République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Ferhat Abbas - Setif 1 -
Faculté des Sciences
Département d'Informatique

N° Ordre :
N° Série :

Thèse

Présentée pour obtenir le diplôme de

Doctorat en sciences
Discipline : Informatique
Option : Génie Logiciel

Par :

Boucherit Ammar

Titre :

Contribution à la Conception d'Architecture Logicielle Sûre des Systèmes Critiques Basés Agent

Soutenue le 13/04/2019

Membres du jury :

Mr.	TOUAHRIA Mohamed	Président	Université de Ferhat Abbas	Sétif
Mr.	KAZAR Okba	Examinateur	Université de M Khider	Biskra
Mr.	AMIRAT Abdelkrim	Examinateur	Université M-C Messaadia	Souk Ahras
Mr.	KHABABA Abdallah	Directeur de thèse	Université de Ferhat Abbas	Sétif
Mme.	M.S CASTRO Laura	Co-directeur	Université de La Corogne	Espagne

$\grave{A}$
$\grave{A}$
$\grave{A}$

Remerciements

e travail de cette thèse est le fruit d'un effort long et continu et qu'il n'aurait pu voir le jour sans la contribution, l'aide et le soutien de nombreuses personnes honorables dont je leur dois exprimer ici mes vives reconnaissances. Mais, On peut pas commencer sans dire mille fois : Alhamdou Li-Allah, le tout puissant pour m'avoir donné la force et m'a guidé vers ces personnes pour achever cette thèse.

Mes remerciements vont d'abord à Pr. Khababa Abdallah d'avoir ma honoré par son encadrement de cette thèse ainsi que pour tous ses conseils, sa confiance, ses remarques et ses encouragements. Vraiment, j'admire sa façon de supervision qui m'a permis d'exprimer mes propres points de vue tout au long de la période de recherche.

Je suis également reconnaissant à ma co-directrice d'Espagne, Dr. Laura M. Souto Castro, pour la supervision, les encouragements et la gracieuse hospitalité au sein du groupe de recherche MADS. Je serai toujours reconnaissant pour tout ça, ...

Je remercie vivement l'ensemble des membres du jury, qui m'ont fait l'honneur d'avoir bien voulu accepter d'examiner et d'évaluer ce travail.

Je suis aussi particulièrement très reconnaissant à Pr. Osman Hasan (NUST), Pakistan, et à Pr. Barkaoui Kamel (CNAM), France pour la collaboration fructueuse qui a eu lieu dans mes papiers. J'admire vos compétences et vos talents et je suis très heureux de l'amitié que nous avons créée ensemble.

Mes remerciements éternelle vont à mes parents de m'avoir soutenu, de mon anniversaire j'usqu'à présent. Je suis persuadé que leur soutien ne cessera jamais tant qu'ils seront en vie. Mes parents, j'espère qu'Allah m'aide à vous faire le meilleur.

Mes sincères reconnaissances à ma femme Nadia qui m'a toujours appuyé et encouragé inconditionnellement au cours de ces longues années. Et cela malgré que mes heures de travail me font oublier les droits de ma famille. Merci à mes merveilleux enfants, Oualaa, Ayat Errahman et Achraf, qui m'ont fait un immense plaisir tous les jours.

Enfin, et ce n'est pas le moins important, je n'oublie pas mes amis proches dont les encouragements soutenus m'ont permis de finaliser cette recherche. je préfère ne citer personne plutôt que d'oublier l'un d'entre eux.

Abstract

ritical software systems are generally complex, distributed and often have intricate system states involving concurrency, parallelism and even real-time characteristics. Furthermore, and because of the rapid evolution of computer systems whether in the field of embedded or distributed systems, such software systems can be found almost everywhere in our daily lives domains like aeronautics, power plants, transportation, and healthcare. Consequently, their failure may lead to catastrophic consequences that may affect (or endanger) human life and its socio-economic environment.

Such kind of computer systems represent a real challenge for designers because it involves different quality levels that are directly proportional to the need for safety and reliability of the system being designed. Therefore, the main interest during the development of such software systems is not just limited to clearly define system functions, properties and interaction between components, but also on identifying/eliminating errors in the proposed code to ensure a successful system implementation.

In this thesis, we will contribute to the safe design of software architecture for agent-based critical systems by proposing a formal development approach combining both model-checking and property-based testing techniques. In particular, we have chosen Petri nets as a formalism to be used in the modeling phase. Then, we have proposed and implemented an automatic generating algorithm of rewriting logic based specification for the proposed Petri net models in order to minimize/avoid error in the specification phase. Finally, model-checking and property-based testing techniques were involved in the verification and testing phases.

Key words : Critical Software Systems, Formal methods, Multi-Agent Systems, Petri nets, Software Architecture

Résumé :

es systèmes logiciels critiques sont généralement complexes, