

rateur de composition $\|$. Cette composition signifie qu'à un instant bien déterminé, l'un des agents effectue une étape du cycle *percevoir-décider-agir*. Les transitions effectuées par un agent donné doivent respecter les étapes du cycle *percevoir-décider-agir*. Entre deux transitions d'un agent particulier, il peut y avoir plusieurs transitions d'autres agents. Ceci est illustré par la figure 4.2.

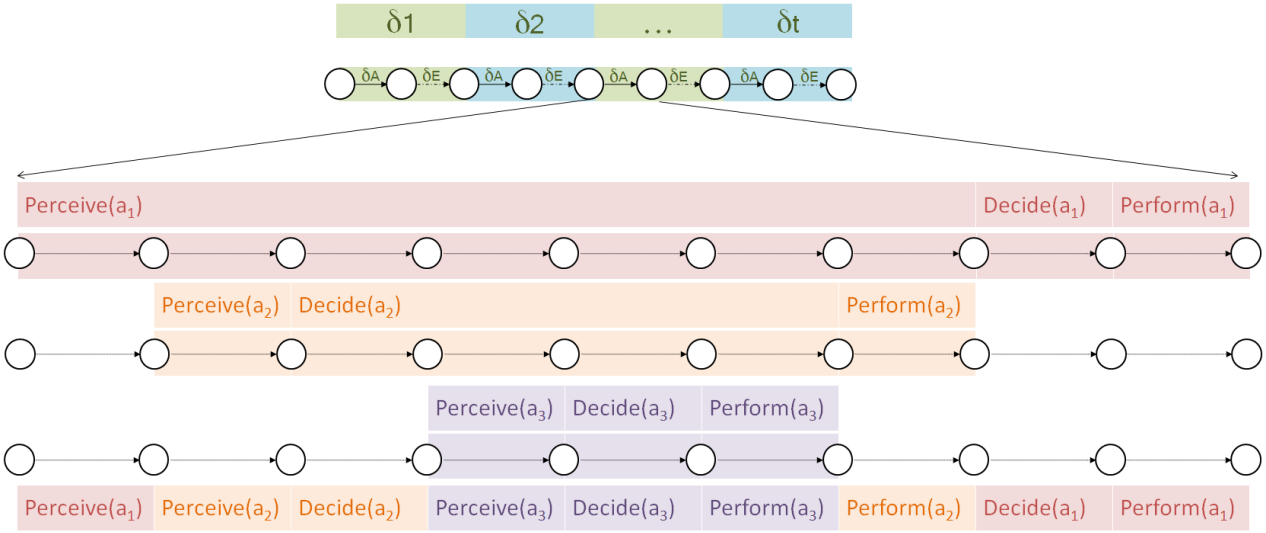

Figure 4.2 — Une transition de trois agents.

4.1.3 Formalisation du niveau macro

Au niveau macro, le focus est mis sur la modélisation de l'**observation** de l'évolution de l'état macroscopique du système en fonction des actions locales des agents, de leurs interactions et des changements de l'environnement. Cette modélisation est une base pour prouver

formellement des propriétés globales. L'état macroscopique du système est décrit à l'aide d'un agrégat des états de ses éléments (les agents et l'environnement). Ainsi, le nombre d'agent se trouvant dans un état particulier ou le nombre d'agent exhibant un comportement bien déterminé peuvent décrire l'état macroscopique du système.
Le système peut être soit dans un état fonctionnellement adéquat qui qualifie une situation dans laquelle il assure la fonction pour laquelle il a été conçu, soit dans un état qui nécessite l'emploi de ses mécanismes d'auto-organisation pour faire face aux perturbations et se remettre à nouveau dans un état fonctionnellement adéquat. La figure 4.3 résume les états du cycle de vie d'un SMA auto-organisateur observé par un observateur externe.

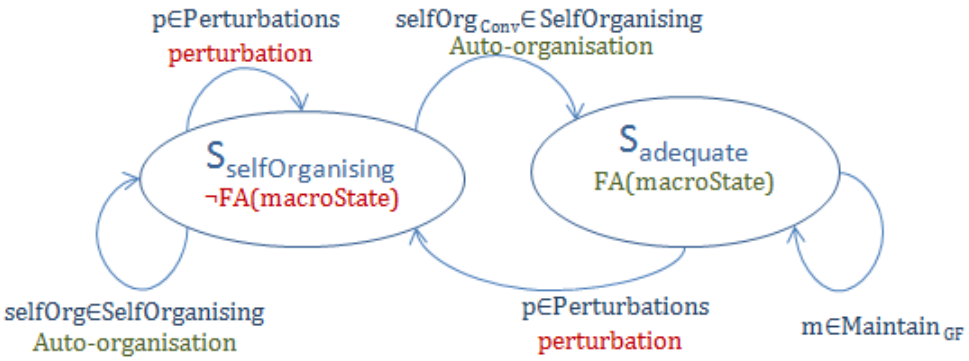

Figure 4.3 — Représentation abstraite du comportement du système au niveau macro

On note par $S_{adequate}$ un état fonctionnellement adéquat dans lequel le système peut se trouver dès son initialisation ou auquel il doit converger lorsqu'il n'est soumis à aucune perturbation. On note par $FA(macroState)$ le prédicat défini en fonction de l'état macroscopique du système $macroState$ et décrivant un état fonctionnellement adéquat. Dés que la fonctionnalité du système est perturbée, il passe à l'état $S_{selfOrganising}$ dans lequel il emploie ses mécanismes d'auto-organisation pour assurer à nouveau sa fonction.
Dans le cadre de ce travail, nous nous intéressons aux SMA se basant sur la coopération comme un moyen pour l'auto-organisation. Par conséquent, en se référant au théorème de l'adéquation fonctionnelle de la théorie des AMAS, le système se trouve dans un état fonctionnellement adéquat lorsque tous les agents sont coopératifs. Ainsi, les états $S_{adequate}$ et $S_{selfOrganising}$ peuvent être décrits en fonction du nombre d'agents se trouvant dans une situation non coopérative. Si on dénote par $Agents_{NCS}$ l'ensemble des agents se trouvant dans une SNC et par $card(Agents_{NCS})$ le nombre d'agents dans cet ensemble, on a $S_{adequate} \equiv card(Agents_{NCS}) = 0$ et $S_{selfOrganising} \equiv card(Agents_{NCS}) > 0$
Les transitions observées entre ces différents états sont dues à des changements pouvant être classés parmi les ensembles de changements suivants :

- $Maintain_{GF}$ est formé par les changements n'ayant aucun effet sur la fonctionnalité adéquate du système.
- $Perturbations$ est l'ensemble des changements qui perturbent le fonctionnement du système et l'empêchent d'assurer sa fonction.
- $SelfOrganising$ regroupe les actions que le système peut entreprendre afin de retrouver un état fonctionnellement adéquat.

Formellement, le SMA auto-organisateur vu par un observateur externe peut être défini comme suit :

Définition 8. *Un système multi-agent observé au niveau macro est un automate* $SYSTEM_{MACRO} = (GS, GS_{init}, GT, G\delta)$ *avec*

- $GS = S_{adequate} \cup S_{selfOrganising}$
  avec $S_{adequate} \cap S_{selfOrganising} = \varnothing$
- GS_{init} *est l'état initial du système.*
  $GS_{init} \in GS$
- $GT = Maintain_{GF} \cup Perturbations \cup SelfOrganising$
  avec $Maintain_{GF} \cap Perturbations \cap SelfOrganising = \varnothing$
- $G\delta \subseteq GS \times GT \times GS$

Nous supposons que les transitions $Maintain_{GF}$ et $Perturbations$ peuvent être définies aussi bien à partir des actions des agents que celles de l'environnement. Ainsi, ces deux transitions vérifient les propriétés suivantes : $Maintain_{GF} \subseteq T$ et $Perturbations \subseteq T$. Les actions d'auto-organisation doivent résulter des seules actions des agents. Ainsi, les transitions $SelfOrganising$ vérifient la proposition $SelfOrganising \subseteq T \setminus TE$.

Le comportement observé du système est défini par une séquence (pouvant être infinie) alternant des états et des actions $gs_0 \xrightarrow{gt_1} gs_1 \xrightarrow{gt_2} gs_2$... où pour tout $i > 0$, $gt_i \in GT$ tel que $(gs_{i-1}, \xrightarrow{gt_i}, gs_i) \in G\delta$. On note par $\varepsilon(GS)$ l'ensemble de toutes les traces observables du système. On note par $state(\epsilon, i)$ l'état observable du système à l'instant i dans la trace $\epsilon \in \varepsilon(GS)$.
Les définitions données ci-dessus vont servir de base pour spécifier formellement les propriétés de convergence et de résilience. Nous adoptons les définitions données par G. de Marzo Serugendo dans [Serugendo, 2009] pour la convergence et la résilience dans le cadre des systèmes auto-organisateurs.
Au niveau macro, nous nous intéressons à prouver les propriétés de convergence et de résilience du SMA conçu. Ces deux propriétés sont définies dans les paragraphes qui suivent.

La convergence du SMA La convergence ([Serugendo, 2009]) indique la capacité du système à atteindre son objectif en l'absence de perturbations. Le système converge soit lorsque dès l'initialisation il se trouve dans un état fonctionnellement adéquat soit on garantit qu'il arrive à atteindre un état fonctionnellement adéquat après un certain nombre de transitions. Formellement, la convergence du système $SYSTEM_{MACRO}$ se traduit par la formule suivante :

$$GS_{init} \in S_{adequate} \vee (\forall \epsilon \in \varepsilon(SYSTEM_{MACRO}) \cdot \exists i \cdot state(\epsilon, i) \in S_{adequate}).$$

La résilience du SMA La résilience ([Serugendo, 2009]) décrit la capacité du système à s'adapter aux changements et aux perturbations qui peuvent avoir lieu. L'analyse de la résilience permet d'évaluer la capacité des mécanismes d'auto-organisation à rétablir l'état du système après des perturbations sans détecter explicitement une erreur.
On note par p une perturbation provenant de l'environnement ou des agents et provoquant un ensemble ψ de transitions ($\psi \subseteq GS \times p \times GS$). Cette perturbation fait passer le système d'un état fonctionnellement adéquat vers un état d'auto-organisation. Le système est dit résilient à la perturbation p s'il est capable de retrouver un état fonctionnellement adéquat après cette perturbation. On note par $\varepsilon(SYSTEM_{MACRO}, gs)$ les traces observables du système à partir de l'état gs.
Formellement, le système $SYSTEM_{MACRO} = (GS, GS_{init}, GT, G\delta)$ est dit résilient par rapport à la perturbation p si pour toute trace $\epsilon \in \varepsilon(SYSTEM_{MACRO})$ tel qu'il existe $i \geqslant 0$ pour lequel $state(\epsilon, i) = gs$ et pour tout état d'auto-organisation gs' vérifiant $(gs, p, gs') \in \psi$, il vérifie la formule suivante :

$$\forall \epsilon' \in \varepsilon(SYSTEM_{MACRO}, gs') \cdot \exists j > 0 \cdot state(\epsilon', j) \in S_{adequate}.$$

4.2 Raisonner sur l'auto-organisation à l'aide de *B-événementiel* et la logique temporelle

Les automates à états/transitions ont fourni un cadre pratique et intuitif pour décrire une formalisation des SMA auto-organisateurs et exprimer les propriétés à prouver formellement. Cette formalisation est traduite en *B-événementiel*. Ainsi, en utilisant les règles de preuve décrites dans la section 3.1.5, il est possible d'automatiser la preuve des propriétés désirées du système au moyen de la plateforme *Rodin*. Dans cette section, nous décrivons la modélisation définie dans ce travail de thèse pour la modélisation des niveaux micro et méso et nous montrons comment les propriétés du niveau macro peuvent être prouvées à l'aide de la logique *TLA* et la logique *LTL*.

4.2.1 Formalisation du niveau micro

Au niveau micro, nous nous intéressons à modéliser le comportement de l'environnement ainsi que les règles décisionnelles des agents. Avec *B-événementiel*, le niveau micro est formalisé par la machine *MicroLevel* (figure 4.5) et son contexte *ContextMicro* (figure 4.4) détaillés dans les deux paragraphes qui suivent.

Le modèle statique : le contexte *ContextMicro* Le contexte *ContextMicro* définit l'ensemble d'agents *Agents* et l'ensemble *Steps* dont les éléments correspondent aux trois étapes du cycle d'un agent (*perceive*, *decide*, *perform*). Ce contexte définit également l'ensemble *Modes* dont les éléments correspondent aux deux modes dans lesquels un agent peut être (*work*, *wait*). Un agent est en mode *work* lorsqu'il est entrain d'exécuter un cycle *percevoir-décider-agir*. Lorsque l'agent termine les trois étapes de son cycle d'activité, il passe au mode *wait*. Tant qu'il existe des agents en mode *work*, ceci signifie qu'une *transition des agents* est

```
context ContextMicro
sets Agents  Steps  Modes  Actions  Properties  Activity

constants perceive  decide  perform  wait  work  DecisionProperties
          enable  disable
axioms
 @axm1 finite(Agents)
 @axm2 partition(Steps,{perceive},{decide},{perform})
 @axm3 partition(Modes,{wait},{work})
 @axm4 partition(Activity,{enable},{disable})
 @axm5 partition(Properties,DecisionProperties)
end
```

Figure 4.4 — Le modèle statique du niveau micro : le contexte *ContextMicro*

en cours. Lorsque tous les agents sont en mode *Wait*, c'est au tour de l'environnement d'exécuter une transition.
Dans le contexte *ContextMicro*, les ensembles *Actions* et *Properties* représentent respectivement les différentes actions résultant des décisions et les différentes propriétés des agents. Au niveau micro, seules les propriétés qui sont modifiées par les règles décisionnelles des agents sont spécifiées. Ainsi, l'axiome *@axm*5 partitionne l'ensemble *Properties* en un seul sous-ensemble noté *DecisionProperties*. L'axiome *@axm*4 définit les éléments de l'ensemble *Activity* : *enable* et *disable* utilisés pour qualifier les deux modes d'activité des actionneurs et des capteurs des agents.

Le modèle dynamique : la machine *MicroLevel* La machine *MicroLevel* (figure 4.5) modélise de manière très abstraite un SMA situé dans un environnement sans préciser la manière avec laquelle ces deux éléments interagissent.
Les variables *agentStep* et *agentMode* dénotent respectivement l'étape du cycle de vie et le mode d'un agent. Les variables *nextAgentAction* et *agentDecisionProp* contiennent les résultats du processus de décision de chaque agent. En effet, *nextAgentAction* dénote l'action qu'un agent a choisi lorsqu'il prend sa décision. La variable *agentDecisionProp* représente les propriétés qui sont modifiées par sa décision et nécessaires pour la réalisation de l'action choisie. Un exemple étant la mémorisation de la position vers laquelle une fourmi a choisi de se déplacer.
Initialement, tous les agents sont en mode *work* et se trouvent dans la phase *perceive* (l'action de l'événement $INITIALISATION$ à la figure 4.5). Les événements *Perceive* et $Perform$ correspondent respectivement aux étapes de perception et d'exécution de l'action choisie. Comme le montre la figure 4.6, la seule action réalisée par chacun de ces événements est de faire passer un agent vers l'étape suivante de son cycle de vie garantissant ainsi la propriété *LocProp*1 (le comportement de chaque agent est conforme au cycle *percevoir-décider-agir*).
Le seul événement détaillé à ce niveau est l'événement *DecideAct* qui modélise l'ensemble des règles décisionnelles des agents. L'événement *EnvironmentChange* (figure 4.7) décrit une transition de l'environnement. Sa garde permet de vérifier que tous les agents ont fini

```
machine MicroLevel
sees ContextMicro
variables agentStep  agentMode  nextAgentAction agentDecisionProp  agentActuators
Invariants
@inv1 agentStep ∈ Agents → Steps
@inv2 agentMode ∈ Agents → Modes
@inv3 nextAgentAction ∈ Agents → Actions
@inv4 agentDecisionProp ∈ Agents → DecisionProperties
@inv5 agentActuators ∈ Agents × Actions → Activity
events
 event INITIALISATION
  then
   @act1 agentMode, agentStep , nextAgentAction, agentDecisionProp ,agentActuators :|
          agentMode'=(Agents×{work}) ∧
          agentStep'=Agents×{perceive} ∧
          nextAgentAction'∈(Agents×Actions) ∧
          agentDecisionProp ' ∈(Agents×DecisionProperties) ∧
          agentActuators ' =(Agents×{disable})
  end
 event Perceive
 event DecideAct
 event Perform
 event EnvironmentChange
end
```

Figure 4.5 — Le modèle dynamique du niveau micro : la machine *MicroLevel*

un cycle *percevoir-décider-agir*. La seule action qu'il effectue à ce niveau d'abstraction est de donner la main aux agents pour commencer une nouvelle transition. Au niveau micro, nous nous intéressons également aux actions de l'environnement : les actions qui modifient les éléments de l'environnement et qui ne représentent pas les conséquences des actions des agents. Dans le contexte de notre travail, nous supposons que ces actions représentent des perturbations et elles seront raffinées lorsque nous nous intéressons à prouver des propriétés globales relatives à la résilience.

4.2.2 Formalisation du niveau méso

Tout au long de son fonctionnement, un agent interagit continuellement avec son environnement (l'environnement d'un agent regroupe les éléments de l'environnement du SMA et les autres agents se trouvant dans son voisinage). Cette interaction implique une influence mutuelle entre l'agent et son environnement : l'agent peut modifier son environnement, l'environnement influence les décisions de l'agent.
Un agent interagit avec son environnement en acquérant des perceptions de son environnement local par le biais de ses capteurs et en agissant sur son environnement à l'aide de ses actionneurs.
La machine *MesoLevel* (figure 4.9), correspondant à la description du niveau méso, formalise cet aspect interactionnel entre agents et environnement en décrivant les opérations de perception ainsi que les différentes actions qu'un agent peut effectuer. La machine *MesoLevel* ainsi que le contexte *ContextMeso* (figure 4.8) qu'elle utilise sont détaillés dans les paragraphes suivants.

```
event Perceive
  any agent
  where
   @grd1 agent ∈Agents
   @grd2 agentMode (agent)=work
   @grd3 agentStep(agent)=perceive
  then
   @act1 agentStep (agent):=decide
  end
```

```
event Perform
  any agent
  where
   @grd1 agent ∈Agents
   @grd2 agentMode (agent)=work
   @grd3 agentStep(agent)=perform
  then
   @act1 agentStep (agent):=perceive
   @act2 agentMode(agent):=wait
  end
```

```
event DecideAct
  any agent
  where
   @grd1 agent ∈Agents
   @grd2 agentMode (agent)=work
   @grd3 agentStep(agent)=decide
   @grd4 act ∈Actions
  then
   @act1 agentStep (agent):=perform
   @act2 nextAgentAction(agent):=act
   @act3  agentDecisionProp(agent):|
   DecideAct(agent,agentDecisionProp(agent),agentDecisionProp'(agent))
   @act4 agentActuators(agent,act):=enabled
end
```

Figure 4.6 — Les événements *Perveive*, *DecideAct* et *Perform* de la machine *MicroLevel*

```
event EnvironmentChange
   where
    @grd1 ∀agent·agent∈Agents⇒agentMode(agent)=wait
   then
    @act1 agentMode :|  agentMode'=(Agents×{work})
end
```

Figure 4.7 — L'événement *EnvironmentChange* de la machine *MicroLevel*

```
context ContextMeso
sets EnvElements   EnvElementsProperties
constants ActionProperties  RepresentationProperties  InternalProperties
axioms
 @axm1 partition(Properties,DecisionProperties,ActionProperties,
         RepresentationProperties,InternalProperties)
end
```

Figure 4.8 — Le modèle statique du niveau meso : le contexte *ContextMeso*

Le modèle statique : le contexte *ContextMeso* Le contexte *ContextMeso* décrit par la figure 4.8 enrichit, dans l'axiome @*axm*1, l'ensemble des propriétés des agents (l'ensemble *Properties* défini dans le contexte *ContextMicro*) par les sous-ensembles :

- *ActionProperties* qui dénote l'ensemble des propriétés des agents modifiées par leurs actions,
- *RepresentationProperties* qui décrit l'ensemble des représentations partielles que les agents possèdent sur leur environnement et mises à jour par les opérations de perception et
- *InternalProperties* qui dénote l'ensemble des propriétés internes de l'agent.

Le contexte *ContextMeso* définit également l'ensemble des éléments de l'environnement (*EnvElements*). L'ensemble *EnvElementsProperties* dénote le type de ces éléments.

```
machine MesoLevel
sees ContextMeso
ariables agentStep  agentMode  nextAgentAction  agentDecisionProp  agentActuators
          agentActionProp  agentSensors  agentRep  envElementsProp

Invariants
@inv1 agentActionProp ∈ Agents → ActionProperties
@inv2 agentSensors ∈ Agents → Activity
@inv3 agentRep ∈ Agents → RepresentationProperties
@inv4 agentIntProp ∈ Agents → InternalProperties
@inv5 envElementsProp ∈ EnvElements → EnvElementsProperties
events
 event INITIALISATION
  then
 @act1 agentMode,agentStep,nextAgentAction,agentDecisionProp,agentActuators,
        agentActionProp,agentSensors, agentRep,agentIntProp,envElementsProp :|
           agentMode'=(Agents×{work})∧
           agentStep'=Agents×{perceive}∧
           nextAgentAction'∈(Agents×Actions)∧
           agentDecisionProp ' ∈(Agents×DecisionProperties)∧
           agentActuators ' =(Agents×{disable})∧
           agentActionProp' ∈(Agents×ActionProperties)∧
           agentSensors ' =(Agents×{enable})∧
           agentRep ' ∈(Agents× RepresentationProperties)∧
           agentIntProp ' ∈(Agents× InternalProperties)∧
           envElementsProp' ∈ EnvElements×EnvElementsProperties
  end
 event PerceiveEnvironment
 event DecideAct
 event PerformAct
 event EnvironmentChange
end
```

Figure 4.9 — Le modèle dynamique du niveau méso : la machine *MesoLevel*

Le modèle dynamique : la machine *MesoLevel* Dans la machine *MesoLevel* (figure 4.9), l'état des agents est défini par les variables :

- ▷ *agentActionProp* : est défini comme une fonction totale de l'ensemble des agents dans l'ensemble *ActionProperties* (@*inv*1), cette variable modélise pour chaque agent les propriétés qui sont modifiées suite à l'exécution de ses actions.

- ▷ *agentSensors* : est défini comme une fonction totale dans l'invariant @*inv*2, cette variable modélise pour chaque agent l'état d'activation de ses capteurs.

- ▷ *agentRep* : identifie pour chaque agent l'ensemble des représentations (@*inv*3).

- ▷ *agentIntProp* : indique les propriétés internes d'un agent.
  Les variables *agentSensors*, *agentRep* et *agentIntProp* évoluent grâce à l'événement *PerceiveEnvironment* donné par la figure 4.10. L'événement *PerceiveEnvironment* est

```
event PerceiveEnvironment
  any agent
  where
   @grd1 agent ∈Agents
   @grd2 agentMode (agent)=work
   @grd3 agentStep(agent)=perceive
   @grd4 agentSensors(agent)=enable
  then
   @act1 agentStep (agent):=decide
   @act2 agentRep(agent),agentIntProp(agent):|
    Perceive(agent,agentRep(agent) ∧agentIntProp(agent),agentRep'(agent)
    ∧agentIntProp'(agent))
   @act3 agentSensors (agent):=disable
  end
```

Figure 4.10 — L'événement *PerceiveEnvironment* de la machine *MesoLevel*

  déclenché lorsqu'il existe un agent *agent* (@*grd*1) en mode *work* (@*grd*2), se trouvant dans l'étape de perception (@*grd*3) et ayant des capteurs activés (@*grd*4). L'exécution de cet événement fait passer l'agent à l'étape de décision (@*act*1), désactive ses capteurs (@*act*3) et met à jour ses représentations de l'environnement et ses propriétés internes, dans @*act*2, selon le prédicat :
  $Perceive(agent, agentRep(agent), agentIntProp(agent), agentRep'(agent), agentIntProp'(agent))$.

L'état de l'environnement est décrit dans la machine *MesoLevel* par la variable *envElementsProp* qui modélise l'ensemble des éléments de l'environnement qui seront modifiés suite aux actions des agents. L'événement $PerformAct$ modélise l'évolution de cette variable comme le montre la figure 4.11 La garde de l'événement $PerformAct$ indique la nature de l'action choisie par l'événement de décision (@*grd*4) ainsi que l'état d'activation des actionneurs responsables de cette action (@*grd*5). L'exécution de cet événement a pour conséquences la désactivation des actionneurs de l'agent *agent* (@*act*4) et la modification, dans @*act*3, de la propriété concernée selon le prédicat $PerformAct(agent, agentActionProp'(agent), envElement')$.

```
event PerformAct
  any agent
  where
   @grd1 agent ∈Agents
   @grd2 agentMode (agent)=work
   @grd3 agentStep(agent)=perform
   @grd4 nextAgentAction(agent)=act
   @act5 agentActuators(agent,act)=enabled
  then
   @act1 agentStep (agent):=perceive
   @act2 agentMode(agent):=wait
   @act3 agentActionProp(agent), envElementsProp:|
   PerformAct(agent,agentActionProp'(agent),envElementsProp')
   @act4 agentActuators(agent,act):=disable
end
```

Figure 4.11 — L'événement *PerformAct* de la machine *MesoLevel*

4.2.3 Formalisation du niveau macro

L'automate du niveau macroscopique donné dans la section 4.1.3 est modélisé en *B-événementiel* à l'aide de la machine *MacroLevel* (figure 4.12). Les variables de cette machine, représentant des variables macroscopiques et modélisées par le vecteur de variables *macroState*, renseignent sur l'état global du système observé par un observateur externe. Les événements correspondent aux actions des différents agents et celles de l'environnement faisant évoluer les variables macroscopiques. Du point de vue d'un observateur externe, ses événements sont :

- ▷ des perturbations (événement p) empêchant les agents d'assurer leurs fonctions.
- ▷ des actions d'auto-organisation permettant aux agents de neutraliser les effets des perturbations (événement $selfOrg$). Concrètement, ces actions permettent aux agents se trouvant dans des situations non coopératives de les résoudre.
- ▷ des actions d'auto-organisation faisant progresser le système vers l'état $S_{adequate}$ (événement $selfOrgConv$). Concrètement, ces actions représentent l'exécution des fonctions assurées par les agents.
- ▷ des actions maintenant le système dans l'état $S_{adequate}$ (l'événement m).

L'événement *observer* est un événement sans action jouant le rôle d'un observateur externe. Son activation permet de signaler que le système atteint un état fonctionnellement adéquat.

Les deux paragraphes suivants décrivent en détail les événements de la machine *MacroLevel* ainsi que les preuves nécessaires à l'analyse des propriétés de convergence et de résilience.

Preuve de la convergence du SMA La convergence désigne la capacité du système à atteindre un état fonctionnellement adéquat vérifiant le prédicat $FA(macroState)$ en l'absence

```
machine MacroLevel sees ContextMacro
variables macroState    FA  convProg
invariants
@inv1 FA ∈ STATES → BOOL
@inv2 macroState∈STATES
@inv3 convProg ∈ STATES → ℕ
variant convProg (macroState)
events
 event INITIALISATION
 event p
 event selfOrg
 event selfOrgConv
 event m
 event observer
    where
       @grd1 FA(macroState)=TRUE
 end
end
```

Figure 4.12 — La machine décrivant le niveau macro

de perturbations. Ainsi, dans ce paragraphe, les événements p et $selfOrg$ sont supposés sans action et par conséquent il n'ont pas d'effets sur l'évolution de la machine *MacroLevel*. En utilisant la logique de prédicats et les opérateurs temporels, cette propriété est exprimée par la formule suivante :

$$MacroLevel \vdash \Diamond\Box\, FA(macroState)$$

La preuve de cette formule se fait selon la règle $LIVE_{\Diamond\Box}$ présentée dans la section 3.1.5.3 est décrite comme suit :

$$\frac{\begin{array}{l} MacroLevel \vdash \nearrow FA(macroState) \\ MacroLevel \vdash \circlearrowleft \neg FA(macroState) \end{array}}{MacroLevel \vdash \Diamond\Box\, FA(macroState)}\ LIVE_{\Diamond\Box}$$

La première prémisse ($MacroLevel \vdash \nearrow FA(macroState)$) garantit que toute trace d'exécution de la machine *MacroLevel* se termine par une séquence infinie d'états satisfaisant le prédicat $FA(macroState)$. La preuve de cette prémisse se fait en garantissant les trois conditions suivantes :

- définir le variant *convProg* dans la machine *MacroLevel*. La variable *convProg* modélise la progression du système vers un état fonctionnellement adéquat et est obtenue en fonction de l'état macroscopique du système. Ici, nous supposons que la variable *convProg* est un entier naturel, mais elle peut être un ensemble fini.

```
variables macroState
invariants
@inv3 convProg ∈ STATES → ℕ
variant convProg (macroState)
```

Figure 4.13 — Définition du variant *convProg* de la machine *MacroLevel*.

▷ prouver que l'événement *selfOrgConv* (figure 4.14) réduit à chaque exécution le variant *convProg*.

```
convergent event selfOrgConv
   any agent
   where
      @grd1 agent∈Agents
      @grd2 actionAgent(agent) ∈ Actions
      @grd3 FA(macroState)=FALSE
      @grd4 convProg (macroState) ≠ 0
   then
      @act1 agentStep (agent):=perceive
      @act2 agentMode(agent):=wait
      @act2 FA, convProg:|FA'(macroState)∈BOOL
          ∧convProg '(macroState)<convProg (macroState)
end
```

Figure 4.14 — Description détaillée de l'événement *selfOrgConv* de la machine *MacroLevel*

▷ prouver que les événements *m* et *observer* n'incrémentent pas le variant *convProg* à chaque exécution. Ces deux événements doivent être marqués ***anticipés*** (***anticipated***) comme le montre la figure 4.15.

```
anticipated event m
   any agent
   where
      @grd1 agent∈Agents
      @grd2 actionAgent(agent) ∈ Actions
      @grd3 FA(macroState)=FALSE
      @grd4 convProg (macroState) ≠ 0
   then
      @act1 agentStep (agent):=perceive
      @act2 agentMode(agent):=wait
      @act2 FA, convProg:|FA'(macroState)∈BOOL ∧
       convProg' (macroState)≤ convProg (macroState)
end

anticipated event observer
   where
      @grd1 FA(macroState)
end
```

Figure 4.15 — Description détaillée des événements *m* et *observer* de la machine *MacroLevel*.

La deuxième prémisse ($MacroLevel \vdash\circlearrowleft \neg FA(macroState)$) assure que la machine *MacroLevel* ne se bloque pas dans un état ne vérifiant pas le prédicat $FA(macroState)$. Prouver cette prémisse revient à ajouter et prouver le théorème suivant à la machine *MacroLevel* :

Preuve de la résilience du SMA Le SMA est dit résilient lorsqu'il est capable de retrouver son état fonctionnellement adéquat suite à des perturbations. Formellement, cette propriété

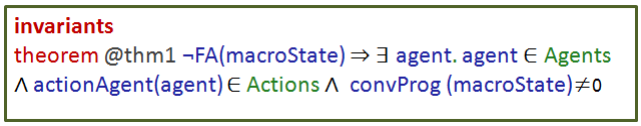

Figure 4.16 — Le théorème de non blocage de la machine *MacroLevel* dans l'état $\neg FA(macroState)$

est exprimée à l'aide de la logique de prédicats et les opérateurs temporels dans la formule suivante :

$$MAS \vdash \Box(\neg FA(macroState) \Rightarrow \Diamond FA(macroState)). \tag{4.1}$$

La formule 4.1 peut s'écrire au moyen de l'opérateur *leads to* (noté $\rightsquigarrow$) comme suit :

$$MAS \vdash \neg FA(macroState) \rightsquigarrow FA(macroState). \tag{4.2}$$

Pour prouver cette formule, deux alternatives sont possibles : utiliser la règle $LIVE_{progress}$ exprimée à l'aide de l'opérateur *Until* de la logique *LTL* ou bien utiliser les règles $WF1$ et $SF1$ basées respectivement sur l'hypothèse d'équité faible et forte de la logique *TLA*.
Dans le cadre de cette thèse, nous avons privilégié l'utilisation des règles de preuve de la logique *TLA* car elles offrent un moyen plus explicite pour exprimer l'ordonnancement de l'exécution des événements et par conséquent pour prouver la formule 4.2.
En considérant les éléments suivants :

- $N = selfOrg \vee selfOrgConv \vee p \vee observer \vee m$,
- $f = convProg(macroState)$,

la règle de preuve $WF1$ est réécrite comme suit :

$$\frac{\begin{array}{ll} WF1.1 & \neg FA(macroState) \wedge [N]_f \Rightarrow (\neg FA'(macroState) \vee FA'(macroState)) \\ WF1.2 & \neg FA(macroState) \wedge \langle N \wedge selfOrgConv\rangle_f \Rightarrow FA'(macroState) \\ WF1.3 & \neg FA(macroState) \Rightarrow Enabled\langle selfOrgConv\rangle_f \end{array}}{\Box[N]_f \wedge WF_f(selfOrgConv) \Rightarrow \neg FA(macroState) \rightsquigarrow FA(macroState)}\ WF1$$

Prouver la formule 4.2 à l'aide de la règle $WF1$ nécessite la condition $WF_f(selfOrgConv)$ formulée comme suit :

$$WF_f(selfOrgConv) \widehat{=} \Diamond\Box Enabled\langle selfOrgConv\rangle_f \Rightarrow \Box\Diamond\langle selfOrgConv\rangle_f \tag{4.3}$$

L'équation 4.3 représente une hypothèse d'équité faible sur l'action de l'événement $selfOrgConv$ et suppose qu'il est continuellement activable ($\Diamond\Box Enabled\langle selfOrgConv\rangle_f$). En l'absence de perturbations, cette condition est vraie. Mais lorsque le système est soumis à des changements, cette supposition n'est plus valide. Par conséquent, la preuve de la résilience nécessite l'application de la règle $SF1$ basée sur une hypothèse d'équité forte sur l'action de l'événement $selfOrgConv$.
La règle de preuve associée est la suivante :

$$\frac{\begin{array}{ll} SF1.1 & \neg FA(macroState) \wedge [N]_f \Rightarrow (\neg FA'(macroState) \vee FA'(macroState)) \\ SF1.2 & \neg FA(macroState) \wedge \langle N \wedge selfOrgConv \rangle_f \Rightarrow FA'(macroState) \\ SF1.3 & \Box\neg FA(macroState) \wedge \Box[N]_f \Rightarrow \Diamond Enabled\langle selfOrgConv \rangle_f \end{array}}{\Box[N]_f \wedge SF_f(selfOrgConv) \Rightarrow P \rightsquigarrow FA'(macroState)} \; SF1$$

La condition $SF_f(selfOrgConv)$ est une hypothèse d'équité forte sur l'action de l'événement $selfOrgConv$. Elle est exprimée à l'aide des opérateurs temporels comme suit :

$$SF_f(selfOrgConv) \widehat{=} \Box\Diamond Enabled\langle selfOrgConv \rangle_f \Rightarrow \Box\Diamond\langle selfOrgConv \rangle_f \tag{4.4}$$

Grâce à la formule 4.4, on démontre que l'action de l'événement $selfOrgConv$ va toujours finir par s'activer lorsque cet événement est infiniment souvent activable (il n'est pas continuellement activable).

4.3 Conclusion

Nous avons présenté dans ce chapitre un cadre général pour la formalisation des SMA auto-organisateurs selon trois niveaux d'abstraction : le niveau micro, méso et macro. Cette formalisation est effectuée en un premier temps à l'aide des automates à états/transitions puis traduite au langage *B-événementiel* afin d'automatiser la preuve des propriétés pertinentes de ces systèmes à savoir :

- la ***correction*** des niveaux micro et méso signifiant que tout agent ne se bloque pas dans les étapes de son cycle de vie. La correction est exprimée à l'aide de la propriété de non blocage de *B-événementiel*. Cette propriété peut être prouvée de manière automatique ou interactive avec le prouveur intégré à la plateforme *Rodin*.
- la **convergence** du SMA au niveau macro exprimée comme une propriété de vivacité à l'aide d'opérateurs de la logique temporelle et prouvée à l'aide des concepts de non blocage et de terminaison d'événements du langage *B-événementiel*. La preuve de la terminaison d'un événement nécessite de montrer qu'il fait décroitre à chaque exécution un entier naturel ou un ensemble fini appelé variant. Dans certains cas, le non déterminisme induit pas les actions des agents et leurs interactions complique la preuve de cette décrémentation. L'utilisation de la règle *LATTICE* et de la règle de preuve $SF1$ de la logique TLA offre une autre alternative pour prouver cette terminaison comme proposé dans [Méry et Poppleton, 2013].
- **la résilience** du SMA au niveau macro exprimée également à l'aide d'opérateurs temporels. Cette propriété a été prouvée à l'aide de la règle de preuve $SF1$ de la logique TLA pour prendre en compte le non déterminisme introduit par les perturbations. La règle $SF1$ est basée sur une hypothèse d'équité forte exprimée à l'aide des opérateurs temporels *always* et *eventually* de la logique temporelle et par conséquent elle ne peut pas être prouvée avec la plateforme *Rodin*. Dans [Méry et Poppleton, 2013], les auteurs proposent de suivre un processus récursif de preuve pourvu que ce processus se termine par une action qui nécessite une hypothèse d'activation avec une équité faible.

La formalisation des SMA auto-organisateurs étant décrite, il est essentiel de l'intégrer dans un processus de développement enrichi avec des guides méthodologiques pour aider le concepteur à développer les agents du système de manière à avoir les propriétés requises. Étant donné cet objectif, nous trouvons tout l'intérêt du langage *B-événementiel* et de la technique de raffinement sur laquelle le développement formel est basé.
Ainsi, nous présentons dans le chapitre suivant un processus de développement basé sur des patrons de raffinement et de preuve pour concevoir des SMA auto-organisateurs. Ce processus, garantit les propriétés de correction, de robustesse et de résilience attendues par un observateur externe au système.

5 RefProSOMAS : un processus formel pour le développement de SMA auto-organisateurs

« Les hommes construisent trop de murs et pas assez de ponts. »
Isaac Newton

CONCEVOIR un système multi-agent dans lequel les agents réussissent à effectuer la tâche qui leur est demandée s'avère particulièrement difficile en tenant compte des caractéristiques des systèmes informatiques complexes mentionnées dans le chapitre 1. La seule façon de faire face à cette difficulté est de permettre aux agents d'interagir de manière appropriée, et le problème revient alors à répondre à la question quand et comment les agents doivent t-ils interagir pour atteindre leurs objectifs ?

Basés sur la distinction entre le niveau micro (celui de l'agent) et le niveau macro (celui du système), nous pouvons identifier deux manières possibles pour répondre à cette question :

- adopter une approche *ascendante* (bottom up) : concevoir les règles comportementales adéquates des agents qui leurs permettent, en interagissant, de satisfaire l'objectif global du système, ou
- adopter une approche *descendante* (top down) : décomposer le problème à résoudre en sous problèmes de manière itérative jusqu'à obtenir une description des entités du systèmes.

Dans la première section de ce chapitre, nous exposons l'expérimentation que nous avons réalisée en adoptant l'approche descendante. Cette exploration nous a permis d'identifier les limites de cette approche dans le cas des SMA auto-organisateurs mais également de comprendre davantage le principe de raffinement associé avec le langage B-événementiel et la manière avec laquelle la preuve est effectuée. Cette section est conclue en mentionnant nos motivations pour choisir l'approche ascendante.

La deuxième section décrit RefProSOMAS : un processus ascendant pour la modélisation et la vérification de SMA auto-organisateur basé sur la formalisation proposée dans le chapitre 4.

5.1 Et si on adopte une approche de raffinement descendante ?

Généralement, l'adoption d'une approche descendante pour le développement de SMA implique la décomposition du problème en sous problèmes, l'identification des agents en fonction de ces sous problèmes et l'attribution de chacun des sous problèmes aux agents identifiés. Cette approche a été utilisée dans [Kacem *et al.*, 2007] pour le développement des SMA à l'aide du langage *TemporalZ* (une extension du langage *Z* avec les opérateurs temporels) et également dans [Pereverzeva *et al.*, 2012b] et [Pereverzeva *et al.*, 2012a] pour la modélisation des SMA critiques et les SMA tolérants aux fautes. Nous jugeons que cette approche n'est pas adaptée pour les SMA auto-organisateurs pour deux raisons. La première est que les objectifs d'un agent changent en fonction de son évolution au cours du temps et ne peuvent pas par conséquent lui être imposés dès le départ. La deuxième raison est que cette manière de décomposition exige généralement la présence d'une entité centrale qui veille à ce que l'objectif global soit achevé (tel est le cas dans [Pereverzeva *et al.*, 2012a]) ce qui se contredit avec la nature décentralisée des SMA auto-organisateurs. L'adaptation de l'approche descendante est alors nécessaire pour pouvoir l'appliquer aux SMA auto-organisateurs.

Nous avons proposé cette adaptation dans [Laibinis *et al.*, 2014], en modélisant une colonie de fourmis dont l'objectif global a été de récolter toute la nourriture se trouvant dans l'environnent des fourmis. L'objectif est de prouver que les fourmis seront capables de ramener toute la nourriture au nid. Cette application nous a permis d'identifier les étapes de raffinement suivantes.

- Décrire le modèle initial qui établit un lien explicite entre l'objectif global du système et le comportement des agents. Ce modèle initial est formé d'un premier événement (appelé *Observer*) dont la garde décrit l'objectif commun du système et d'un deuxième événement (appelé *Change*) qui modélise un changement éventuel dans l'environnement et ayant une influence sur l'achèvement de l'objectif global. Dans les raffinements qui suivent, le but est de prouver que l'événement *Observer* finira par être activé ce qui signifie l'achèvement de l'objectif global.
- Réaliser un premier raffinement pour établir le lien entre les changements qui peuvent avoir lieu dans l'environnement et les actions des fourmis. Ce premier raffinement se fait en introduisant les agents, raffinant l'événement *Change* par l'événement *Perform* pour modéliser une action d'un agent et en introduisant les événements *Decide* et *Perceive* qui correspondent respectivement aux décisions et aux opérations de mise à jour des représentations partielles. A ce niveau de raffinement, il faut prouver la terminaison des événements *Decide* et *Perceive*.
- Effectuer un deuxième raffinement pour établir le lien entre les décisions et les actions

des fourmis. Ainsi, l'événement *Decide* est décomposé en les différentes règles de décision des fourmis et l'événement $Perform$ est décomposé pour modéliser les actions correspondantes à ces décisions.

- ▷ Introduire les opérations de mise à jour des perceptions partielles au troisième raffinement. Cette étape permettra de faire le lien entre les règles de décisions et les perceptions.
- ▷ Prouver la convergence de tous les événements modélisant les actions des fourmis (ceux qui raffinent l'événement $Perform$).

L'application de l'approche descendante nous a permis de comprendre comment employer la technique de raffinement conjointement avec les concepts d'événements convergents et anticipés pour prouver que le système va converger finalement vers son objectif global. Cependant, elle présente quelques limites par rapport aux caractéristiques des SMA auto-organisateurs. Ces limites sont reliées au fait d'être obligé de définir l'objectif global du système au début de la spécification. En effet, pour un SMA auto-organisateur, la convergence peut se manifester de plusieurs manières. Dans le cas des fourmis fourrageuses, la convergence du système se manifeste lorsque les fourmis réussissent à ramener toute la nourriture au nid et lorsqu'elles montrent une exploration efficace de leur environnement.

Finalement, nous sommes convaincus que l'approche ascendante est mieux adaptée pour les SMA auto-organisateurs et permet de développer plus naturellement ces systèmes. Dans la section suivante, nous décrivons RefProSOMAS, un processus ascendant pour la modélisation et la vérification des SMA auto-organisateurs.

5.2 RefProSOAMS : un processus pour le développement formel des SMA auto-organisateurs

Le processus que nous proposons est formé de trois phases guidées par des patrons de raffinement comme le montre la figure 5.1. Ces patrons décrivent les liens entre les machines *MicroLevel*, *MesoLevel* et *MacroLevel* décrites dans le chapitre précédent. La première étape (figure 5.1 ❶), basée sur le patron P_MAS, focalise sur la modélisation du comportement nominal des agents. L'application de ce patron garantit la correction (absence de blocage) des spécifications obtenues. L'étape suivante (figure 5.1 ❷), a pour objectif de prouver que le comportement conçu permet au système d'atteindre son objectif global et d'identifier les conditions nécessaires pour cela. Le processus de preuve est guidé par le patron P_CONV et peut engendrer une amélioration du comportement des agents obtenu dans la première étape. La dernière étape (figure 5.1 ❸) se base sur le patron P_RES et a pour but de prouver que le système est capable de s'auto-adapter face aux perturbations qui peuvent avoir lieu dans l'environnement. Cette étape permettra d'identifier les conditions nécessaires pour que le système soit résilient et peut également apporter des améliorations au niveau des règles comportementales des agents. Les sections suivantes décrivent chacune de ces étapes ainsi que les patrons qui leur sont associés.

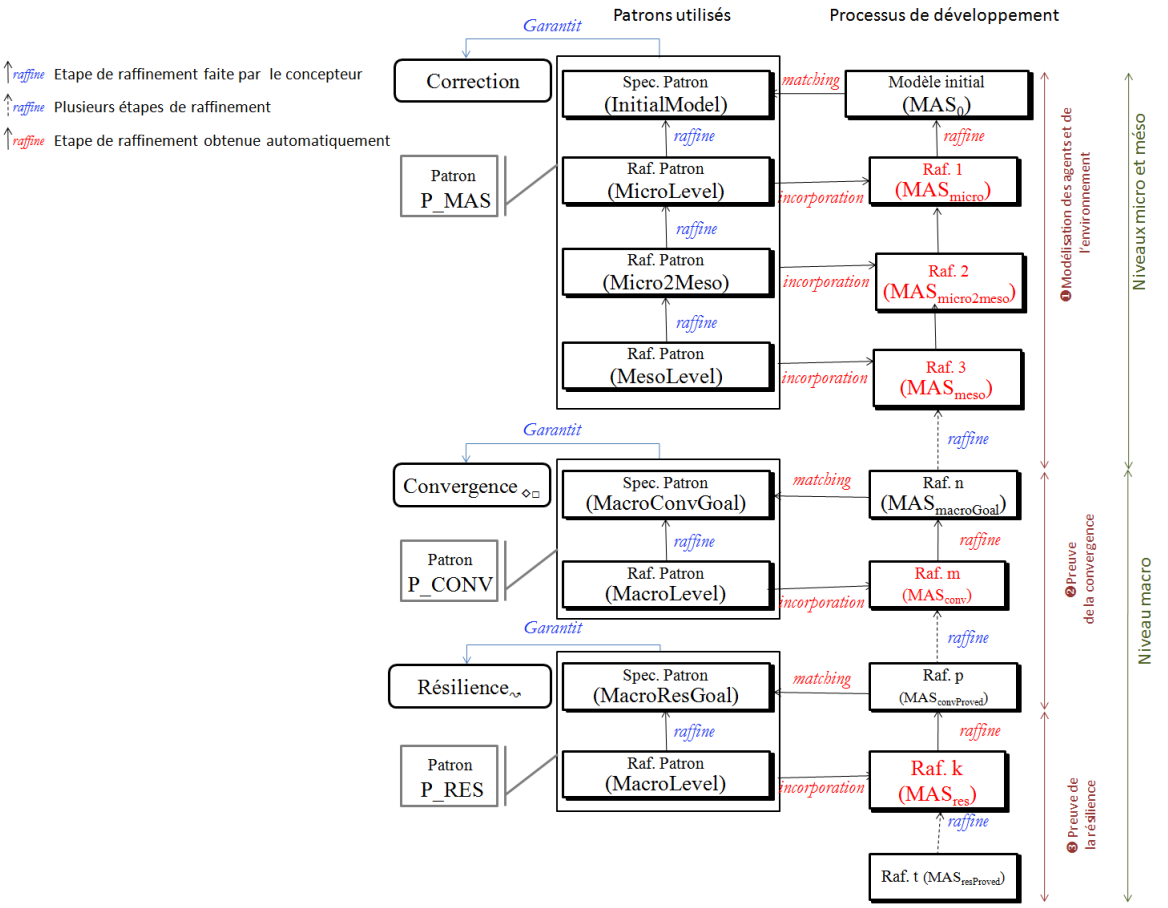

Figure 5.1 — Approche de formalisation des SMA auto-organisateurs basée sur les patrons.

5.2.1 Phase 1 : modélisation du comportement nominal des agents avec le patron *P_MAS*

Dans cette phase, nous définissons le comportement nominal des agents constituant le SMA. Afin d'assister le concepteur tout au long de cette phase, nous proposons le patron de raffinement *P_MAS* (figure 5.2). Ce patron décrit une stratégie de raffinement pour la conception des agents en garantissant le non blocage dans leur comportement. Il est formé d'un modèle initial et de trois étapes de raffinement qui génèrent la machine *MicroLevel* puis la machine *MesoLevel* décrites dans les sections 4.2.1 et 4.2.2 du chapitre précédent.

La machine la plus abstraite du patron *P_MAS* appelée *InitialModel* et donnée dans la figure 5.3, décrit le système comme un ensemble d'agents fonctionnant selon le cycle *percevoir-décider-agir* situés dans un environnement dont le comportement est modélisé par l'événement *EnvironmentChange*. Le contexte *ContextInit*, qui lui est associé, décrit les modes *wait* et *work* (ensemble *Modes*) et les trois étapes (ensemble *Steps*) dans lesquels chaque agent peut se retrouver.

Chacun des événements *Perceive*, *Decide* et *Perform* (figure 5.4) correspond à une étape du cycle de vie d'un agent et permet de faire passer l'agent considéré vers l'étape suivante. Ce modèle abstrait garantit la propriété *LocProp*1 (Tout agent fonctionne selon le cycle

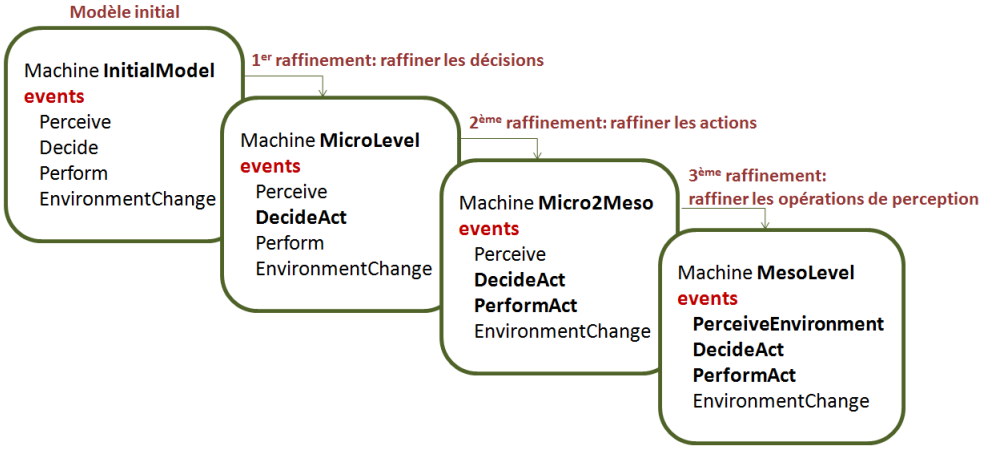

Figure 5.2 — Le patron de raffinement *P_MAS*

```
machine InitialModel
sees ContextInit
variables agentStep  agentMode
Invariants
@inv1 agentStep ∈ Agents → Steps
@inv2 agentMode ∈ Agents → Modes
events
 event INITIALISATION
  then
   @act1 agentMode, agentStep :| agentMode'=(Agents×{work}) ∧ agentStep'=Agents×{perceive}
 end
 event Perceive
 event Decide
 event Perform
 event EnvironmentChange
  where
   @grd1 ∀agent agent∈Agents⇒agentMode(agent)=wait
  then
   @act1 agentMode :| agentMode'=(Agents×{work})
end
end
```

Figure 5.3 — La machine initiale du patron *P_MAS*

percevoir-décider-agir) et il sera soumis à une série de raffinements composés de trois étapes.

Le premier raffinement consiste à détailler les règles décisionnelles des agents. Ainsi, le raffinement de la machine *InitialModel* se fait en divisant l'événement *Decide* en plusieurs événements dont chacun représente une règle décisionnelle. La règle décisionnelle d'un agent doit lui permettre de choisir sa prochaine action à exécuter, d'activer les actionneurs correspondant à cette action et de modifier les propriétés de décisions relatives à sa décision. Cette forme générique d'une décision est modélisée par l'événement *DecideAct* résultat du raffinement de l'événement *Decide*. Cet événement est décrit dans la figure 4.6 de la section 4.2.1.

Ainsi, lors du raffinement de la machine *InitialModel* par la machine *MicroLevel*, nous introduisons également les variables qui représentent les propriétés relatives aux décisions, aux actionneurs et aux prochaines actions à exécuter. Ce raffinement permet l'obtention de la machine *MicroLevel* modélisant le niveau micro du système et décrite par la figure 4.5 de la section 4.2.1.

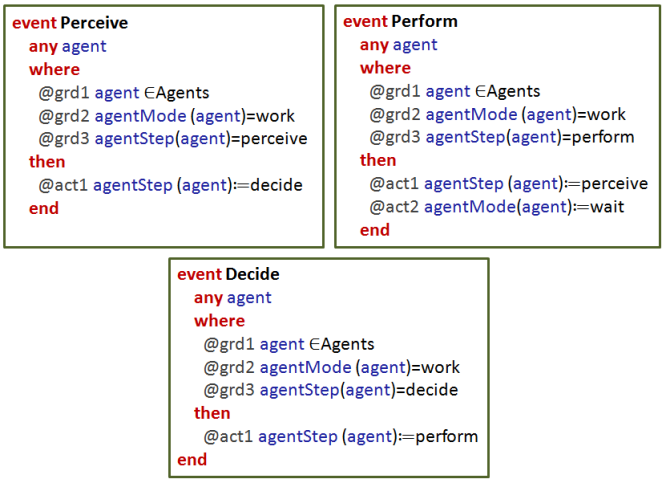

Figure 5.4 — Les événements *Perceive*, *Decide* et $Perform$ de la machine initiale du patron P_MAS

L'invariant de collage $GluingActuators$ est introduit dans la machine $MicroLevel$ pour formaliser le lien entre l'étape du cycle de vie d'un agent d'une part et l'état de ses actionneurs et l'action qu'il doit exécuter d'autre part. Cet invariant est décrit comme suit :

$$
\begin{aligned}
&GluingActuators \mathrel{\widehat{=}} \\
&\forall a.a \in Agents \wedge agentStep(a) = perform \Leftrightarrow \\
&agentActuators(a, agentAction(a)) = enabled
\end{aligned}
$$

L'incorporation de cette étape de raffinement dans le processus de développement génère les événements correspondant aux règles décisionnelles des agents conformément à l'événement $DecideAct$ dans la machine MAS_{micro} (figure 5.1).

Lors de la deuxième étape de raffinement du patron P_MAS, l'événement $Perform$ est raffiné en les différentes actions qu'un agent peut effectuer. Chacune de ces actions est modélisées conformément à l'événement $PerformAct$ décrit lors du chapitre précédent (figure 4.11). Ainsi, la définition d'une action revient à définir les conditions de son activation (les actionneurs correspondant à l'action choisie doivent être activés) et les modifications qu'elle va engendrer. Ces modifications peuvent concerner certaines propriétés internes de l'agent ou bien certains éléments de l'environnement. Au niveau de ce raffinement, on ajoute donc les éléments de l'environnement (sous forme de variables) ainsi que les propriétés des agents qui seront modifiées par les actions.

La propriété $LocProp2$ vérifie que tout agent a ayant pris une décision, décrite par le prédicat avant-après $DecideAct\_i(a, A_{prop}, A'_{prop})$, ne sera pas bloqué dans l'étape de décision et pourra exécuter l'action choisie. Cette propriété est prouvée à ce niveau de raffinement et est formalisée comme suit :

$$
\begin{array}{l}
LocProp2 \mathrel{\widehat{=}} \\
\forall a \cdot a \in Agents \wedge DecideAct\_i(a, A_{prop}, A'_{prop}) \Rightarrow \\
\exists PerformAct\_i \cdot G\_PerformAct\_i(a, pPerformAct\_i)
\end{array}
$$

Dans cette propriété, $G\_PerformAct\_i(a, pPerformAct\_i)$ désigne la garde d'un événement d'action définie en fonction de l'agent a et d'un ensemble de paramètres $pPerformAct\_i$. Informellement, cette propriété indique que pour tout agent a ayant pris une décision (passé à l'étape d'action), il existe un événement d'action $PerformAct\_i$ activable pour l'agent a. L'incorporation de cette étape de raffinement donne lieu à la machine $MAS_{micro2meso}$ qui décrit les décisions et les actions des agents et qui représente une machine intermédiaire entre la description du niveau micro et celle du niveau méso.

Lors du raffinement de la machine *Micro*2*Meso* (le troisième raffinement dans la figure 5.2), les représentations partielles des agents ainsi que leurs capteurs sont introduits et l'événement *Perceive* est raffiné par l'événement *PerceiveEnvironment* (figure 4.10 du chapitre précédent) en spécifiant les différentes opérations de mise à jour de ses représentations. Les capteurs de tout agent se trouvant dans la phase de perception, doivent être activés. Cette contrainte est assurée via l'invariant de collage *GluingSensors*.

$$
\begin{array}{l}
GluingSensors \mathrel{\widehat{=}} \\
\forall a.a \in Agents \wedge agentStep(a) = perceive \Leftrightarrow \\
agentSensors(a) = enabled
\end{array}
$$

Pour garantir que tout agent ne sera pas bloqué dans l'étape de perception, la propriété *LocProp*3 (énoncée si-dessous) doit être prouvée.

$$
\begin{array}{l}
LocProp3 \mathrel{\widehat{=}} \\
\forall a \cdot a \in Agents \wedge PerceiveEnvironment\_i(a, A_{rep}, A'_{rep}) \Rightarrow \\
\exists DecideAct\_i \cdot G\_DecideAct\_i(a, pDecideAct\_i)
\end{array}
$$

Dans cette propriété, $G\_DecideAct\_i(a, pDecideAct\_i)$ désigne la garde d'un événement de décision définie en fonction de l'agent a et d'un ensemble de paramètres $pDecideAct\_i$.

L'incorporation de ce raffinement dans le processus de développement donne lieu à la machine MAS_{meso} qui modélise un comportement correct des agents.

La première phase du processus RefProSOMAS a permis la formalisation des règles décisionnelles des agents ainsi que les actions et les opérations de perceptions leurs permettant d'interagir (les niveaux micro et méso). Les deux prochaines phases ont pour objet de raisonner sur la convergence et la résilience du système au niveau macro.

5.2.2 Phase 2 : preuve de la convergence du SMA avec le patron P_CONV

Cette deuxième phase du processus a pour objectif la preuve de la convergence du SMA conçu. La machine MAS_{meso} (résultat de la première phase) est raffinée par la machine $MAS_{macroGoal}$ en introduisant l'objectif global des agents. Ce raffinement se fait manuellement et permet de préparer l'application du patron P_CONV qui sert de guide afin de prouver la convergence des agents vers l'objectif global du système.

Le patron P_CONV est composé par :

- la machine *MacroConvGoal* qui décrit de façon généralisée le comportement d'un ensemble d'agent ainsi que l'objectif global à atteindre. Cette machine est composée des événements de la machine *MesoLevel* du patron P_MAS et de l'événement *observer* qui modélise l'objectif global du système.
- la machine *MacroLevel* décrite à la section 4.2.3 par la figure 4.12. Rappelons que la machine *MacroLevel* est formée d'un ensemble d'événements qui permettent d'atteindre la fonction globale du système (noté $selfOrgConv$), ceux qui la maintiennent (notés m), les perturbations (notés p), les événements d'auto-organisation notés $selfOrg$ et de l'événement *observer*. A ce niveau de raffinement, nous nous intéressons à prouver la convergence du système. Ainsi, seuls les événements m, $selfOrgConv$ et *observer* sont considérés.

L'intégration de ce patron dans le processus de conception permet d'automatiser le raffinement de la machine $MAS_{macroGoal}l$ en MAS_{conv}. Cette intégration est effectuée en deux étapes. La première consiste à raffiner tous les événements d'action par l'événement $selfOrgConv$ du patron P_CONV et décrit par la figure 4.14 dans la section 4.2.3. La deuxième étape a pour objet la preuve de la terminaison des événements d'action et le non blocage de la machine MAS_{conv} dans un état non désiré. Pour prouver la terminaison des événements d'action, qui passe par la définition d'un variant, plusieurs étapes de raffinement sont parfois nécessaires. En effet, à chaque étape, un ensemble d'événements décrémentant le même variant sont considérés par la preuve. Les événements non considérés doivent être définis comme des événements *anticipés* pour signaler que la preuve de leur terminaison est reportée pour les prochaines étapes de raffinement. Comme indiqué dans le chapitre 4 (section 4.1.3), le choix du variant n'est pas toujours évident et dans certains cas, il faut avoir recours aux hypothèses d'équité forte pour prouver la terminaison.

5.2.3 Phase 3 : preuve de la résilience du SMA avec le patron P_RES

Le comportement décrit par la machine $MAS_{convProved}$ permet aux agents d'atteindre l'adéquation fonctionnelle en l'absence de perturbations. Lors de cette troisième phase, la machine $MAS_{convProved}$ est raffinée conformément au patron P_RES afin de spécifier puis prouver le comportement adéquat pour faire face aux perturbations et aux changements de l'environnement.

La spécification du patron *P_RES* est composée des deux machines suivantes :

- *MacroResGoal* : modélise le comportement d'un ensemble d'agents et l'objectif de la résilience.
- *MacroLevel* : raffine *MacroResGoal* en tenant compte des perturbations et des mécanismes d'auto-organisation.

L'intégration du patron *P_RES* dans le processus de développement se fait en raffinant :

- l'événement *EnvironmentChange* dans la machine MAS_{res} pour modéliser les perturbations de l'environnement et
- les événements d'action pour modéliser les mécanismes d'auto-organisation permettant aux agents de neutraliser les effets provoqués par les perturbations et revenir aux règles comportementales nominales pour converger à nouveau vers l'objectif du SMA.

La preuve de la résilience se fait en prouvant la formule $\neg FA(macroState) \rightsquigarrow FA(macroState)$ conformément à la discussion menée au chapitre 4. Elle peut nécessiter plusieurs itérations de raffinement et donne lieu à la machine $MAS_{resProved}$. Cette machine marque la fin du processus de développement et elle comporte les règles comportementales nominales des agents et celles relatives à leur capacité d'auto-organisation.

5.3 Conclusion

Dans ce chapitre, le processus RefProSOMAS de modélisation et de vérification formelle de SMA auto-organisateurs est décrit. Ce processus, formé de trois phases, se base sur des patrons de raffinement visant à guider le développement et la vérification dans chacune de ses étapes. Pour illustrer ce processus, nous présentons dans le chapitre suivant une modélisation de l'étude de cas des fourmis fourrageuses à l'aide de RefProSOMAS.

Troisième partie

Application

6 Application de RefProSOMAS sur le cas d'étude des fourmis fourrageuses

Dans ce chapitre, nous appliquons le processus ascendant RefProSOMAS sur l'étude de cas des fourmis fourrageuses. A travers cette application, nous visons aussi à illustrer les modèles génériques décrits sous forme de patrons au chapitre précédent.
La description de notre cas d'étude fait l'objet de la première section. Les trois sections suivantes décrivent l'application de chacune des phases RefProSOMAS pour modéliser le comportement local des fourmis et ensuite prouver les propriétés globales de convergence et de résilience.

6.1 Description du cas d'étude des fourmis fourrageuses

L'exemple que nous présentons est une formalisation des comportements d'une colonie de fourmis fourrageuses [Topin *et al.*, 1999]. Le système est composé de plusieurs fourmis qui se déplacent à la recherche de nourriture dans un environnement qui est un ensemble de cellules reliées entre elles. L'objectif principal des fourmis est de ramener toute la nourriture placée dans l'environnement vers leur nid. Elles n'ont pas d'information sur les emplacements des sources de nourriture, mais elles sont capables de percevoir la nourriture qui est à l'intérieur de leur champ de perception. Les fourmis interagissent entre elles via l'environnement en déposant des phéromones. Les perturbations provenant de l'environnement sont principalement l'évaporation de la phéromone, l'apparition d'obstacles et la disparition ou l'apparition de nourriture.

Le comportement des fourmis est décrit comme suit. Initialement, toutes les fourmis sont dans le nid. Lors de l'exploration de l'environnement, la fourmi met à jour ses représentations dans son champ de perception et décide vers quelle cellule se déplacer. Lors d'un déplacement, la fourmi doit éviter les obstacles. Selon ce qu'elle perçoit, trois cas sont possibles :

1. la fourmi perçoit la nourriture : elle décide de prendre la direction dans laquelle le gradient de la nourriture est le plus fort,

2. la fourmi sent uniquement de la phéromone : elle décide de s'orienter vers la direction dans laquelle le gradient de phéromone est le plus fort et
3. la fourmi ne perçoit rien : elle choisit sa prochaine destination au hasard.

Quand une fourmi atteint une source de nourriture, elle en prend une partie et rentre au nid. Si de la nourriture reste à cet endroit, la fourmi dépose de la phéromone sur son chemin de retour. En arrivant au nid, la fourmi dépose la nourriture récoltée et recommence une autre exploration.

En plus des propriétés de non blocage *LocProp*1, *LocProp*2 et *LocProp*3 (décrites à la section 3), les propriétés suivantes doivent être vérifiées au niveau micro.

- ▷ *LocInv*1 : la fourmi doit éviter les obstacles et
- ▷ *LocInv*2 : un endroit donné ne peut pas contenir à la fois des obstacles et de la nourriture.

Les propriétés globales auxquelles nous nous intéressons sont décrites comme suit :

- ▷ la convergence : les fourmis sont en mesure d'apporter toute la nourriture au nid
- ▷ la résilience : lorsqu'on rajoute une nouvelle source de nourriture dans l'environnement, les fourmis sont capables de la détecter.

6.2 Phase N°1 : modélisation du comportement des fourmis à l'aide du patron P_MAS

6.2.1 Modèle initial

Le modèle initial décrit un ensemble de fourmis dont chacune fonctionne selon le cycle *percevoir-décider-agir*. Ce modèle est donné par la machine *AntsInitModel* donnée par la figure 6.1.

Dans le contexte *ContextAnts*_0, nous définissons les ensembles *Steps* et *Modes* correspondant respectivement aux étapes du cycle de vie et les modes par lesquels une fourmi peut passer. Ces ensembles sont définies par les axiomes *axm*1 et *axm*2 du contexte *ContextAnts*_0 décrit par la figure 6.2.

6.2.2 Modélisation des règles décisionnelles des fourmis

La machine *AntsInitModel* est raffinée conformément au patron *P_MAS* afin de modéliser les règles décisionnelles des fourmis. La réalisation de ce raffinement sous la plateforme *Rodin* au moyen du plugin *Pattern* nécessite deux étapes. La première consiste à appliquer le patron *P_MAS* quatre fois pour modéliser les quatre événements suivants :

machine AntsInitModel
SEES
ContextAnts_0
VARIABLES
antStep
antMode
INVARIANTS
$defAntStep : antStep \in Ants \rightarrow Steps$
$defAntMode : antMode \in Ants \rightarrow Modes$
EVENTS
INITIALISATION
THEN
$initStep : antMode, antStep : |antMode' = Ants \times \{work\} \wedge antStep' = Ants \times \{perceive\}$
END
EVENT PerceiveAnts CONVERGENT
ANY
ant
WHERE
$grd1 : ant \in Ants$
$grd2 : antMode(ant) = work$
$grd3 : antStep(ant) = perceive$
THEN
$updStepAg : stepAgent(ant) := decide$
END
EVENT DecideAnts CONVERGENT
ANY
ant
WHERE
$grd1 : ant \in Ants$
$grd2 : antMode(ant) = work$
$grd3 : antStep(ant) = decide$
THEN
$updStepAg : stepAgent(ant) := perform$
END
EVENT PerformAnts
ANY
ant
WHERE
$grd1 : ant \in Ants$
$grd2 : antMode(ant) = work$
$grd3 : antStep(ant) = perform$
THEN
$act1 : stepAgent(agent) := perceive$
$act2 : antMode(ant) := wait$
END
END

Figure 6.1 — Le modèle initial : la machine *AntsInitModel*

- ▷ *DecideAntsDropFood* : correspond à une décision de dépôt de nourriture dans le nid,
- ▷ *DecideAntsDropPheromone* : correspond à une décision de dépôt de phéromone sur le chemin de retour,
- ▷ *DecideAntsHarvestFood* : modélise une décision de récolte de nourriture,
- ▷ *DecideAntsMove* : modélise la décision de faire un déplacement.

L'application du patron *P_MAS* donne lieu à quatre machines dont chacune raffine la machine initiale *AntsInitModel* et décrit une règle de décision parmi les quatre déci-

context ContextAnts_0
SETS
Ants
Steps
Modes
CONSTANTS
perceive
decide
perform
wait
work
AXIOMS
axm1 : $finite(Ants)$
axm2 : $partition(Steps, perceive, decide, perform)$
axm3 : $partition(Modes, wait, work)$
END

Figure 6.2 — Le contexte du modèle initial *ContextAnts_0*

sions qu'une fourmi peut prendre. Dans la seconde étape, ces quatre machines sont fusionnées à l'aide du plugin *Feature Composition Plug-in* pour donner lieu à la machine *AntsDecisions0*. Cette étape de fusion est suivie par une étape de raffinement de l'événement *DecideAntsMove* pour différencier le mouvement d'exploration de l'environnement et le mouvement de retour au nid.
La figure 6.3 illustre ces différentes étapes. Les relations d'extensions définies entre les contextes de ces modèles sont décrites par la figure 6.4. Ces contextes seront détaillés dans la section 6.2.2.3.

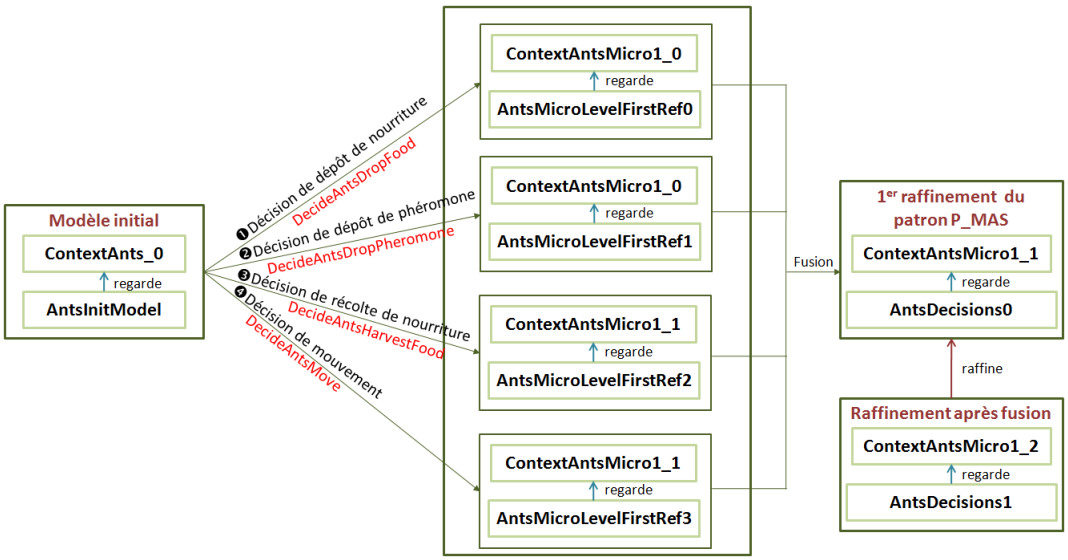

Figure 6.3 — Les différents modèles utilisés pour décrire les règles décisionnelles des fourmis

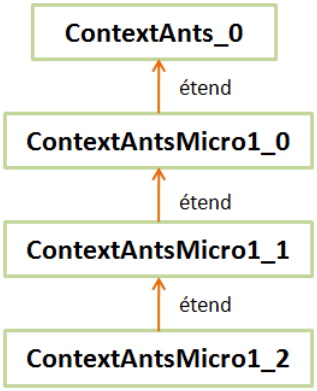

Figure 6.4 — Les contextes utilisés par les modèles décrivant les règles décisionnelles des fourmis

6.2.2.1 Exemple d'illustration : application du premier raffinement du patron *P_MAS* pour raffiner l'événement *DecideAntsMove*

Dans cette section, nous décrivons l'utilisation du patron *P_MAS* pour modéliser l'événement *DecideAntsMove* (la flèche étiquetée ❹ sur la figure 6.3). L'utilisation d'un patron nécessite de faire une correspondance entre les éléments du modèle abstrait du patron avec ceux du modèle abstrait de l'application. Dans le tableau 6.1, nous définissons cette correspondance.

	Éléments du modèle générique *InitialModel* (section 5.2.1)	**Éléments du système des fourmis fourrageuses *AntsInitModel***
Ensembles définis dans le contexte	Agents Steps Modes	Ants Steps Modes
Variables	agentStep agentMode	antStep antMode
Evènements	Perceive Decide Perform	PerceiveAnts DecideAnts PerformAnts

Tableau 6.1 — Correspondance entre les éléments du modèle initial du patron *P_MAS* avec le modèle initial *AntsInitModel*

Une fois cette correspondance effectuée, le raffinement de l'événement *DecideAnts* (de la machine *AntsInitModel*) par l'événement *DecideAntsMove* dans la machine *AntsMicroLevelFirstRef3* est réalisé automatiquement. Comme indiqué dans le patron *P_MAS*, ce raffinement consiste à définir les variables représentant la prochaine action à entreprendre (*nextAgentAction*), l'actionneur à activer (*agentActuators*) et la propriété de l'agent à modifier (*agentDecisionProp*). La définition de ces éléments pour l'événement *DecideAntsMove* est donnée par le tableau 6.2.

	Éléments du modèle générique *MicroLevel* (section 4.2.1	**Éléments du système des fourmis fourrageuses *AntsMicroLevelFirstRef3***
Variables	nextAgentAction agentActuators agentDecisionProp	antNextAction paw nextLocation
Evènements	DecideAct	DecideAntsMove

Tableau 6.2 — Renommage des éléments du modèle concret du patron *P_MAS* lors du raffinement de l'événement *DecideAnts*

L'événement *DecideAntsMove* résultat de l'application du patron *P_MAS* est donné par la figure 6.5. Il est à noter que l'événement obtenu directement suite à l'application du patron a été modifié légèrement pour avoir celui présenté dans la figure. Ces modifications ont concerné essentiellement la transformation de certaines actions non déterministes en des actions déterministes.

EVENT DecideAntsMove
REFINES DecideAnts
ANY
  $agent$
  $nextLoc$
WHERE
  $grd1 : agent \in Ants$
  $grd2 : antStep(agent) = decide$
  $grd3 : antMode(agent) = work$
  $grd4 : nextLoc \in Locations$
WITH
  $ant : ant = agent$
THEN
  $act1 : antStep := antStep \lhd\!\!+ \{agent \mapsto perform\}$
  $act2 : antNextAction := antNextAction \lhd\!\!+ \{agent \mapsto move\}$
  $act3 : nextLocation(agent) := nextLoc$
  $act4 : paw := paw \lhd\!\!+ \{agent \mapsto move \mapsto enable\}$
END

Figure 6.5 — L'événement *DecideAntsMove* de la machine *AntsMicroLevelFirstRef3*

Le tableau 6.3 récapitule des statistiques sur les obligations de preuves générées pour les quatre machines obtenues suite à l'application du patron *P_MAS* pour la modélisation des règles décisionnelles des fourmis. [1]

1. Ces statistiques (et parfois des statistiques dans le reste du document) montrent que certaines obligations de preuve ne sont pas déchargées. Ceci est dû principalement à des affectations non déterministes pour lesquelles il faut montrer la faisabilité. Ces démonstrations sont tout à fait faisables mais elles n'ont pas été déchargées au moment de la rédaction de ce manuscrit.

	Nombre total d'Obligations de Preuve (OP)	Nombre d'OP déchargées automatique-ment	Nombre d'OP déchargées manuellement
AntsMicroLevelFirstRef0	19	18	1
AntsMicroLevelFirstRef1	19	18	1
AntsMicroLevelFirstRef2	26	21	2
AntsMicroLevelFirstRef3	20	18	2

Tableau 6.3 — Statistiques concernant les obligations de preuves des machines générées suite à l'application de patron *P_MAS* pour modéliser les décisions des fourmis

6.2.2.2 La fusion des règles décisionnelles dans une seule machine

Cette phase de fusion a pour but de regrouper tous les événements qui décrivent les règles décisionnelles des fourmis dans une seule machine. La fusion donne lieu à la machine *AntsDecisions*0 (figure 6.3) dont un extrait est donné par la figure 6.6.

```
Machine AntsDecisions0
  REFINES AntsInitModel
  SEES
    ContextAntsMicro1_1
  VARIABLES
    antStep
    antMode
    antNextAction
    nextLocation
    paw
    ...
  INVARIANTS
    ...
  EVENTS
  INITIALISATION
  EVENT PerceiveAnts
  EVENT DecideAntsMove
  EVENT DecideAntsDropFood
  EVENT DecideAntsHarvestFood
  EVENT DecideAntsDropPheromone
  EVENT PerformAnts
  EVENT EnvironmentChange
END
```

Figure 6.6 — Extrait de la machine *AntsDecisions*0

Dans la machine *AntsDecisions*0, les événements *PerceiveAnts*, *PerformAnts* et *EnvironmentChange* sont ceux de la machine *AntsInitModel*. Les événements *DecideAntsMove*, *DecideAntsDropFood*, *DecideAntsHarvestFood* et *DecideAntsDropPheromone* raffinent l'événement *DecideAnts*.
Pour prouver la correction de la fusion, 66 obligations de preuve ont été générées dont 64 ont été déchargées automatiquement et 2 ont été prouvées manuellement à l'aide du prouveur interactif de *Rodin*.

6.2.2.3 Raffinement de la machine *AntsDecisions*0

Cette étape de raffinement est nécessaire pour distinguer les mouvements d'exploration de l'environnement des mouvements conduisant les fourmis au nid. Lors du raffinement de la machine *AntsDecisions*0 par la machine *AntsDecisions*1, l'événement *DecideAntsMove* est divisé en deux événements : *DecideAntsMoveExplore* et *DecideAntsMoveBack* qui modélisent respectivement une décision de faire un déplacement en quête de nourriture et un déplacement pour retourner au nid. Ce raffinement permet de spécifier comment la prochaine position de la fourmi est déterminée lorsqu'elle est en exploration et dans le cas où elle rebrousse chemin vers son nid.

Afin de déterminer sa direction (la prochaine position à visiter), la fourmi se base sur ses aptitudes lui permettant d'évaluer le gradient de phéromone et de nourriture à proximité et d'attribuer des poids pour chacune des positions qui lui sont accessibles en fonction de ces gradients. La définition abstraite de ces aptitudes est donnée dans le contexte *ContextAntsMicro*1_2. La formalisation des déplacements des fourmis nécessite la modélisation de l'environnement. Le contexte *ContextAntsMicro*1_0 contient les axiomes permettant de décrire l'environnement sous forme d'une grille ainsi que les relations de voisinage entre les positions la constituant. Les paragraphes suivants décrivent les principaux éléments des contextes *ContextAntsMicro*1_0, *ContextAntsMicro*1_1 et *ContextAntsMicro*1_2 ainsi que les événements *DecideAntsMoveExplore* et *DecideAntsMoveBack*.

Le contexte *ContextAntsMicro*1_0 : formalisation de l'environnement

La figure 6.7 est un extrait du contexte *ContextAntsMicro*1_0 décrivant les axiomes relatifs à la description de l'environnement. Le contexte *ContextAntsMicro*1_0 définit un en-

$$
\begin{array}{l}
axm1 : Nest \in Locations \\
axm2 : Locations \setminus \{Nest\} \neq \varnothing \\
axm3 : Grid \in Locations \leftrightarrow\!\!\!\!\!\rightarrow Locations \\
axm4 : Grid \neq \varnothing \\
axm5 : Grid = Grid^{-1} \\
axm6 : (Locations \lhd id) \cap Grid = \varnothing \\
axm7 : Next \in Locations \rightarrow \mathbb{P}(Locations) \\
axm8 : \forall loc \cdot loc \in Locations \Rightarrow Next(loc) = Grid[\{loc\}] \\
axm9 : \forall loc1 \cdot loc1 \in Locations \Rightarrow (\exists loc2 \cdot loc2 \in Next(loc1))
\end{array}
$$

Figure 6.7 — Les axiomes définissant l'environnement dans le contexte *ContextAntsMicro*1_0

semble de positions *Locations* dont une position particulière représentant le nid ($axm1$). Cet ensemble doit contenir au moins une position en plus du nid ($axm2$). L'environnement est modélisé dans l'axiome $axm3$ par un graphe (appelé *Grid*) comme une relation sur l'ensemble des positions. Il est définit ainsi par un ensemble de couples de positions non vide ($axm4$). Chaque lien d'une position $loc1$ vers une position $loc2$ signifie que $loc2$ est accessible à partir de $loc1$ (un déplacement de $loc1$ vers $loc2$ est possible). L'axiome $axm5$ rend ce lien symétrique ($loc1$ est aussi accessible à partir de $loc2$). L'axiome $axm6$ permet d'éliminer les boucles dans le graphe (les couples de la forme $(loc1, loc1)$ ne doivent pas appartenir à *Grid*).

L'axiome *axm*7 définit la fonction totale *Next* de l'ensemble *Locations* vers l'ensemble des parties de *Locations*. La fonction *Next* définit pour chaque position l'ensemble des positions qui sont accessibles à partir de cette position (*axm*8). L'axiome *axm*9 indique qu'à partir de n'importe quelle position, il existe au moins une position dans son voisinage.

Le contexte *ContextAntsMicro*1_1

Le contexte *ContextAntsMicro*1_1 étend le contexte *ContextAntsMicro*1_0. Il définit la quantité maximale de nourriture qu'une position de l'environnement peut contenir (*QuantityFoodMax*) ainsi que la charge maximale d'une fourmi (*MaxLoad*). Les axiomes relatifs à ces deux constantes sont donnés par la figure 6.8.

$$axm1 : QuantityFoodMax \geq 1$$
$$axm2 : MaxLoad \in \mathbb{N}$$

Figure 6.8 — Les axiomes du contexte *ContextAntsMicro*1_1

Le contexte *ContextAntsMicro*1_2 **: formalisation des aptitudes d'une fourmi**

Dans ce paragraphe, nous décrivons quelques extraits du contexte *ContextAntsMicro*1_2 et nous commençons par les aptitudes *Favour* et *Weight* permettant à une fourmi d'attribuer des poids pour toutes les positions qui forment son voisinage en fonction de ses perceptions.

$$axm19 : Favour \in Locations \times Locations \nrightarrow 0 \mathinner{..} maxWeight$$
$$axm20 : \forall loc1, loc2 \cdot (loc1 \mapsto loc2) \in dom(Favour) \Rightarrow loc2 \in Next(loc1)$$
$$axm21 : Weight \in 0 \mathinner{..} maxBoundSmell \times 0 \mathinner{..} maxBoundSmell \rightarrow 0 \mathinner{..} maxWeight$$
$$axm22 : \forall c, dir, FoodDist, PheroDist \cdot c \in Locations \wedge dir \in Next(c) \wedge FoodDist \in (Locations \rightarrow \mathbb{N}) \wedge PheroDist \in (Locations \rightarrow \mathbb{N}) \Rightarrow$$
$$Favour(c \mapsto dir) = Weight(FoodPerception(FoodDist)(c \mapsto dir) \mapsto PheromonePerception(PheroDist)(c \mapsto dir))$$

Figure 6.9 — Les axiomes définissant les fonctions *Weight* et *Favour* dans le contexte *ContextAntsMicro*1_2

La fonction *Favour* fait correspondre pour chaque paire de positions une valeur numérique entre zéro et *maxWeight* (*maxWeight* étant un entier naturel non nul représentant le poids maximum qu'une fourmi peut attribuer à une position). La première position dans cette paire représente la position courante de la fourmi, la deuxième représente une position parmi celles accessibles à partir de la position courante (*axm*20).
La détermination de la prochaine position à visiter se base sur les aptitudes de la fourmi à percevoir la nourriture (*FoodPerception*) et la phéromone (*PheromonePerception*) à proximité. Ces aptitudes sont déterminées dans les axiomes donnés par la figure 6.10.
La fourmi est aussi capable de percevoir la phéromone du nid dont elle se sert pour identifier son chemin de retour. Cette aptitude est modélisée par la fonction *NestSmell* définie par les axiomes de la figure 6.11.

Nous avons défini trois niveaux possibles pour les perceptions des fourmis :

- *zero* pour indiquer l'absence de nourriture ou de phéromone à proximité de la fourmi

$axm7 : PheromonePerception \in (Locations \rightarrow \mathbb{N}) \rightarrow (Locations \times Locations \nrightarrow 0 .. maxBoundSmell)$
$axm8 : \forall PheroDist, loc1, loc2 \cdot PheroDist \in (Locations \rightarrow \mathbb{N}) \land (loc1 \mapsto loc2) \in dom(PheromonePerception(PheroDist))$
$\quad \Leftrightarrow loc2 \in Next(loc1) axm9 : FoodPerception \in (Locations \rightarrow \mathbb{N}) \rightarrow (Locations \times Locations \nrightarrow 0 .. maxBoundSmell)$
$axm10 : \forall FoodDist, loc1, loc2 \cdot FoodDist \in (Locations \rightarrow \mathbb{N}) \land (loc1 \mapsto loc2) \in dom(FoodPerception(FoodDist))$
$\quad \Leftrightarrow loc2 \in Next(loc1)$

Figure 6.10 — Les axiomes définissant les aptitudes *FoodPerception* et *PheromonePerception* dans le contexte *ContextAntsMicro*1_2

$axm12 : NestSmell \in Locations \times Locations \nrightarrow 0 .. maxBoundSmell$
$axm13 : \forall loc1, loc2 \cdot (loc1 \mapsto loc2) \in dom(NestSmell) \Leftrightarrow loc2 \in Next(loc1)$

Figure 6.11 — Les axiomes définissant l'aptitude *NestSmell* dans le contexte *ContextAntsMicro*1_2

- *low* pour indiquer que la fourmi perçoit une faible quantité de nourriture ou de phéromone dans son voisinage
- *high* pour indiquer que la nourriture ou la phéromone perçue par la fourmi est élevée

Ces trois niveaux sont décrits formellement dans le contexte *ContextAntsMicro*1_2 par les axiomes donnés par la figure 6.12.

$axm1 : partition(\mathbb{N}, zero, low, high)$
$axm2 : lowBoundSmell \in \mathbb{N} \land maxBoundSmell \in \mathbb{N} \land lowBoundSmell < maxBoundSmell$
$axm3 : \forall p \cdot p = 0 \Leftrightarrow p \in zero$
$axm4 : \forall p \cdot p \in 1 .. lowBoundSmell \Leftrightarrow p \in low$
$axm5 : \forall p \cdot p > lowBoundSmell \land p \leq maxBoundSmell \Leftrightarrow p \in high$

Figure 6.12 — Les axiomes définissant l'aptitude *NestSmell* dans le contexte *ContextAntsMicro*1_2

L'axiome $axm1$ partitionne l'ensemble des entiers naturels en trois sous ensembles *zero*, *low* et *high*. L'axiome $axm2$ détermine les bornes *lowBoundSmell* et *maxBoundSmell* de ces sous-ensembles et la relation d'ordre entre eux ($lowBoundSmell < maxBoundSmell$). Finalement les axiomes $axm3$, $axm4$ et $axm5$ définissent l'appartenance à chacun des sous ensembles *zero*, *low* et *high* en fonction de *lowBoundSmell* et *maxBoundSmell*. Ces trois niveaux permettent d'évaluer les différentes perceptions et ainsi déterminer la position préférée de la fourmi lors de son processus de décision.
Les aptitudes des fourmis étant définies, nous passons à la description de leur utilisation dans les événements *DecideAntsMoveExplore* (figures 6.13) et *DecideAntsMoveBack* (figure 6.14).

Les événements *DecideAntsMoveExplore* **et** *DecideAntsMoveBack*

Ces événements raffinent *DecideAntsMove* en précisant la manière avec laquelle la prochaine position d'une fourmi est déterminée lors d'un déplacement en quête de nourriture et lors d'un déplacement pour retourner au nid.

L'événement *DecideAntsMoveExplore*

Lorsque la fourmi est en quête de nourriture, elle choisit de se déplacer vers une position parmi celles ayant le poids le plus fort qui l'éloigne le plus du nid. L'événement *DecideAntsMoveExplore* (figure 6.13) raffine l'événement *DecideAntsMove* en ajoutant ces contraintes sur la prochaine position de la fourmi.

EVENT DecideAntsMoveExplore
REFINES DecideAntsMove
ANY
$agent$
$nextLoc$
f
$maxFavour$
$maxFavourDom$
$nextDirSet$
$minNestSmell$
$minNestSmellDom$
ns
$comeBack$
WHERE
$grd1 : agent \in Ants$
$grd2 : antStep(agent) = decide$
$grd3 : antMode(agent) = work$
$grd4 : comeBack = FALSE$
$grd5 : nextLoc \in Next(antCurrentLoc(agent))$
$grd6 : f = \{dir \cdot dir \in Next(antCurrentLoc(agent)) | Favour(antCurrentLoc(agent) \mapsto dir)\}$
$grd7 : maxFavour = max(f)$
$grd8 : maxFavourDom = dom(Favour \rhd \{maxFavour\})$
$grd9 : nextDirSet = ran(\{antCurrentLoc(agent)\} \lhd maxFavourDom)$
$grd10 : ns = \{dir \cdot dir \in nextDirSet | NestSmell(antCurrentLoc(agent) \mapsto dir)\} \land ns \neq \varnothing$
$grd11 : minNestSmell = \{min(ns)\}$
$grd12 : minNestSmellDom = dom(NestSmell \rhd minNestSmell)$
$grd13 : nextLoc \in ran(\{antCurrentLoc(agent)\} \lhd minNestSmellDom)$
THEN
$act1 : antStep := antStep \Leftarrow \{agent \mapsto perform\}$
$act2 : antNextAction := antNextAction \Leftarrow \{agent \mapsto move\}$
$act3 : nextLocation(agent) := nextLoc$
$act4 : paw := paw \Leftarrow \{agent \mapsto move \mapsto enable\}$
END

Figure 6.13 — L'événement *DecideAntsMoveExplore* de la machine *AntsDecisions*1

Les paramètres de l'événement *DecideAntsMoveExplore* sont interprétés comme suit :

- agent : la fourmi concernée (paramètre défini dans *grd*1).
- comeBack : ce paramètre indique que la fourmi doit poursuivre l'exploration lorsqu'il est à $faux$ (paramètre défini dans *grd*4).
- f : l'ensemble des poids que la fourmi a attribué pour les différentes positions qui lui sont accessibles à partir de sa position courante. Cet ensemble est défini par la garde (paramètre défini dans *grd*6).
- maxFavour : la valeur du poids maximum (paramètre défini dans *grd*7).
- maxFavourDom : l'ensemble des positions ayant le poids maximum (paramètre défini dans *grd*8).

- nextDirSet : l'ensemble des positions se trouvant dans le voisinage de la fourmi et ayant le poids maximum (paramètre défini dans *grd*9).
- ns : l'ensemble des gradients du nid que la fourmi a attribué à chacune des directions possibles ayant le poids maximum (paramètre défini dans *grd*10).
- minNestSmell : la valeur minimale du gradient du nid (paramètre défini dans *grd*11).
- minNestSmellDom : l'ensemble des positions accessibles à la fourmi ayant le gradient du nid minimum et le poids maximum (paramètre défini dans *grd*12).
- nextLoc : la position vers laquelle la fourmi décide de se déplacer parmi l'ensemble *minNestSmellDom* (paramètre défini dans *grd*13).

L'événement *DecideAntsMoveBack*

L'événement *DecideAntsMoveBack* (figure 6.14) raffine l'événement *DecideAntsMove* en ajoutant les gardes qui permettent de spécifier les contraintes sur la prochaine position de la fourmi lors de son retour au nid. Les paramètres de l'événement *DecideAntsMoveBack* sont

EVENT DecideAntsMoveBack
REFINES DecideAntsMove

ANY
agent
nextLoc
ns
maxNestSmell
maxNestSmellDom
comeBack
WHERE
$grd1 : agent \in Ants$
$grd2 : antStep(agent) = decide$
$grd3 : antMode(agent) = work$
$grd4 : nextLoc \in Locations$
$grd5 : nextLoc \in Next(antCurrentLoc(agent))$
$grd6 : ns = \{dir \cdot dir \in Next(antCurrentLoc(agent)) | NestSmell(antCurrentLoc(agent) \mapsto dir)\}$
$grd7 : maxNestSmell = \{max(ns)\}$
$grd8 : maxNestSmellDom = dom(NestSmell \rhd maxNestSmell)$
$grd9 : nextLoc \in ran(\{antCurrentLoc(agent)\} \lhd maxNestSmellDom)$
$grd10 : comeBack = TRUE$
THEN
$act1 : antStep := antStep \Leftarrow \{agent \mapsto perform\}$
$act2 : antNextAction := antNextAction \Leftarrow \{agent \mapsto move\}$
$act3 : nextLocation(agent) := nextLoc$
$act4 : paw := paw \Leftarrow \{agent \mapsto move \mapsto enable\}$
END

Figure 6.14 — L'événement *DecideAntsMoveBack* de la machine *AntsDecisions*1

interprétés comme suit :

- agent : la fourmi concernée (*grd*1).
- nextLoc : la position vers laquelle la fourmi décide de se déplacer (*grd*4, *grd*5, *grd*9).

- ns : l'ensemble des gradients du nid que la fourmi a attribué à chacune des directions possibles (*grd*6).
- maxNestSmell : la valeur maximale du gradient du nid (*grd*7).
- maxNestSmellDom : l'ensemble des positions accessibles à la fourmi ayant le gradient du nid maximum (*grd*8).
- comeBack : ce paramètre indique que la fourmi doit retourner au nid lorsqu'il est à *vrai* (*grd*10).

Le raffinement de la machine *AntsDecisions*0 par la machine *AntsDecisions*1, donne lieu à 16 obligations de preuve dont la majorité sont relatives à la vérification de la bonne définition des gardes ajoutées. Parmi ces obligations de preuve, 10 ont été prouvées automatiquement et 6 prouvées manuellement à l'aide du prouveur interactif de *Rodin*.

Dans cette section, nous avons déterminé les décisions des fourmis en utilisant le premier raffinement du patron P_MAS. Nous passons dans ce qui suit à la modélisation des actions par l'application du deuxième raffinement de ce patron.

6.2.3 Modélisation des actions des fourmis

La modélisation des actions des fourmis est réalisée en divisant l'événement $Perform$ du modèle initial en plusieurs événements dont chacun représente une action. Ce raffinement se fait conformément au deuxième raffinement du patron P_MAS. L'application de ce raffinement dans le cadre de notre cas d'étude nécessite l'application du patron P_MAS quatre fois puis l'utilisation de la fusion deux fois. La première fusion a pour but de regrouper les événements des actions dans une même machine. La deuxième fusion permet de regrouper les règles de décisions et les actions dans une seule machine. Ces étapes sont illustrées par la figure 6.15 et sont détaillées par les paragraphes qui suivent.

La réalisation du deuxième raffinement du patron P_MAS nécessite les cinq étapes décrites par la figure 6.15. La première consiste à appliquer le patron P_MAS sur la machine du modèle initial *AntsInitModel* quatre fois afin de modéliser les quatre actions suivantes :

- $PerformAntsDropFood$: correspond au dépôt de nourriture dans le nid,
- $PerformAntsDropPheromone$: correspond au dépôt de phéromone sur le chemin de retour,
- $PerformAntsHarvestFood$: modélise une récolte de nourriture,
- $PerformAntsMove$: modélise la réalisation un déplacement.

L'application du patron P_MAS donne lieu à quatre machines dont chacune raffine la machine initiale *AntsInitModel* et décrit une action parmi les quatre actions qu'une fourmi peut réaliser. Dans la seconde étape, ces quatre machines sont fusionnées pour

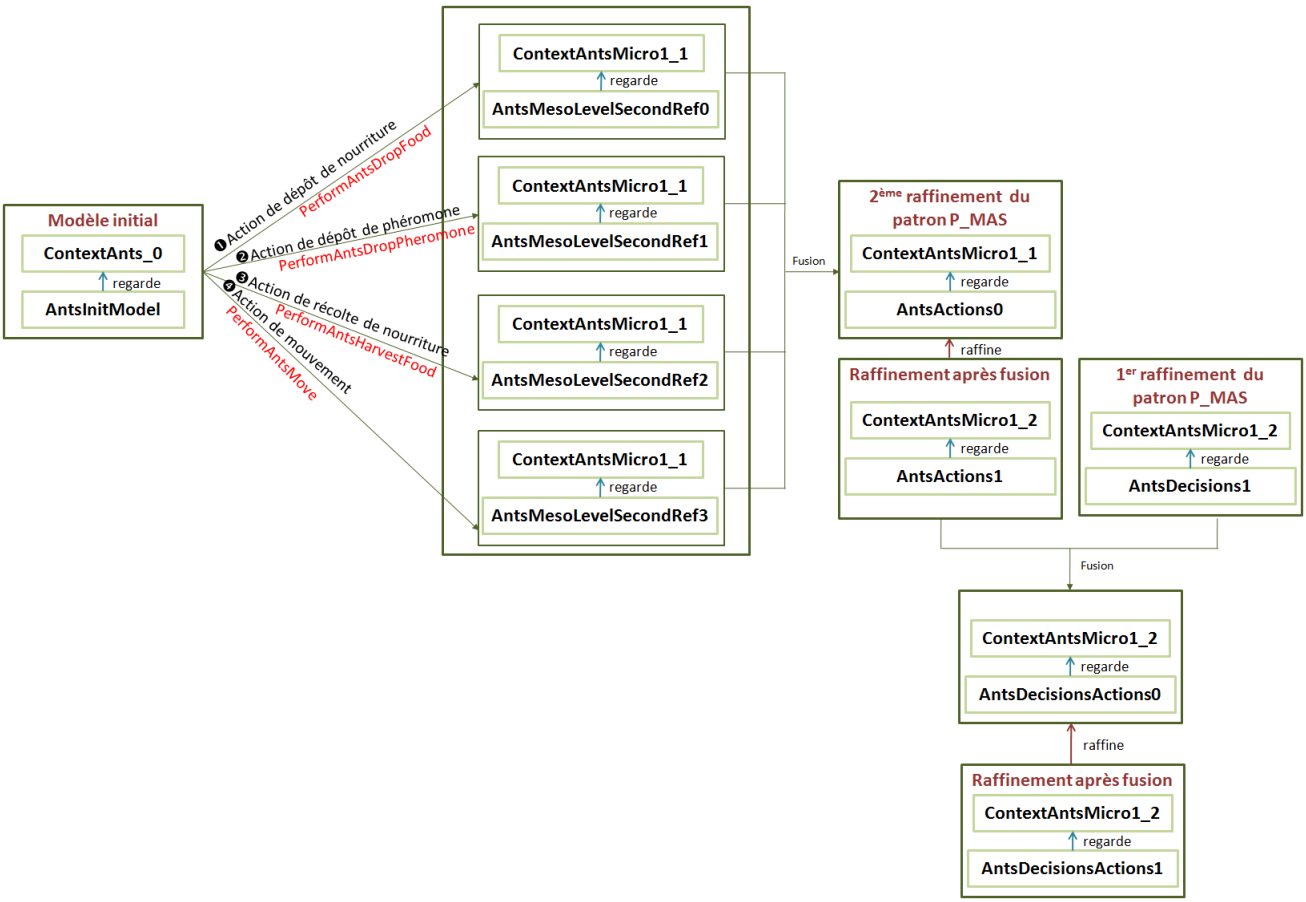

Figure 6.15 — Les différents modèles utilisés pour décrire les actions des fourmis et les fusionner avec les règles décisionnelles

donner lieu à la machine *AntsActions*0. La troisième étape consiste à raffiner l'événement *PerformAntsMove* de la machine *AntsActions*0 en le divisant en l'événement *PerformAntsMoveExplore* (la fourmi fait un déplacement pour explorer son environnement) et l'événement *PerformAntsMoveBack* (La fourmi fait un déplacement pour rejoindre son nid).
La quatrième étape permet de regrouper les événements de décisions et ceux des actions dans une seule machine *AntsDecisionsActions*0 en fusionnant la machine *AntsActions*1 avec la machine *AntsDecisions*1 (obtenue suite à l'application du premier raffinement du patron *P_MAS*). La dernière étape permet d'établir les liens entre les actions et les décisions d'une fourmi en raffinant la machine *AntsDecisionsActions*0 par la machine *AntsDecisionsActions*1.

6.2.3.1 Exemple d'illustration : application du deuxième raffinement du patron *P_MAS* pour raffiner l'événement *PerformAntsMove*

Dans cette section, nous décrivons l'utilisation du deuxième raffinement du patron *P_MAS* pour modéliser l'événement *PerformAntsMove* (la flèche étiquetée ❹ sur la figure 6.15). La correspondance entre les éléments du modèle initiale *InitialModel* du patron avec

ceux du modèle abstrait de l'application *AntsInitModel* est identique à celle utilisée pour le premier raffinement (tableau 6.1).

Le raffinement de l'événement $Perform Ants$ (de la machine *AntsInitModel*) par l'événement $Perform AntsMove$ dans la machine $AntsMesoLevelSecondRef3$ est réalisé automatiquement. Comme indiqué dans l'événement $Perform Act$ du patron *P_MAS* (figure 4.11), ce raffinement consiste à introduire les variables représentant l'action à entreprendre (*nextAgentAction*), l'actionneur devant être activé pour la réalisation de l'action concernée (*agentActuators*) et la propriété de l'agent (*agentActionProp*) qui sera modifiée par l'action en question. La définition de ces éléments pour l'événement $Perform AntsMove$ est donnée par le tableau 6.4.

	Éléments du modèle générique MesoLevel (section 4.2.2)	**Éléments du système des fourmis fourrageuses** $AntsMesoLevelSecondRef3$
Variables	nextAgentAction agentActuators agentActionProp	antNextAction paw currentLoc
Evènements	PerformAct	PerformAntsMove

Tableau 6.4 — Renommage des éléments du modèle concret du patron *P_MAS* lors du raffinement de l'événement $Perform Ants$

L'événement $Perform AntsMove$ obtenu suite à l'incorporation du raffinement de l'événement $Perform Ants$ selon le patron *P_MAS* est donné par la figure 6.16. Les cinq premières

EVENT PerformAntsMove
REFINES PerformAnts
ANY
  $agent$
  $newLoc$
WHERE
  $grd1 : agent \in Ants$
  $grd2 : antStep(agent) = perform$
  $grd3 : antMode(agent) = work$
  $grd4 : antNextAction(agent) = move$
  $grd5 : paw(agent \mapsto move) = enable$
  $grd6 : newLoc \in Next(currentLoc(agent))$
  $grd7 : newLoc \notin Obstacles$
WITH
  $ant : ant = agent$
THEN
  $act1 : antStep := antStep \lhd\!\!+ \{agent \mapsto perform\}$
  $act2 : antMode := antMode \lhd\!\!+ \{agent \mapsto wait\}$
  $act3 : currentLoc(agent) := newLoc$
END

Figure 6.16 — L'événement $Perform eAntsMove$ de la machine $AntsMesoLevelSecondRef3$

gardes de l'événement $Perform AntsMove$ sont obtenues directement suite à l'application du patron de raffinement. Elles assurent que l'agent est bien dans la phase d'action de son

cycle de vie (*grd*2), qu'il est en mode *work* (*grd*3), que l'action qu'il doit réaliser est bien un déplacement (*grd*4) et que ses pattes sont activées (*grd*5). Les deux dernières gardes sont ajoutées manuellement pour garantir que la position vers laquelle la fourmi va se déplacer est lui est accessible à partir de sa position courante (*grd*6) et qu'elle ne contient pas d'obstacles (*grd*7).
Les événements d'action peuvent modifier, en plus des propriétés intrinsèques aux agents (tel le cas de l'événement $PerformAntsMove$ qui modifie la position courante de la fourmi), les propriétés des éléments de l'environnement. Ainsi, lors de la modélisation de certains événements d'action, nous introduisons des variables globales qui servent pour décrire l'état de l'environnement. Dans le cas d'étude présenté, il s'agit d'ajouter les variables $QuantityFood$ et $DensityPheromone$ qui renseignent respectivement sur la quantité de nourriture et la densité de phéromone dans les différentes positions de l'environnement.
La variable $QuantityFood$ est introduite dans la machine $AntsMesoLevelSecondRef2$ lors de la définition de l'événement $PerformAntsHarvestFood$ décrivant une action de récolte de nourriture. Formellement, cette variable est une fonction totale de l'ensemble des positions vers l'intervalle $0..QuantityFoodMax$ et fait associer à chaque position la quantité de nourriture qu'elle contient. La variable $QuantityFood$ doit vérifier l'invariant :

$$QuantityFood \in Locations \rightarrow 0..QuantityFoodMax$$

La variable $DensityPheromone$ est introduite pour modéliser la densité de phéromone dans les différentes positions de l'environnement. Elle évolue suite à l'action de dépôt de phéromone formalisée par l'événement $PerformAntsDropPheromone$ dans la machine $AntsMesoLevelSecondRef1$. La variable $DensityPheromone$ représente une fonction totale de l'ensemble des positions vers l'ensemble des entiers naturels et décrit la répartition de la phéromone dans l'environnement. La variable $DensityPheromone$ doit vérifier l'invariant :

$$DensityPheromone \in Locations \rightarrow \mathbb{N}$$

L'évolution de cette variable peut être due soit à une action de dépôt de phéromone (l'événement $PerformAntsDropPheromone$) soit à une action d'évaporation faite par l'environnement. A ce niveau, nous supposons que seules les fourmis sont responsables de la modification de cette variable et nous considérons l'action de l'environnement lors la preuve de la résilience du système (troisième phase du processus RefProSOMAS).
Le raffinement des actions des fourmis conformément au patron P_MAS donne lieu à quatre machines. Le tableau 6.5 récapitule des statistiques sur les obligations de preuves générées pour les quatre machines obtenues suite à l'application du patron P_MAS pour modéliser les événements des actions des fourmis.

6.2.3.2 La fusion des actions dans une seule machine

Cette phase de fusion a pour but de regrouper tous les événements qui décrivent les actions des fourmis dans une seule machine. La fusion donne lieu à la machine $AntsActions0$ dont un extrait est donné par la figure 6.17.

Dans la machine $AntsActions0$, les événements $PerceiveAnts$, $DecideAnts$ et $EnvironmentChange$ sont ceux de la machine $AntsInitModel$. Les événe-

	Nombre total d'Obligations de Preuve (OP)	Nombre d'OP déchargées automatique-ment	Nombre d'OP déchargées manuellement
AntsMesoLevelSecondRef0	27	25	2
AntsMesoLevelSecondRef1	28	25	3
AntsMesoLevelSecondRef2	34	28	3
AntsMesoLevelSecondRef3	27	25	2

Tableau 6.5 — Statistiques concernant les obligations de preuves des machines générées suite à l'application de patron *P_MAS* pour modéliser les actions des fourmis

*Machine AntsActions*0
REFINES AntsInitModel
SEES
*ContextAntsMicro*1_1
VARIABLES
antStep
antMode
currentLoc
nextLocation
antNextAction
paw
QuantityFood
DensityPheromone
...
INVARIANTS
...
EVENTS
INITIALISATION
EVENT PerceiveAnts
EVENT DecideAnts
EVENT PerformAntsMove
EVENT PerformAntsDropFood
EVENT PerformAntsHarvestFood
EVENT PerformAntsDropPheromone
EVENT PerformAnts
EVENT EnvironmentChange
END

Figure 6.17 — Extrait de la machine *AntsActions*0

ments *PerformAntsMove*, *PerformAntsDropFood*, *PerformAntsHarvestFood* et *PerformAntsDropPheromone* qui raffinent l'événement *PerformAnts* sont obtenus en appliquant le deuxième raffinement du patron *P_MAS*. Pour prouver la correction de la fusion, 89 obligations de preuve ont été générées dont 84 ont été déchargées automatiquement et 2 ont été prouvées manuellement à l'aide du prouveur interactif de *Rodin*.

6.2.3.3 Le raffinement de l'événement $PerformAntsMove$ dans la machine $AntsActions1$

Cette étape est nécessaire pour ramener la machine $AntsActions0$ au même niveau de raffinement que la machine $AntsDecisions1$ avant de les fusionner ensemble. Ce raffinement consiste à diviser l'événement $PerformAntsMove$ en les deux événements suivants :

- $PerformAntsMoveExplore$: la fourmi réalise un déplacement pour explorer son environnement,
- $PerformAntsMoveBack$: la fourmi réalise un déplacement dans la direction du nid.

A titre d'exemple, la figure 6.18 illustre l'événement $PerformAntsMoveExplore$.

EVENT PerformAntsMoveExplore
REFINES PerformAntsMove
ANY
  $agent$
  $newLoc$ WHERE
  $grd1 : agent \in Ants$
  $grd2 : antStep(agent) = perform$
  $grd3 : antMode(agent) = work$
  $grd4 : antNextAction(agent) = move$
  $grd5 : paw(agent \mapsto move) = enable$
  $grd6 : newLoc \notin Obstacles$
WITH
  $newLoc : newLoc = nextLocation(agent)$
THEN
  $act1 : antStep := antStep \lhd\!\!+ \{agent \mapsto perform\}$
  $act2 : antMode := antMode \lhd\!\!+ \{agent \mapsto wait\}$
  $act3 : currentLoc(agent) := newLoc$
END

Figure 6.18 — L'événement $PerformeAntsMoveExplore$ de la machine $AntsActions1$

6.2.3.4 La fusion de la machine $AntsActions1$ avec la machine $AntsDecisions1$

Cette étape de fusion a pour objectif de regrouper les événements des décisions et les événements des actions dans une seule machine afin de préparer l'établissement du lien entre les décisions et les actions. La figure 6.19 montre un extrait de la machine $AntsDecisionsActions0$ obtenue suite à la fusion des machines $AntsActions1$ et $AntsDecisions1$. Pour prouver la correction de la fusion, 122 obligations de preuve ont été générées dont 115 ont été déchargées automatiquement et 2 ont été prouvées manuellement à l'aide du prouveur interactif de *Rodin*.

6.2.3.5 Le raffinement de la machine $AntsDecisionsActions0$ par la machine $AntsDecisionsActions1$

Cette étape permet d'établir le lien entre les règles décisionnelles et les actions des fourmis. Concrètement, établir ce lien se fait en exprimant les paramètres locaux abstraits d'un événement à l'aide des variables déclarées au niveau de la machine. En *B-événementiel*, ceci se traduit par le raffinement de l'événement en question et la création de témoins

*Machine AntsDecisionsActions*0
REFINES AntsInitModel
SEES
*ContextAntsMicro*1_2
VARIABLES
antStep
antMode
...
INVARIANTS
...
EVENTS
INITIALISATION
EVENT PerceiveAnts
EVENT DecideAntsMoveExplore
EVENT DecideAntsMoveBack
EVENT DecideAntsDropFood
EVENT DecideAntsHarvestFood
EVENT DecideAntsDropPheromone
EVENT PerformAntsMoveExplore
EVENT PerformAntsMoveBack
EVENT PerformAntsDropFood
EVENT PerformAntsHarvestFood
EVENT PerformAntsDropPheromone
EVENT PerformAnts
EVENT EnvironmentChange
END

Figure 6.19 — Extrait de la machine *AntsDecisionsActions*0

(appelés aussi *Witness*). A titre d'exemple, nous présentons le raffinement de l'événement $PerformAntsMoveExplore$ décrit par la figure 6.20 modélisant le déplacement d'une fourmi vers une position donnée par le paramètre *newLoc*. La position vers laquelle la fourmi va se déplacer est déterminée par l'événement *DecideAntsMove* et elle est contenue dans la variable *nextLocation*. L'événement $PerformAntsMoveExplore$ est alors raffiné en éliminant le paramètre *nextLoc* de sa liste de paramètre et en ajoutant le témoin $nextLoc : nextLoc = nextLocation(agent)$. Les actions sont également modifiées comme le montre la figure 6.20. L'événement $PerformAntsDropPheromone$ qui modélise le dépôt de phéromone dans une position *loc* est raffiné également pour remplacer ce paramètre *loc* par la position courante de la fourmi au moyen du témoin $loc : loc = currentLoc(agent)$.
Les événements de décision et d'action étant regroupés ensemble, il devient possible de prouver la propriété de non blocage à la phase de décision. Cette propriété garantit qu'une fois une fourmi a pris une décision, elle ne se bloque pas dans cette étape, c'est à dire qu'elle est capable d'exécuter l'action qu'elle a choisi ou de débuter un nouveau cycle. Dans notre cas d'étude, il existe deux situations qui peuvent empêcher une fourmi de réaliser l'action qu'elle a choisi. La première situation survient lorsqu'elle décide de récolter de la nourriture sur une position particulière et qu'elle n'arrive pas à exécuter l'action de récolte car cette nourriture a été déjà récoltée par une autre fourmi. La deuxième situation a lieu lorsque la fourmi décide de se déplacer sur une position et qu'elle n'arrive pas à faire ce déplacement car un obstacle vient se poser sur cette position.
A ce niveau de raffinement, nous considérons uniquement la première situation car elle représente le résultat des actions des fourmis sur l'environnement. La propriété de non blocage qu'il faut prouver est alors formaliser avec le théorème *LocProp*2*HarvestFood* donné

EVENT PerformAntsMoveExplore
REFINES PerformAntsMove
ANY
$agent$
WHERE
$grd1 : agent \in Ants$
$grd2 : antStep(agent) = perform$
$grd3 : antMode(agent) = work$
$grd4 : antNextAction(agent) = move$
$grd5 : paw(agent \mapsto move) = enable$
$grd6 : nextLocation(agent) \notin Obstacles$
WITH
$newLoc : newLoc = nextLocation(agent)$
THEN
$act1 : antStep := antStep \lhd\!\!+ \{agent \mapsto perform\}$
$act2 : antMode := antMode \lhd\!\!+ \{agent \mapsto wait\}$
$act3 : currentLoc(agent) := nextLocation(agent)$
END

Figure 6.20 — L'événement $PerformeAntsMoveExplore$ après son raffinement

par la figure 6.21.

$$LocProp2HarvestFood : \forall a \cdot (a \in Ants \land antNextAction(a) = harvestFood \land antStep(a) = perform \land antMode(a) = work \land mandible(a \mapsto harvestFood) = enable \land (QuantityFood(currentLoc(a)) \geq 1 \lor QuantityFood(currentLoc(a)) = 0)) \Rightarrow (antNextAction(a) = harvestFood \land antStep(a) = perform \land antMode(a) = work \land mandible(a \mapsto harvestFood) = enable \land QuantityFood(currentLoc(a)) \geq 1) \lor (antNextAction(a) = harvestFood \land antStep(a) = perform \land antMode(a) = work \land mandible(a \mapsto harvestFood) = enable)$$

Figure 6.21 — La propriété de non blocage d'une fourmi après avoir décidé de récolter de la nourriture

Pour prouver la correction de cette étape de raffinement, 39 obligations de preuve ont été générées dont 36 ont été déchargées automatiquement et une a été prouvée manuellement à l'aide du prouveur interactif de *Rodin*.

6.2.4 Modélisation des opérations de perception des fourmis

Le troisième raffinement décrit par le patron P_MAS permet de spécifier les opérations de perception des fourmis. La figure 6.22 montre les différents modèles utilisés pour obtenir une description complète du comportement nominal des fourmis.

6.2.4.1 Application du patron P_MAS

Dans cette section, nous décrivons le troisième raffinement du patron P_MAS pour modéliser l'événement *PerceiveAnts* (la flèche étiquetée "Application du patron P_MAS" sur la figure 6.22). La correspondance entre les éléments du modèle abstrait du patron avec ceux du modèle abstrait de l'application est celle présentée dans le tableau 6.1. Le raffinement de l'événement *PerceiveAnts* (de la machine $AntsInitModel$) par l'événement $PerceiveEnvironmentAnts$ dans la machine $AntsMesoLevelThirdRef$ est réalisé automatiquement.

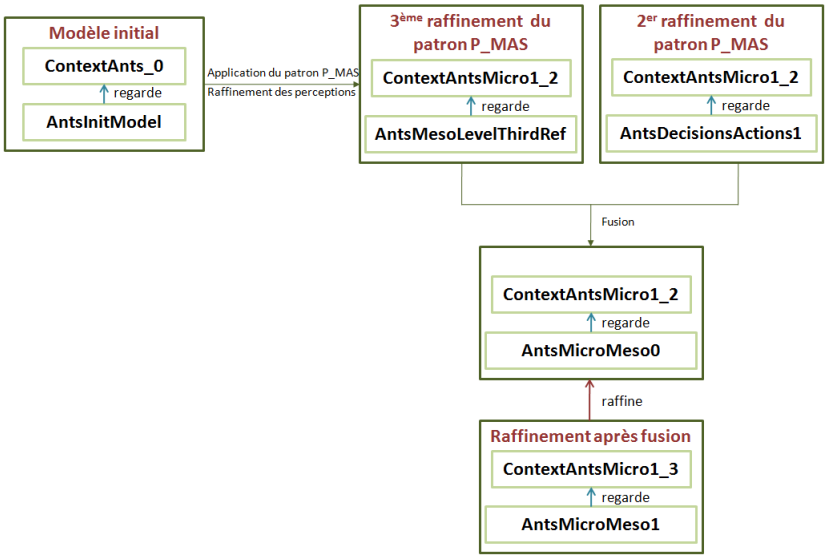

Figure 6.22 — Les différents modèles utilisés pour décrire le comportement nominal des fourmis

Les opérations de perception sont réalisées par les capteurs de l'agent et permettent essentiellement de mettre à jour les représentations partielles de l'environnement. Elles permettent aussi la mise à jour des propriétés internes aux agents. Cette étape de raffinement consiste alors à introduire les variables représentant les représentations partielles des fourmis par l'instanciation de la variable *agentRep*, le capteur devant être activé pour obtenir ces représentation en instanciant la variable *agentSensor* et l'ensemble des propriétés de la fourmi (*agentIntProp*) mises à jour par l'opération de perception. La définition de ces éléments pour l'événement *PerceiveEnvironmentAnts* est donnée par le tableau 6.6.

	Éléments du modèle générique *MesoLevel* (section 4.2.2)	**Éléments du système des fourmis fourrageuses (*AntsMesoLevelThirdRef*)**
Variables	agentRep agentSensors agentIntProp	food , pheromone sensibleCorn comeBackAnt , dropPheroBack
Evènements	PerceiveEnvironment	PerceiveEnvironmentAnts

Tableau 6.6 — Renommage des éléments du modèle concret du patron *P_MAS* lors du raffinement de l'événement *PerceiveAnts*

L'événement *PerceiveEnvironmentAnts* obtenu suite à l'incorporation du raffinement de l'événement *PerceiveAnts* selon le patron *P_MAS* est donné par la figure 6.23.

La plateforme *Rodin* a généré 35 obligations de preuve permettant de prouver la correction de la génération de la machine *AntsMesoLevelThirdRef*. Parmi ces obligations de preuve, 26 ont été prouvées automatiquement et 3 ont été prouvées de manière interactive.

EVENT PerceiveEnvironmentAnts
REFINES PerceiveAnts
ANY
$agent$
fp
php
$foodDistribution$
$densityPheromone$
$loadAnt$
loc
WHERE
$grd1 : agent \in Ants$
$grd2 : antStep(agent) = perceive$
$grd3 : antMode(agent) = work$
$grd4 : sensibleCorn(agent) = enable$
$grd5 : loc \in Locations$
$grd6 : loadAnt \in \mathbb{N}$
$grd7 : foodDistribution \in Locations \rightarrow \mathbb{N}$
$grd8 : densityPheromone \in Locations \rightarrow \mathbb{N}$
$grd9 : fp \in (Locations \times Locations) \nrightarrow \mathbb{N}$
$grd10 : fp = FoodPerception(foodDistribution)$
$grd11 : php \in (Locations \times Locations) \nrightarrow \mathbb{N}$
$grd12 : php = PheromonePerception(densityPheromone)$
WITH
$ant : ant = agent$
THEN
$act1 : antStep := antStep \Leftarrow \{agent \mapsto decide\}$
$act2 : pheromone(agent) := \{dir \cdot dir \in Next(loc) | dir \mapsto php(loc \mapsto dir)\}$
$act3 : food(agent) := \{dir \cdot dir \in Next(loc) | dir \mapsto fp(loc \mapsto dir)\}$
$act4 : comeBackAnt(agent) := bool(loadAnt = MaxLoad)$
$act5 : dropPheroBack(agent) := bool((loc \neq Nest \wedge loadAnt = MaxLoad \wedge foodDistribution(loc) > 1) \vee$
$(loc \neq Nest \wedge loadAnt = MaxLoad \wedge dropPheroBack(agent) = TRUE))$
END

Figure 6.23 — L'événement *PerceiveEnvironmentAnts* de la machine *AntsMesoLevelThirdRef*

6.2.4.2 Obtention du comportement des fourmis dans la machine *CompoAntMeso*1

La génération du comportement nominal correct des fourmis nécessite deux étapes : une étape de fusion et une étape de raffinement.

La fusion

L'étape de fusion permet de regrouper l'événement *PerceiveEnvironmentAnts* avec les événements d'action et de décision développés dans la section précédente et regroupés dans la machine *AntsDecisionsActions*1. Un extrait de la machine *AntsMicroMeso*0 résultat de la fusion est donné dans la figure 6.24.

Il est à noter qu'à cette étape de raffinement, les événements *DecideAntsMoveExplore* et *PerformAntsMoveExplore* modélisent respectivement une décision de faire un déplacement et une action de déplacement indépendamment des perceptions de la fourmi. Dans la prochaine étape de raffinement, les perceptions sont pris en compte.

Le raffinement

L'étape de raffinement permet de créer les liens entre l'événement de perception et les

Machine AntsMicroMeso0
REFINES AntsInitModel
SEES
ContextAntsMicro1_2
VARIABLES
antStep
antMode
...
INVARIANTS
...
EVENTS
INITIALISATION
EVENT PerceiveEnvironmentAnts
EVENT DecideAntsMoveExplore
EVENT DecideAntsDropFood
EVENT DecideAntsHarvestFood
EVENT DecideAntsDropPheromone
EVENT PerformAntsMoveExplore
EVENT PerformAntsDropFood
EVENT PerformAntsHarvestFood
EVENT PerformAntsDropPheromone
EVENT PerformAnts
EVENT EnvironmentChange
END

Figure 6.24 — Extrait de la machine *AntsMicroMeso0*

événements de décision et de montrer la propriété de non blocage à la phase de perception. L'événement *DecideAntsMoveExplore* est divisé en les trois événements suivants :

- *DecideAntsMoveExploreFollowFood* : décider de faire un déplacement dans la direction de la nourriture perçue,
- *DecideAntsMoveExploreFollowPheromone* : décider de faire un déplacement dans la direction de la phéromone perçue,
- *DecideAntsMoveExploreRandom* : décider de faire un déplacement vers une position choisie aléatoirement,

A titre d'exemple, nous présentons dans la figure 6.25 l'événement *DecideAntsMoveExploreFollowFood*.

L'événement *PerformAntsMoveExplore* est également divisé en les trois événements suivants :

- *PerformAntsMoveExploreFollowFood* : effectuer un déplacement dans la direction de la nourriture perçue,
- *PerformAntsMoveExploreFollowPheromone* : effectuer un déplacement dans la direction de la phéromone perçue,
- *PerformAntsMoveExploreRandom* : effectuer un déplacement vers une position choisie aléatoirement,

EVENT DecideAntsMoveExploreFollowFood
REFINES DecideAntsMoveExplore
ANY
$agent$
$nextLoc$
f
$maxFavour$
$maxFavourDom$
$nextDirSet$
$minNestSmell$
$minNestSmellDom$
ns
$comeBack$
WHERE
$grd1 : agent \in Ants$
$grd2 : antStep(agent) = decide$
$grd3 : antMode(agent) = work$
$grd4 : comeBack = FALSE$
$grd5 : nextLoc \in Next(antCurrentLoc(agent))$
$grd6 : f = \{dir \cdot dir \in Next(antCurrentLoc(agent)) | Favour(antCurrentLoc(agent) \mapsto dir)\}$
$grd7 : maxFavour = max(f)$
$grd8 : maxFavourDom = dom(Favour \rhd \{maxFavour\})$
$grd9 : nextDirSet = ran(\{antCurrentLoc(agent)\} \lhd maxFavourDom)$
$grd10 : ns = \{dir \cdot dir \in nextDirSet | NestSmell(antCurrentLoc(agent) \mapsto dir)\} \land ns \neq \varnothing$
$grd11 : minNestSmell = \{min(ns)\}$
$grd12 : minNestSmellDom = dom(NestSmell \rhd minNestSmell)$
$grd13 : nextLoc \in ran(\{antCurrentLoc(agent)\} \lhd minNestSmellDom)$
$grd14 : food(agent)(nextLoc) \notin zero \land food(agent)(nextLoc) > otherAnts(agent)(nextLoc)$ THEN
$act1 : antStep := antStep \lhd\!\!+ \{agent \mapsto perform\}$
$act2 : antNextAction := antNextAction \lhd\!\!+ \{agent \mapsto moveFollowFood\}$
$act3 : nextLocation(agent) := nextLoc$ //Copy from pattern refinement
$act4 : paw := paw \lhd\!\!+ \{agent \mapsto move \mapsto enable\}$
END

Figure 6.25 — L'événement *DecideAntsMoveExploreFollowFood* de la machine *AntsMicroMeso*1

La figure 6.26 illustre l'événement *PerformAntsMoveExploreFollowFood*. La seule modification par rapport à l'événement *PerformAntsMoveExplore* est au niveau de la garde *grd*4 qui donne plus de précision concernant la nature de l'action de déplacement à entreprendre : il s'agit d'un déplacement dans la direction de la nourriture. La correction de ce raffinement a été prouvée en déchargeant 34 obligations de preuve dont 22 d'entre elles ont été prouvées automatiquement et 12 ont été prouvées à l'aide du prouveur interactif de *Rodin*.

La machine *AntsMicroMeso*1 obtenue dans la première phase du processus RefProSOMAS est une modélisation du comportement des fourmis. Cette phase garantit le non blocage des fourmis dans les différentes étapes de leur cycle de vie ainsi que la préservation des invariants reliées à l'application. Cependant, aucune garantie concernant la convergence vers l'objectif global et la résilience n'est encore assurée. Les deux prochaines sections traitent la preuve de ces propriétés globales.

EVENT PerformAntsMoveExploreFollowFood
REFINES PerformAntsMoveExplore
ANY
  $agent$
WHERE
  $grd1 : agent \in Ants$
  $grd2 : antStep(agent) = perform$
  $grd3 : antMode(agent) = work$
  $grd4 : antNextAction(agent) = moveFollowFood$
  $grd5 : paw(agent \mapsto move) = enable$
  $grd7 : nextLocation(agent) \notin Obstacles$
WITH
  $newLoc : newLoc = nextLocation(agent)$
THEN
  $act1 : antStep := antStep \lhd\!\!+ \{agent \mapsto perform\}$
  $act2 : antMode := antMode \lhd\!\!+ \{agent \mapsto wait\}$
  $act3 : currentLoc(agent) := nextLocation(agent)$
END

Figure 6.26 — L'événement $PerformAntsMoveExploreFollowFood$ de la machine $AntsMicroMeso1$

6.3 Phase N°2 : preuve de la convergence

L'objectif de cette phase est de prouver que le comportement modélisé à la phase précédente permet aux fourmis d'atteindre leur objectif global (la collecte de toute la nourriture au nid) en supposant qu'il n'y a pas de perturbations en provenance de l'environnement. Cette phase est guidée par le patron P_CONV et elle nécessite la spécification de l'événement observateur ainsi que la preuve de la terminaison de chacun des événements décrivant les actions des fourmis.

6.3.1 L'événement observateur

L'événement observateur est le seul événement responsable de détecter si le système a réussi à réaliser sa fonctionnalité globale. Il s'agit d'un événement particulier sans action et dont la garde décrit l'état du système lorsqu'il atteint sa fonctionnalité globale.
Pour l'exemple des fourmis fourrageuses, nous avons considéré que la fonctionnalité globale du système est de ramener la nourriture, initialement dispersée dans l'environnement, vers le nid. Ainsi, la garde de l'événement observateur appelé $AllFoodAtNest$ est décrite par l'expression suivante :

$$\forall loc \cdot loc \in Locations \setminus \{Nest\} \Rightarrow QuantityFood(loc) = 0 \wedge$$
$$TotalFood(InitFoodDist \mapsto Locations) = QuantityFood(Nest)$$

Dans cette expression, la fonction $QuantityFood$ donne pour chaque position de la grille (l'environnement) la quantité de nourriture qu'elle contient. La fonction $TotalFood$ permet de calculer la somme des quantité de nourriture dans l'environnement et elle est définie par les axiomes suivants :

$axm1 : TotalFood \in ((Locations \to \mathbb{N}) \times (\mathbb{P}(Locations))) \to \mathbb{N}$
$axm2 : \forall foodDist, ls \cdot foodDist \in Locations \to \mathbb{N} \land ls = (dom(foodDist \rhd \{0\})) \Rightarrow TotalFood(foodDist \mapsto ls) = 0$
$axm3 : \forall foodDist, loc \cdot foodDist \in Locations \to \mathbb{N} \land loc \in Locations \Rightarrow TotalFood(foodDist \mapsto \{loc\}) = foodDist(loc)$
$axm4 : \forall foodDist, l, ls \cdot foodDist \in Locations \to \mathbb{N} \land l \in Locations \land ls \in \mathbb{P}(Locations) \land l \in ls \Rightarrow$
$TotalFood(foodDist \mapsto ls) = foodDist(l) + TotalFood(foodDist \mapsto (ls \setminus \{l\}))$

L'axiome $axm1$ définit le domaine et le co-domaine de la fonction totale $TotalFood$. Cette fonction prend en entrée une distribution donnée de nourriture (de type $(Locations \to \mathbb{N})$) et un ensemble de positions (de type $\mathbb{P}(Locations)$) et renvoie comme résultat un entier naturel représentant la somme des quantités de nourriture dans les positions considérées. Le reste des axiomes donnent une définition par récurrence de la fonction $TotalFood$. Ainsi :

- $TotalFood$ renvoie la valeur zéro dans le cas où la quantité de nourriture dans l'ensemble des positions considérées est nulle ($axm2$),
- pour une position donnée loc, $TotalFood$ renvoie la quantité de nourriture contenue dans cette position ($axm3$) et
- l'axiome $axm4$ définit la fonction $TotalFood$ pour plusieurs positions.

Dans cette deuxième phase du processus RefProSOMAS, la machine $AntsMicroMeso1$ (contenant une spécification correcte du comportement des fourmis) est raffinée en introduisant l'événement observateur $AllFoodAtNest$. Cette étape de raffinement est suivie par de nouveaux raffinements dont l'objectif est de prouver la terminaison des différents événements modélisant les actions des fourmis. Ils sont détaillés dans la section suivante.

6.3.2 La preuve de la terminaison des événements d'action

Comme nous l'avons mentionné au chapitre 3, la preuve de la terminaison des événements nécessite la définition d'une expression entière ou un ensemble fini, appelé variant. Il est possible de définir un seul variant par machine, ainsi cette preuve de terminaison nécessite plusieurs étapes de raffinement au cours de chacune d'entre elles, un seul événement d'action sera considéré. Le tableau 6.7, décrit pour chaque événement le variant défini pour montrer sa terminaison. Une fois les variants définis, l'étape suivante consiste à prouver que chacun de ces événements, en s'exécutant, décrémente le variant qui lui correspond. Dans certains cas, cette preuve est triviale (le cas des événements $PerformAntsDropFood$, $PerformAntsHarvestFood$ et $PerformAntsDropPheromone$) et nécessite une légère modification des gardes et des actions des événements responsables de la modification du variant. Dans d'autres cas, la preuve nécessite l'ajout de nouveaux axiomes (le cas des événements $PerformAntsMoveBack$, $PerformAntsMoveExploreFollowFood$ et $PerformAntsMoveExploreFollowPheromone$). Pour l'événement $PerformAntsMoveExploreRandom$, des hypothèses d'équité forte ont été nécessaires pour montrer qu'il décrémente son variant. La preuve relative à ce dernier événement est faite à l'aide de la logique TLA.
Dans les paragraphes suivants nous détaillons le preuve de terminaison d'un événement de chaque groupe et nous nous intéressons particulièrement aux événements $PerformAntsDropFood$, $PerformAntsMoveBack$ et $PerformAntsMoveExploreRandom$.

Evénement	Variant
PerformAntsDropFood	V1 : l'ensemble des fourmis en train de déposer de la nourriture dans le nid
PerformAntsHarvestFood	V2 : la somme totale de nourriture dans l'environnement à l'exception du nid
PerformAntsDropPheromone	V3 : l'ensemble des fourmis en train de déposer de la phéromone
PerformAntsMoveBack	V4 : la somme des distances entre les positions des fourmis retournant au nid et le nid
PerformAntsMoveExploreFollowFood	V5 : la somme des distances entre les positions des fourmis en quête d'une source de nourriture et la position de la source de nourriture en question
PerformAntsMoveExploreFollowPheromone	V6 : la somme des distances entre les positions des fourmis poursuivant de la phéromone et la position de la phéromone en question
PerformAntsMoveExploreRandom	V7 : l'ensemble des fourmis en train de faire un déplacement aléatoire

Tableau 6.7 — Définition des variants nécessaires pour montrer la terminaison des événements d'action

6.3.2.1 La preuve de la terminaison de l'événement $PerformAntsDropFood$

Pour prouver que l'événement $PerformAntsDropFood$ est convergent, nous définissons l'ensemble $AntsDroppingFoodAtNest$ comme étant l'ensemble des fourmis en train de déposer de la nourriture dans le nid. Cet ensemble joue également le rôle du variant $V1$. Les événements $DecideAntsDropFood$ et $PerformAntsDropFood$ sont les événements responsables de modifier ce variant. Ils sont alors raffinés comme le montre les deux paragraphes suivants.

Raffinement de l'événement $DecideAntsDropFood$

Ce raffinement modélise la mise à jour de l'ensemble $AntsDroppingFoodAtNest$ en y rajoutant les fourmis qui se trouvent chargées dans le nid et qui décident donc de déposer la nourriture qu'elle transportent. Ce raffinement consiste à ajouter l'action $addAntDrpping$ (définie ci-dessous) parmi les actions de l'événement $DecideAntsDropFood$.

$$addAntDrpping : AntsDroppingFoodAtNest := AntsDroppingFoodAtNest \cup \{agent\}$$

$agent$ étant un paramètre de l'événement $DecideAntsDropFood$ représentant une fourmi.

Raffinement de l'événement $PerformAntsDropFood$

Pour montrer que l'événement $PerformAntsDropFood$ décrémente à chaque exé-

cution l'ensemble *AntsDroppingFoodAtNest*, il est nécessaire de rajouter l'action *removeAntsDropFood* (définie ci-dessous).

$$removeAntsDropFood : AntsDroppingFoodAtNest := AntsDroppingFoodAtNest \setminus \{agent\}$$

agent étant un paramètre de l'événement *DecideAntsDropFood* représentant une fourmi.

6.3.2.2 La preuve de la terminaison de l'événement $PerformAntsMoveBack$

Afin de prouver la convergence de l'événement *PerformAntsMoveBack*, nous définissons le variant $V4$ formalisé par l'expression suivante.

$$SumDistances(\{a \cdot a \in AntsApproachingNest | a \mapsto Dist(currentLoc(a) \mapsto Nest)\})$$

Informellement, le variant $V4$ est la somme des distances entre le nid et les positions de toutes les fourmis qui rentrent au nid (les fourmis de l'ensemble *AntsApproachingNest*). Dans cette expression, la fonction *SumDistances* renvoie la somme de distances et *Dist* est une fonction mesurant la distance entre deux positions dans l'environnement.

6.3.2.3 La preuve de la terminaison de l'événement $PerformAntsMoveExploreRandom$

Afin de prouver la terminaison de l'événement *PerformAntsMoveExploreRandom*, nous ajoutons la variable *AntsMovingRandom* représentant l'ensemble des fourmis en train de faire un mouvement aléatoire. Aussi, nous raffinons l'événement *PerformAntsMoveExploreRandom* en les deux événements $PerformAntsMoveExploreRandomRef$ et $PerformAntsMoveExploreRandomConv$. L'événement $PerformAntsMoveExploreRandomRef$ modélise un mouvement aléatoire qui garde la fourmi concernée dans l'ensemble *AntsMovingRandom* (le prochain mouvement de cette fourmi sera aussi aléatoire). L'événement $PerformAntsMoveExploreRandomConv$ décrit un mouvement aléatoire ayant permis à la fourmi qui l'a réalisé de quitter l'ensemble *AntsMovingRandom* et ainsi de poursuivre de la nourriture ou de la phéromone ou d'éviter une concurrence lors de sa prochaine action. Montrer la convergence des deux événements $PerformAntsMoveExploreRandomRef$ et $PerformAntsMoveExploreRandomConv$ revient à montrer la formule $P \rightsquigarrow Q$ dans laquelle P dénote la cardinalité courante de l'ensemble *AntsMovingRandom* et Q dénote une décrémentation de la cardinalité de cet ensemble. En utilisant la règle de preuve $SF1$ et $LATTICE$ de la logique TLA, il est possible de montrer cette formule. En considérant :
$N \widehat{=} PerformAntsMoveExploreRandomRef \vee PerformAntsMoveExploreRandomConv$
$A_{MovRandConv} \widehat{=} PerformAntsMoveExploreRandomConv$
$P \widehat{=} card(AntsMovingRandom) = n + 1$, $Q \widehat{=} card(AntsMovingRandom) = n$ avec $card(AntsMovingRandom)$ dénote la cardinalité de l'ensemble *AntsMovingRandom*,
la règle $SF1$ s'écrit :

$SF1.1 \quad P \wedge [N]_{AntsMovingRandom} \Rightarrow (P' \vee Q')$
$SF1.2 \quad P \wedge \langle N \wedge A_{MovRandConv} \rangle_{AntsMovingRandom} \Rightarrow Q'$
$SF1.3 \quad \Box P \wedge \Box[N]_{AntsMovingRandom} \Rightarrow \Diamond Enabled\langle A_{MovRandConv} \rangle_{AntsMovingRandom}$

$SF1.A \quad \Box[N]_{AntsMovingRandom} \wedge SF_{AntsMovingRandom}(A_{MovRandConv}) \Rightarrow P \rightsquigarrow Q$

La formule $SF1.1$ spécifie que suite à l'exécution d'un événement parmi les événements $PerformAntsMoveExploreRandomRef$ et $PerformAntsMoveExploreRandomConv$ à partir de l'état P, le système peut soit passer à l'état $P' = card(AntsMovingRandom) = n + 1$ (la cardinalité de l'ensemble $AntsMovingRandom$ ne change pas) soit passer à l'état $Q' = card(AntsMovingRandom) = n$ (la cardinalité de l'ensemble $AntsMovingRandom$ a été décrémentée). Dans la formule $SF1.2$, l'action $A_{MovRandConv}$ permet d'atteindre l'état $Q' = card(AntsMovingRandom) = n$. La formule $SF1.3$ indique que l'action $A_{MovRandConv}$ finira par être activable.
L'hypothèse d'équité forte sur l'exécution de l'action $A_{MovRandConv}$ formulée par $SF_{AntsMovingRandom}(A_{MovRandConv})$ permet de prouver la propriété $P \rightsquigarrow Q$. En appliquant LATTICE, il est possible de prouver que la cardinalité de l'ensemble $AntsMovingRandom$ converge vers 0.

6.4 Phase N°3 : preuve de la résilience

Les deux phases du processus RefProSOMAS ont donné lieu à la formalisation du comportement des fourmis. Les modèles obtenus vérifient les propriétés locales de non blocage ainsi que les contraintes locales issues de la description de l'étude de cas. Ces modèles permettent également la convergence vers un état dans lequel toute la nourriture est récoltée. Cette convergence a été prouvée en supposant que l'environnement reste inchangé. Ainsi, les événements relatifs à l'apparition de nouvelles sources de nourriture ou l'apparition de nouveaux obstacles n'ont pas été pris en compte.

Dans cette section, nous nous intéressons à la réaction des fourmis lorsque de nouvelles sources de nourriture sont ajoutées dans l'environnement. Nous voulons prouver que les fourmis sont capables de la détecter. Nous exprimons cette propriété à l'aide de la logique temporelle par la formule 6.1.

$$ResNewFood \mathrel{\widehat{=}} \Box(\forall loc.(loc \in NewFoodLocations \wedge \exists ant.comeBackAnt(ant) = FALSE \wedge nextLocation(ant) = loc \Rightarrow \Diamond(loc \in DetectedFoodLocations)) \quad (6.1)$$

Dans la formule 6.1, la variable $NewFoodLocations$ désigne l'ensemble des positions dans lesquelles la nourriture est rajoutée. La variable $DetectedFoodLocations$ désigne l'ensemble des sources de nourriture détectée.
Informellement, cette formule signifie que toute source de nourriture (loc) ajoutée dans l'environnement ($loc \in NewFoodLocations$) finira par être détectée. La condition

$\exists ant.comeBackAnt(ant) = FALSE \wedge nextLocation(ant) = loc$) spécifie qu'il existe une fourmi (*ant*) en mode exploration ($comeBackAnt(ant) = FALSE$) qui a détecté une source de nourriture en faisant un déplacement dans sa direction.

Selon le patron *P_RES*, il faut modéliser la perturbation en raffinant l'événement *EnvironmentChange* par l'événement *EnvironmentChangeAddFood* pour modéliser l'ajout de nourriture dans l'environnement. L'événement concret est donné par la figure 6.27. L'action *act*3 permet de modifier la quantité de nourriture dans la position concernée par

EVENT EnvironmentChangeAddFood
REFINES EnvironmentChange
ANY
  $newFood$
WHERE
  $grd1 : \forall agent \cdot agent \in Ants \Rightarrow antMode(agent) = wait$
  $grd2 : newFood \subseteq (\{loc \cdot loc \in Locations \wedge QuantityFood(loc) = 0 | loc\} \setminus \{Nest\})$
THEN
  $act1 : antMode : |antMode' = Ants \times \{work\}$
  $act2 : NewFoodLocations := NewFoodLocations \cup newFood$
  $act3 : QuantityFood : |QuantityFood'(newFood) \in 1..QuantityFoodMax \wedge$
    $\forall loc \in Locations \setminus \{newFood\} \Rightarrow QuantityFood'(loc) = QuantityFood(loc)$
END

Figure 6.27 — L'événement *EnvironmentChangeAddFood*

l'ajout en conservant la quantité de nourriture des autres positions.

Le patron *P_RES* recommande aussi la modélisation :

- ▷ des événements qui correspondent aux actions du système lui permettant de "corriger" la perturbation (notés $selfOrg$ dans le patron *P_RES*). Dans notre cas, ces événements sont : $PerformAntsMoveExploreFollowFood$ $PerformAntsMoveExploreFollowPheromone$ et $PerformAntsMoveExploreFollowRandom$. Une réflexion sur la manière avec laquelle les fourmis peuvent découvrir de nouvelles sources de nourriture, nous emmène à doter les fourmis d'un comportement coopératif leur dictant d'éviter d'aller dans les directions qui contiennent beaucoup de fourmis même si elles contiennent de la nourriture. Ainsi, nous enrichissons l'ensemble des événements, modélisant les actions d'exploration de l'environnement, par l'événement $PerformAntsMoveExploreAvoidCompetition$. Nous jugeons que le rôle de cet événement est primordial pour la résilience du système.
- ▷ les événements qui correspondent aux actions permettent au système de progresser vers l'état désiré ((notés $selfOrgConv$ dans le patron *P_RES*)). Dans notre cas, un seul événement fait progresser le système vers un état où toute la nourriture ajoutée est détectée. Il s'agit de l'événement $PerformAntsMoveExploreFollowFoodFirstTime$ qui modélise la détection d'une source de nourriture. L'événement $PerformAntsMoveExploreFollowFoodFirstTime$ raffine l'événement $PerformAntsMoveExploreFollowFood$ en ajoutant la condition $nextLocation(ant) \in NewFoodLocations$ dans sa garde et en ajoutant les deux actions

décrites ci-dessous parmi l'ensemble de ses actions.

$actionDetectedUpdate : DetectedFoodLocations := DetectedFoodLocations \cup \{nextLocation(ant)\}$
$actionNewFooddUpdate : NewFoodLocations := NewFoodLocations \setminus \{nextLocation(ant)\}$

Figure 6.28 — Les actions ajoutées pour l'événement $PerformAntsMoveExploreFollowFoodFirstTime$

- l'événement *ResilienceObserver* qui est un événement sans action dont la garde décrit l'état ou toutes la nourriture ajoutée est détectée. Cette garde est donne par l'expression $NewFoodLocations = \varnothing$.

Les éléments du modèle étant définis, nous passons à la preuve de la propriété *ResNewFood*. Nous considérons les deux prédicats P et $Q_{Detected}$ définis en fonction de la cardinalité de l'ensemble $NewFoodLocations$ $P \widehat{=} card(NewFoodLocations) = n + 1$,
$Q_{Detected} \widehat{=} card(NewFoodLocations) = n$
et nous souhaitons prouver la formule $P \rightsquigarrow Q_{Detected}$.
On définit N et $A_{Detected}$ comme suit :
$N \widehat{=} PerformAntsMoveExploreFollowFoodFirstTime \vee$
$PerformAntsMoveExploreFollowFood \vee$
$PerformAntsMoveExploreAvoidCompetition \vee$
$PerformAntsMoveExploreFollowPheromone \vee$
$PerformAntsMoveExploreRandom \vee$
$EnvironmentChangeAddFood \vee$
$ResilienceOberver$
et
$A_{Detected} \widehat{=} PerformAntsMoveExploreFollowFoodFirstTime$.
En appliquant $SF1$, il est possible de prouver $P \rightsquigarrow Q_{Detected}$:

$SF1.1$ $\quad P \wedge [N]_{NewFoodLocations} \Rightarrow (P' \vee Q'_{Detected})$
$SF1.2$ $\quad P \wedge \langle N \wedge A_{Detected} \rangle_{NewFoodLocations} \Rightarrow Q'_{Detected})$
$SF1.3$ $\quad \Box P \wedge \Box[N]_{NewFoodLocations} \Rightarrow \Diamond Enabled \langle A_{Detected} \rangle_{NewFoodLocations}$

$SF1.H$ $\quad \Box[N]_{NewFoodLocations} \wedge SF_{NewFoodLocations}(A_{Detected}) \Rightarrow P \rightsquigarrow Q_{Detected}$

La condition $SF1.1$ décrit une étape durant laquelle le système progresse vers un état vérifiant le prédicat P ou vers un état vérifiant l'état $Q_{Detected}$. La condition $SF1.2$ décrit une étape d'induction durant laquelle l'action $\langle A_{Detected} \rangle_{NewFoodLocations}$ (la détection d'une position contenant de la nourriture ajoutée) donne lieu à un état vérifiant $Q_{Detected}$. La condition $SF1.3$ assure que $\langle A_{Detected} \rangle_{NewFoodLocations}$ finira par être activée.

En supposant que les opérations d'ajout de nourriture dans l'environnement finiront par être arrêté et en appliquant la règle $LATTICE$, on peut prouver que la cardinalité de l'ensemble *NewFoodLocations* finira par s'annuler ce qui activera l'événement *ResilienceObserver*.

6.5 Conclusion

L'objectif de ce chapitre est d'illustrer le processus RefProSOMAS et voir jusqu'à quelle mesure les patrons sur lesquels il se base peuvent apporter de l'aide au concepteur. Depuis cette application, nous avons tiré les conclusions suivantes :

- l'utilisation du patron *P_MAS* pour la modélisation des règles comportementales des fourmis peut apporter une aide considérable au concepteur. Cependant, le plugin que nous avons utilisé ne permet pas le raffinement d'un événement en plusieurs événement en une seule itération. Ainsi, nous avons été obligés de faire ce raffinement en plusieurs étapes et utiliser la fusion ensuite pour regrouper les différents événements obtenus dans une seule machine.
- l'utilisation des patrons *P_CONV* et *P_RES* apporte une assistance pour la modélisation mais pas pour la preuve. La preuve reste toujours à la charge du concepteur.
- l'utilisation de la logique *TLA* conjointement avec les modèles en *B-événementiel* a nettement simplifié la preuve et nous la considérons comme une piste prometteuse pour raisonner formellement sur des systèmes auto-organisateurs.

Conclusion et perspectives

« Chercher n'est pas une chose et trouver une autre, mais le gain de la recherche, c'est la recherche même. »

Saint Grégoire de Nysse – Homélies sur l'Ecclésiaste

TOUT au long de ce travail, notre objectif a été de proposer des modèles formels et une stratégie pour la vérification des systèmes multi-agents auto-adaptatifs. Situé à la frontière des méthodes formelles et des systèmes auto-adaptatifs, ce travail a nécessité de faire des choix élémentaires afin de relier ces deux domaines différents.

Le choix du langage B-événementiel. Trois raisons essentielles ont motivé le choix du langage B-événementiel :

- B-évènementiel est un langage bien supporté par des outils (en l'occurrence la plateforme RODIN) permettant la modélisation et la vérification de systèmes distribués,
- il s'agit d'un langage assez expressif, grâce à la théorie des ensembles et la logique de premier ordre, pour modéliser les aspects relatifs aux SMA auto-organisateurs, et
- il repose sur le principe de raffinement qui consiste en le développement de systèmes de manière itérative ce qui permet de réduire fortement la difficulté lors de la conception et la preuve des SMA auto-organisateurs.

Le choix de la logique TLA. Dans ce travail, les règles de preuve ($WF1$ et $SF1$) de la logique TLA ont été utilisées pour prouver les propriétés de résilience et de convergence exprimées à l'aide des opérateurs temporels. Comparée aux travaux de T.S. Hoang et J.-R. Abrial ([Hoang et Abrial, 2011]) portant sur la preuve des propriétés de vivacité à l'aide des concepts de B-événementiel, la logique TLA offre un moyen plus explicite pour exprimer l'ordonnancement de l'exécution des événements et par conséquent elle fournit un cadre plus intuitif pour le raisonnement sur les propriétés de vivacité.

Le choix d'une approche ascendante. Après une exploration de l'approche descendante, nous avons trouvé que l'approche ascendante convient mieux à la modélisation et la vérification des SMA auto-organisateurs. En effet, cette approche nous permet de suivre le

même principe utilisé pour simuler ces systèmes (le moyen le plus naturel pour les vérifier) en modélisant d'abord les comportements locaux des agents et en les observant ensuite s'exécuter pour "évaluer" la fonctionnalité émergente.

Au terme de ce travail, nous avons pu atteindre une bonne partie des objectifs fixés au départ.

La formalisation des SMA auto-organisateurs. Nous avons réussi à décrire les SMA auto-organisateurs à l'aide du langage B-événementiel. Cette formalisation est faite selon trois niveaux d'abstraction et a permis de spécifier formellement les propriétés pertinentes tant au niveau local qu'au niveau global. Ainsi, au niveau micro, nous nous sommes intéressés à décrire les règles comportementales des agents, à spécifier les contraintes sur ces comportements locaux et à modéliser les propriétés locales de non blocage. Au niveau macro, la vérification formelle de l'activité globale du système porte sur la vérification de sa convergence et sa résilience. Ces deux propriétés ont été exprimées à l'aide de la logique temporelle.
Une partie de ces contributions a fait l'objet de la publication [Graja *et al.*, 2014a] élu meilleur papier lors des journées francophones sur les SMA (JF'SMA 2014).

La stratégie de vérification. Nous avons constaté que la majorité des travaux portant sur la vérification formelle des systèmes auto-organisateurs manque de guides méthodologiques pour appliquer les formalismes proposés. De ce fait, nous avons voulu assister les concepteurs de SMA auto-organisateur, surtout les non spécialistes des méthodes formelles, lors du processus de vérification qui peut représenter aussi un processus de modélisation.
Nos réflexions dans ce but ont donné lieu au processus RefProSOMAS. Ce processus a la particularité d'être ascendant et donc bien adapté aux SMA auto-organisateurs. Composé de trois phases de raffinement, RefProSOMAS débute par la spécification des comportements locaux des agents (niveau micro) qui satisfont les contraintes locales et tout en prouvant les non blocages, propose ensuite de prouver les propriétés de convergence et se termine par la preuve des propriétés de résilience.
Chaque phase est un raffinement de la phase qui la précède et vient enrichir les règles comportementales des agents. A la fin du processus, nous obtenons une spécification des règles comportementales des agents correcte (du point de vue des contraintes imposées et des non blocage) et qui assurent la convergence et la résilience du système au niveau macro. Ces spécifications peuvent être alors traduites pour être implémentées.

Les patrons de raffinements.
Toujours dans le but d'apporter de l'assistance aux concepteurs de SMA auto-organisateurs, nous avons proposé d'associer à chacune des étapes du processus RefProSOMAS un patron de raffinement. Un patron de raffinement décrit de manière générique le modèle de chaque étape sous forme de variables, d'événements ainsi que les raffinements à effectuer au sein de cette étape.

Nous associons le patron P_MAS à la première étape. Ce patron est formé par :

- ▷ des variables locales modifiées et/ou utilisées dans les règles comportementales des agents,
- ▷ des variables décrivant l'environnement où évoluent les agents,
- ▷ de trois étapes de raffinement qui décrivent d'abord les règles de décision des agents, ensuite leurs actions et enfin les opérations qui leurs permettent de mette à jour leurs représentations partielles de l'environnement.

Nous associons le patron *P_CONV* à la deuxième étape. Ce patron est formé par des variables globales décrivant l'état du système au niveau macro. Les événements dans ce patron sont des raffinements des actions des agents et décrivent l'évolution de ces variables globales suite aux actions des agents. Ce qui nous permet de faire le lien entre le niveau micro et le niveau macro. Ce patron décrit aussi les théorèmes nécessaires pour prouver la convergence du systèmes vers les objectifs attendus par un observateur externe au système.

Nous associons le patron *P_RES* à la troisième étape. L'objectif de ce patron est de prouver des propriétés de résilience, il est formé de variables qui décrivent l'état de l'environnement, d'événements qui spécifient les perturbations en provenance de l'environnement ainsi que les théorèmes nécessaires à la preuve de la résilience.

Cette contribution a fait l'objet d'une publication [Graja *et al.*, 2014b] dans ICAART'2014.

Nos perspectives concernent notamment le processus de modélisation et de vérification RefProSOMAS et elles sont résumées en les quatre points suivants :

Automatisation du processus de développement et de preuve. L'application de RefProSOMAS sur l'étude de cas des fourmis fourrageuses a révélé la possibilité d'automatiser plusieurs étapes dans ce processus. En l'occurrence, il est possible de dériver toutes les actions des agents et créer automatiquement tous les événements qui leur correspondent une fois les règles décisionnelles définies. Cette automatisation peut être effectuée grâce à un plug-in intégré à la plateforme *Rodin*
L'instanciation du patron *P_MAS*, réalisée par le plug-in *Pattern* pour définir les différentes règles de décision ainsi que les actions des agents, a nécessité l'application de ce patron plusieurs fois. Nous considérons qu'une amélioration de ce plug-in permettant la décomposition d'un événement en plusieurs événements pourra être très utile et réduira les efforts d'instanciation.

Extension du processus RefProSOMAS pour permettre une vérification quantitative. La preuve à l'aide de B-événementiel et la logique TLA permet une évaluation qualitative du système et n'est pas donc convenable pour vérifier des propriétés d'optimisation ou pour faire une étude comparative. L'investigation dans la technique du model-checking probabiliste est une voie prometteuse pour de telles situations. Nous jugeons donc que la prise en compte des aspects quantitatifs dans le processus RefProSOMAS est d'une grande importance. Nous pouvons pour cela nous baser sur les travaux de A. Tarasyuk

et *al* [Tarasyuk *et al.*, 2015] portant sur l'intégration du raisonnement stochastique dans le développement à l'aide de B-événementiel.

Intégration du processus RefProSOMAS avec la méthode ADELFE. La méthode ADELFE a été proposée au sein de l'équipe SMAC pour développer des SMA auto-organisateurs coopératifs. Cette méthode repose sur un langage spécifique et qui étend le langage UML appelé AMAS-ML. L'intégration de ADELFE avec RefProSOMAS aura pour objectif de rendre la modélisation formelle la plus transparente possible au concepteur tout en bénéficiant de l'avantage de la preuve formelle. Au moyen des techniques de transformation de modèles issues de l'IDM (Ingénierie Dirigée par les modèles), il est possible de traduire les modèles décrits en AMAS-ML en modèles formels décrits en B-événementiel qui serviront de base pour assurer la preuve.

Expérimentation du processus de vérification RefProSOMAS sur d'autres études de cas. Le processus de modélisation et de vérification que nous avons proposé a été appliqué à l'étude de cas des fourmis fourrageuses. Nous souhaitons l'expérimenter sur de nouveaux problèmes auxquels l'équipe s'intéresse notamment dans le domaine des systèmes ambiants et l'apprentissage.

Bibliographie personnelle

Conférences et workshops internationaux

- Zeineb GRAJA, Frédéric MIGEON, Christine MAUREL, Marie-Pierre GLEIZES, Ahmed HADJ KACEM. A Stepwise Refinement Based Development of Self-Organizing Multi-Agent Systems : Application to the Foraging Ants, Second International Workshop, EMAS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers, Lecture Notes in Computer Science, 2014, Vol. 8758, pp 40-57.
- Linas LAIBINIS, Elena TROUBITSYNA, Zeineb GRAJA, Frédéric MIGEON et Ahmed HADJ KACEM. Formal modelling and verification of cooperative ant behaviour in Event-B. Dans Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings, pages 363 ?377, 2014.
- Zeineb GRAJA, Frédéric MIGEON, Christine MAUREL, Marie-Pierre GLEIZES, Linas LAIBINIS, Amira REGAYEG, Ahmed HADJ KACEM. A Pattern based Modelling for Self-Organizing Multi-Agent Systems with Event-B (short paper). ICAART 2014 - Proceedings of the 6th International Conference on Agents and Artificial Intelligence, Volume 2, ESEO, Angers, Loire Valley, France, 6-8 March, 2014. SciTePress 2014 ISBN 978-989-758-016-1, pp 229-236.

Workshops nationaux

- Zeineb GRAJA, Frédéric MIGEON, Christine MAUREL, Marie Pierre GLEIZES, Ahmed HADJ KACEM. Vers une modélisation formelle basée sur le raffinement des systèmes multi-agents auto-organisateurs. JFSMA 14 - Vingt-deuxièmes Journées Francophones sur les Systèmes Multi-Agents, Loriol-sur-Drôme, France, Octobre 8-10, 2014. Cepadues Editions 2014 ISBN 978-2-36493-154-1, pp 139-148. (prix du meilleur papier)

Bibliographie

Jean-Raymond ABRIAL : *The B-book - assigning programs to meanings*. Cambridge University Press, 2005. ISBN 978-0-521-02175-3.

Jean-Raymond ABRIAL : *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. ISBN 978-0-521-89556-9. URL http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521895569.

Rasmus ADLER, Ina SCHAEFER, Tobias SCHUELE et Eric VECCHIÉ : From model-based design to formal verification of adaptive embedded systems. *In In Proc. of ICFEM 2007*. Springer, 2007.

Jevtić ALEKSANDAR et Andina DIEGO : Swarm intelligence and its applications in swarm robotics. *In Proceedings of the 6th WSEAS International Conference on Computational Intelligence, Man-machine Systems and Cybernetics*, CIMMACS'07, pages 41–46. World Scientific and Engineering Academy and Society WSEAS, 2007. ISBN 978-960-6766-25-1. URL http://dl.acm.org/citation.cfm?id=1984502.1984510.

Adnan AZIZ, Kumud SANWAL, Vigyan SINGHAL et Robert K. BRAYTON : Model-checking continous-time markov chains. *ACM Trans. Comput. Log.*, 1(1):162–170, 2000.

R. J. R. BACK et R. KURKI-SUONIO : Decentralization of process nets with centralized control. *In Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 131–142, New York, NY, USA, 1983. ACM. ISBN 0-89791-110-5. URL http://doi.acm.org/10.1145/800221.806716.

Ralph-Johan BACK : *On the Correctness of Refinement Steps in Program Development*. Thèse de doctorat, 1978.

Ralph-Johan BACK : *Correctness Preserving Program Refinements : Proof Theory and Applications*, volume 131 de *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.

Ralph-Johan BACK et Kaisa SERE : Stepwise refinement of action systems. *Structured Programming*, 12(1):17–30, 1991.

Ralph-Johan BACK et Joakim von WRIGHT : *Refinement Calculus - A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998. ISBN 978-0-387-98417-9. URL http://dx.doi.org/10.1007/978-1-4612-1674-2.

Michael BALSER, Wolfgang REIF, Gerhard SCHELLHORN et Kurt STENZEL : Kiv 3.0 for provably correct systems. *In FM-Trends*, pages 330–337, 1998.

Steven Carl BANKES : Robustness, adaptivity, and resiliency analysis. *In AAAI Fall Symposium : Complex Adaptive Systems*, volume FS-10-03 de *AAAI Technical Report*. AAAI, 2010.

Simon BÄUMLER, Gerhard SCHELLHORN, Bogdan TOFAN et Wolfgang REIF : Proving linearizability with temporal logic. *Formal Asp. Comput.*, 23(1):91–112, 2011.

Carole BERNON, Marie-Pierre GLEIZES et Gauthier PICARD : Enhancing self-organising emergent systems design with simulation. *In Engineering Societies in the Agents World VII, 7th International Workshop, ESAW 2006, Dublin, Ireland, September 6-8, 2006 Revised Selected and Invited Papers*, pages 284–299, 2006. URL http://dx.doi.org/10.1007/978-3-540-75524-1_16.

Eric BONABEAU, Marco DORIGO et Guy THERAULAZ : *Swarm Intelligence : From Natural to Artificial Systems*. Oxford University Press, Inc., New York, NY, USA, 1999. ISBN 0-19-513159-2.

Noélie BONJEAN, Carole BERNON et Pierre GLIZE : Towards a guide for engineering the collective behaviour of a MAS. *Simulation Modelling Practice and Theory*, 18(10):1506–1514, 2010. URL http://dx.doi.org/10.1016/j.simpat.2010.05.001.

Valérie CAMPS : *Vers une théorie de l'auto-organisation dans les systèmes multi-agents basée sur la coopération : application à la recherche d'information dans un système d'information répartie*. Thèse de doctorat, Université Paul Sabatier, Toulouse, France, janvier 1998. URL http://www.irit.fr/~Valerie.Camps/Rech/Fichiers/theseVCamps.pdf.

Davy CAPERA, Jean-Pierre GEORGÉ, Marie-Pierre GLEIZES et Pierre GLIZE : The AMAS Theory for Complex Problem Solving Based on Self-organizing Cooperative Agents. *In International Workshop on Theory And Practice of Open Computational Systems (TAPOCS at IEEE WETICE 2003 (TAPOCS - WETICE), Linz, Austria, 09/06/2003-11/06/2003*, pages 389–394, http ://www.computer.org, juin 2003. IEEE Computer Society.

Matteo CASADEI et Mirko VIROLI : Using probabilistic model checking and simulation for designing self-organizing systems. *In Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 2103–2104, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8. URL http://doi.acm.org/10.1145/1529282.1529747.

Federica CIOCCHETTA et Jane HILLSTON : B.

Massimo COSSENTINO, Giancarlo FORTINO, Alfredo GARRO, Samuele MASCILLARO et Wilma RUSSO : PASSIM : a simulation-based process for the development of multi-agent systems. *IJAOSE*, 2(2):132–170, 2008. URL http://dx.doi.org/10.1504/IJAOSE.2008.017313.

Massimo COSSENTINO, Giancarlo FORTINO, Marie-Pierre GLEIZES et Juan PAVÓN : Simulation-based design and evaluation of multi-agent systems. *Simulation Modelling Practice and Theory*, 18(10):1425–1427, 2010.

Tom DE WOLF et Tom HOLVOET : A Taxonomy for Self-Properties in Decentralised Autonomic Computing. CRC Press, Taylor and Francis Group, 2007. URL https://lirias.kuleuven.be/handle/123456789/147438.

Tom DE WOLF, Tom HOLVOET et Giovanni SAMAEY : Development of self-organising emergent applications with simulation-based numerical analysis. *Engineering Self-organising Systems*, 3910:138–152, 2005. URL https://lirias.kuleuven.be/handle/123456789/125112.

Giovanna DI MARZO SERUGENDO, Marie-Pierre GLEIZES et Anthony KARAGEORGOS, éditeurs. *Self-organising Software - From Natural to Artificial Adaptation*. Natural Computing Series. Springer, http ://www.springerlink.com, octobre 2011. ISBN 978-3-642-17348-6. URL http://www.springer.com/computer/ai/book/978-3-642-17347-9.

Edsger W. DIJKSTRA : Notes on Structured Programming. URL http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF. circulated privately, avril 1970.

Edsger W. DIJKSTRA : Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, novembre 1974. ISSN 0001-0782. URL http://doi.acm.org/10.1145/361179.361202.

Hartmut EHRIG, Karsten EHRIG, Ulrike PRANGE et Gabriele TAENTZER : *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. ISBN 978-3-540-31187-4, 978-3-540-31188-1.

Hartmut EHRIG, Claudia ERMEL, Olga RUNGE, Antonio BUCCHIARONE et Patrizio PELLICCIONE : Formal analysis and verification of self-healing systems. *In FASE*, pages 139–153, 2010.

Radek ERBAN, Jonathan CHAPMAN et Philip MAINI : A practical guide to stochastic simulations of reaction-diffusion processes, 2007. URL http://arxiv.org/abs/0704.1908.

Jacques FERBER : *Multi-agent systems - an introduction to distributed artificial intelligence*. Addison-Wesley-Longman, 1999.

Luca GARDELLI, Mirko VIROLI et Andrea OMICINI : On the role of simulations in engineering self-organising MAS : The case of an intrusion detection system. *In Engineering Self-Organising Systems*, pages 153–166, 2005.

Luca GARDELLI, Mirko VIROLI et Andrea OMICINI : Exploring the dynamics of self-organising systems with stochastic π-calculus : Detecting abnormal behaviour in MAS. *In IN MAS. IN : FIFTH INTERNATIONAL SYMPOSIUM FROM AGENT THEORY TO AGENT IMPLEMENTATION (AT2AI5*, 2006.

Alfredo GARRO et Wilma RUSSO : easyABMS : A domain-expert oriented methodology for agent-based modeling and simulation. *Simulation Modelling Practice and Theory*, 18(10):1453–1467, 2010. URL http://dx.doi.org/10.1016/j.simpat.2010.04.004.

Jean-Pierre GEORGÉ : *Résolution de problèmes par émergence, Etude d'un Environnement de Programmation émergente*. Thèse de doctorat, Université Paul Sabatier, Toulouse, France, juillet 2004. URL ftp://ftp.irit.fr/IRIT/SMAC/DOCUMENTS/RAPPORTS/TheseJPGeorge_0704.pdf.

Jean-Pierre GEORGÉ, Marie-Pierre GLEIZES et Pierre GLIZE : Conception de systèmes adaptatifs à fonctionnalité émergente : la théorie des AMAS. *Revue d'Intelligence Artificielle*, 17 (4/2003):591–626, 2003.

D. T. GILLESPIE : Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

Fausto GIUNCHIGLIA, James ODELL et Gerhard WEISS, éditeurs. *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002, Revised Papers and Invited Contributions*, volume 2585 de *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-00713-X.

Pierre GLIZE : *L'Adaptation des Systemes à Fonctionnalité Emergente par Auto-Organisation Cooperative*. Habilitation à diriger des recherches, Université Paul Sabatier, Toulouse, France, juin 2001. URL ftp://ftp.irit.fr/IRIT/SMAC/DOCUMENTS/RAPPORTS/HdRPGlize_0601.pdf.

Jorge J. GÓMEZ-SANZ, Carlos R. FERNÁNDEZ et Javier ARROYO : Model driven development and simulations with the INGENIAS agent framework. *Simulation Modelling Practice and Theory*, 18(10):1468–1482, 2010. URL http://dx.doi.org/10.1016/j.simpat.2010.05.012.

Zeineb GRAJA, Frédéric MIGEON, Christine MAUREL, Marie-Pierre GLEIZES et Ahmed HADJ KACEM : Vers une modélisation formelle basée sur le raffinement des systèmes multi-agents auto-organisateurs. *In Principe de Parcimonie - JFSMA 14 - Vingt-deuxièmes Journées Francophones sur les Systèmes Multi-Agents, Loriol-sur-Drôme, France, Octobre 8-10, 2014*, pages 139–148, 2014a.

Zeineb GRAJA, Frédéric MIGEON, Christine MAUREL, Marie-Pierre GLEIZES, Linas LAIBINIS, Amira REGAYEG et Ahmed HADJ KACEM : A pattern based modelling for self-organizing multi-agent systems with Event-B. *In ICAART 2014 - Proceedings of the 6th International Conference on Agents and Artificial Intelligence, Volume 2, ESEO, Angers, Loire Valley, France, 6-8 March, 2014*, pages 229–236, 2014b. URL http://dx.doi.org/10.5220/0004906902290236.

Pierre-P. GRASSÉ : La reconstruction du nid et les coordinations interindividuelles chezbellicositermes natalensis etcubitermes sp. la théorie de la stigmergie : Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80, 1959. ISSN 0020-1812. URL http://dx.doi.org/10.1007/BF02223791.

Stephen GUERIN et Daniel KUNKLE : Emergence of constraint in self-organizing systems. *Nonlinear Dynamics, Psychology, and Life Sciences*, 8, 2004.

Hans HANSSON et Bengt JONSSON : A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

Jane HILLSTON : Fluid flow approximation of PEPA models. *In Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*, pages 33–43, 2005a.

Jane HILLSTON : Process algebras for quantitative analysis. *In LICS*, pages 239–248, 2005b.

Jane HILLSTON : Tuning systems : From composition to performance (the Needham lecture). *Comput. J.*, 48(4):385–400, 2005c.

Andrew HINTON, Marta Z. KWIATKOWSKA, Gethin NORMAN et David PARKER : PRISM : A tool for automatic verification of probabilistic systems. *In TACAS*, pages 441–444, 2006.

Thai Son HOANG et Jean-Raymond ABRIAL : Reasoning about liveness properties in Event-B. *In Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, pages 456–471, 2011. URL http://dx.doi.org/10.1007/978-3-642-24559-6_31.

Paul HORN : Autonomic computing : IBM's perspective on the state of information technology. 15 Oct. 2001. URL www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

Huu Tue HUYNH, Van Son LAI et Issouf SOUMARE : *Stochastic Simulation and Applications in Finance with MATLAB Programs*. Wiley Publishing, 2009.

M. Usman IFTIKHAR et Danny WEYNS : A case study on formal verification of self-adaptive behaviors in a decentralized system. *In FOCLASA*, pages 45–62, 2012.

Alexei ILIASOV, Elena TROUBITSYNA, Linas LAIBINIS et Alexander ROMANOVSKY : Patterns for refinement automation. *In Formal Methods for Components and Objects - 8th International Symposium, FMCO 2009, Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, pages 70–88, 2009. URL http://dx.doi.org/10.1007/978-3-642-17071-3_4.

C. B. JONES : Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, octobre 1983. ISSN 0164-0925. URL http://doi.acm.org/10.1145/69575.69577.

Ahmed HADJ KACEM, Amira REGAYEG et Mohamed JMAIEL : ForMAAD : A formal method for agent-based application design. *Web Intelligence and Agent Systems*, 5(4):435–454, 2007. URL http://iospress.metapress.com/content/h045h3r48141u757/.

Elsy KADDOUM, Claudia RAIBULET, Jean-Pierre GEORGÉ, Gauthier PICARD et Marie-Pierre GLEIZES : Criteria for the evaluation of self-* systems. *In 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, Cape Town, South Africa, May 3-4, 2010*, pages 29–38, 2010. URL http://doi.acm.org/10.1145/1808984.1808988.

Jeffrey O. KEPHART et David M. CHESS : The vision of autonomic computing. *Computer*, 36: 41–50, January 2003. ISSN 0018-9162. URL http://dx.doi.org/10.1109/MC.2003.1160055.

Ioannis G. KEVREKIDIS, C. William GEAR et Gerhard HUMMER : Equation-free : The computer-aided analysis of complex multiscale systems. *AIChE J.*, 50(7):1346–1355, 2004. ISSN 1547-5905. URL http://dx.doi.org/10.1002/aic.10106.

Savas KONUR, Clare DIXON et Michael FISHER : Analysing robot swarm behaviour via probabilistic model checking. *Robot. Auton. Syst.*, 60(2):199–213, février 2012. ISSN 0921-8890. URL http://dx.doi.org/10.1016/j.robot.2011.10.005.

Marta Z. KWIATKOWSKA, Gethin NORMAN et David PARKER : PRISM : probabilistic model checking for performance and reliability analysis. *SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

Marta Z. KWIATKOWSKA, Gethin NORMAN et David PARKER : PRISM 4.0 : Verification of probabilistic real-time systems. *In CAV*, pages 585–591, 2011.

Linas LAIBINIS, Elena TROUBITSYNA, Zeineb GRAJA, Frédéric MIGEON et Ahmed HADJ KACEM : Formal modelling and verification of cooperative ant behaviour in Event-B. *In Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, pages 363–377, 2014. URL http://dx.doi.org/10.1007/978-3-319-10431-7_29.

Leslie LAMPORT : The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. URL http://doi.acm.org/10.1145/177492.177726.

Sylvain LEMOUZY, Carole BERNON et Marie-Pierre GLEIZES : Living Design : Simulation for Self-Designing Agents. *In European Workshop on Multi-Agent Systems (EUMAS), Hammamet, 13/12/2007-14/12/2007*. Ecole Nationale des Sciences de l'Informatique (ENSI, Tunisie), décembre 2007.

Zohar MANNA et Amir PNUELI : Adequate proof principles for invariance and liveness properties of concurrent programs. Rapport technique, Stanford, CA, USA, 1984.

Mieke MASSINK, Manuele BRAMBILLA, Diego LATELLA, Marco DORIGO et Mauro BIRATTARI : On the use of Bio-PEPA for modelling and analysing collective behaviours in swarm robotics. *Swarm Intelligence*, 7(2-3):201–228, 2013.

Wafa MEFTEH, Frédéric MIGEON, Marie-Pierre GLEIZES et Faïez GARGOURI : S-DLCAM : A self-design and learning cooperative agent model for adaptive multi-agent systems. *In 2013 Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises, Hammamet, Tunisia, June 17-20, 2013*, pages 39–41, 2013.

Tan Chee MENG, Sandeep SOMANI et Pawan DHAR : Modeling and simulation of biological systems with stochasticity. *In Silico Biology*, 4, 2004. URL http://dblp.uni-trier.de/db/journals/isb/isb4.html#MengSD04.

Dominique MÉRY et Michael POPPLETON : Formal modelling and verification of population protocols. *In Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, pages 208–222, 2013. URL http://dx.doi.org/10.1007/978-3-642-38613-8_15.

Robin MILNER, Joachim PARROW et David WALKER : A calculus of mobile processes, (i et ii). *Inf. Comput.*, 100(1):1–40, 1992.

Olfa MOSBAHI et Leila JEMNI BEN AYED : Utilisation conjointe de B-événementiel et la logique temporelle TLA+ pour la modélisation et la vérification de systèmes réactifs. *In 6ème Conférence Francophone de Modélisation et Simulation Modélisation, Optimisation et Simulation des Systèmes*, Rabat, Maroc, 2006. URL http://www.isima.fr/mosim06/actes/articles/12-modelisation\%20et\%20verification\%20des\%20systemes\%20reactifs/255.pdf.

Florian NAFZ, Jan-Philipp STEGHÖFER, Hella SEEBACH et Wolfgang REIF : Formal modeling and verification of self-* systems based on Observer/Controller architectures. *In Assurances for Self-Adaptive Systems*, pages 80–111. 2013.

Vincent NIMAL : Statistical approaches for probabilistic model checking. Mémoire de D.E.A., Oxford University Computing Laboratory, 2010. URL http://www.prismmodelchecker.org/bibitem.php?key=Nim10.

Toshinori NIWA, Masaru OKAYA et Tomoichi TAKAHASHI : TENDENKO : Agent-Based Evacuation Drill and Emergency Planning System. *In Multi-Agent-Based Simulation XV - International Workshop, MABS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, pages 167–179, 2014. URL http://dx.doi.org/10.1007/978-3-319-14627-0_12.

Andrea OMICINI et Franco ZAMBONELLI : Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999. URL http://dblp.uni-trier.de/db/journals/aamas/aamas2.html#OmiciniZ99.

Dalimir ORFANUS, Peter JANACIK, Frank ELIASSEN et Pål ORTEN : High-level construction of emergent self-organizing behavior in massively distributed embedded systems. *In NaBIC'11*, pages 335–341, 2011.

Inna PEREVERZEVA, Elena TROUBITSYNA et Linas LAIBINIS : Development of fault tolerant MAS with cooperative error recovery by refinement in Event-B. *CoRR*, abs/1210.7035, 2012a. URL http://arxiv.org/abs/1210.7035.

Inna PEREVERZEVA, Elena TROUBITSYNA et Linas LAIBINIS : Formal development of critical multi-agent systems : A refinement approach. *In 2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 156–161, 2012b. URL http://doi.ieeecomputersociety.org/10.1109/EDCC.2012.24.

Andrew PHILLIPS et Luca CARDELLI : Efficient, correct simulation of biological processes in the stochastic Pi-calculus. *In CMSB*, pages 184–199, 2007.

Stefano PICASCIA, Bruce EDMONDS et Alison J. HEPPENSTALL : Agent based exploration of urban economic dynamics under the rent-gap hypotheses. *In Multi-Agent-Based Simulation XV - International Workshop, MABS 2014, Paris, France, May 5-6, 2014, Revised Selected Papers*, pages 213–227, 2014. URL http://dx.doi.org/10.1007/978-3-319-14627-0_15.

Corrado PRIAMI : Stochastic Pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

Aviv REGEV, William SILVERMAN et Ehud Y. SHAPIRO : Representation and simulation of biochemical processes using the Pi-calculus process algebra. *In Pacific Symposium on Biocomputing*, pages 459–470, 2001.

Urban RICHTER, Moez MNIF, Jürgen BRANKE, Christian MÜLLER-SCHLOER et Hartmut SCHMECK : Towards a generic Observer/controller architecture for organic computing. *In Informatik 2006 - Informatik für Menschen, Band 1, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden*, pages 112–119, 2006. URL http://subs.emis.de/LNI/Proceedings/Proceedings93/article4732.html.

Gabi Dreo RODOSEK, Kurt GEIHS, Hartmut SCHMECK et Stiller BURKHARD : Self-healing systems : Foundations and challenges. *In* Artur ANDRZEJAK, Kurt GEIHS, Onn SHEHORY et John WILKES, éditeurs : *Self-Healing and Self-Adaptive Systems*, numéro 09201 *in* Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL http://drops.dagstuhl.de/opus/volltexte/2009/2110.

M SCAPPIN et P CANU : Analysis of reaction mechanisms through stochastic simulation. *Chemical Engineering Science*, 56(17):5157 – 5175, 2001. URL http://www.sciencedirect.com/science/article/pii/S0009250901001804.

Ina SCHAEFER et Arnd POETZSCH-HEFFTER : Model-based verification of adaptive embedded systems under environment constraints. *SIGBED Review*, 6(3):9, 2009.

Giovanna Di Marzo SERUGENDO : Robustness and dependability of self-organizing systems - A safety engineering perspective. *In Stabilization, Safety, and Security of Distributed Systems, 11th International Symposium, SSS 2009, Lyon, France, November 3-6, 2009. Proceedings*, pages 254–268, 2009. URL http://dx.doi.org/10.1007/978-3-642-05118-0_18.

Claude E. SHANNON : A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948. URL http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf.

R.E. SHANNON : *Systems simulation : the art and science*. Prentice-Hall, 1975. ISBN 9780138818395.

Zhiguo SHI, Jun TU, Qiao ZHANG, Lei LIU et Junming WEI : A survey of swarm robotics system. *In Advances in Swarm Intelligence - Third International Conference, ICSI 2012, Shenzhen, China, June 17-20, 2012 Proceedings, Part I*, pages 564–572, 2012. URL http://dx.doi.org/10.1007/978-3-642-30976-2_68.

Graeme SMITH : *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

Graeme SMITH, Jeffrey W. SANDERS et Kirsten WINTER : Reasoning about adaptivity of agents and multi-agent systems. *In ICECCS*, pages 341–350, 2012.

Graeme SMITH et Luke WILDMAN : Model checking Z specifications using SAL. *In* Helen TREHARNE, Steve KING, Martin C. HENSON et Steve A. SCHNEIDER, éditeurs : *ZB*, volume 3455 de *Lecture Notes in Computer Science*, pages 85–103. Springer, 2005. ISBN 3-540-25559-1. URL http://dblp.uni-trier.de/db/conf/zum/zb2005.html#SmithW05.

Anton TARASYUK, Elena TROUBITSYNA et Linas LAIBINIS : Integrating stochastic reasoning into Event-B development. *Formal Asp. Comput.*, 27(1):53–77, 2015. URL http://dx.doi.org/10.1007/s00165-014-0305-z.

Guy THÉRAULAZ : L'intelligence collective des fourmis. *Le courrier de la Nature*, (250):46–53, 2010.

Xavier TOPIN, Christine RÉGIS, Marie-Pierre GLEIZES et Pierre GLIZE : Comportements individuels adaptatifs dans un environnement dynamique pour l'exploitation collective de ressources . *In Intelligence Artificielle Située, cerveau, corps et environnement (IAS'99), Paris, France, 25/10/1999-26/10/1999*. Hermès, octobre 1999.

Adelinde M. UHRMACHER et Danny WEYNS : *Multi-Agent Systems : Simulation and Applications*. CRC Press, Inc., Boca Raton, FL, USA, 1st édition, 2009. ISBN 1420070231, 9781420070231.

Norha M. VILLEGAS, Hausi A. MÜLLER, Gabriel TAMURA, Laurence DUCHIEN et Rubby CASALLAS : A framework for evaluating quality-driven self-adaptive software systems. *In 2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, pages 80–89, 2011.

Danny WEYNS, Henry Van Dyke PARUNAK, Fabien MICHEL, Tom HOLVOET et Jacques FERBER : Environments for multiagent systems : State of the art and research challenges. *In Environments for Multiagent Systems*, New York, NY, USA, Jul. 2004. Springer.

Franco ZAMBONELLI et Andrea OMICINI : Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, 2004. URL http://dx.doi.org/10.1023/B:AGNT.0000038028.66672.1e.

Hong ZHU : A formal specification language for agent-oriented software engineering. *In Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, AAMAS '03, pages 1174–1175, New York, NY, USA, 2003. ACM. ISBN 1-58113-683-8. URL http://doi.acm.org/10.1145/860575.860852.

Hong ZHU : Formal reasoning about emergent behaviours of multi-agent systems. *In* William C. CHU, Natalia Juristo JUZGADO et W. Eric WONG, éditeurs : *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE 2005), Taipei, Taiwan, Republic of China, July 14-16, 2005*, pages 280–285, 2005. ISBN 1-891706-16-0.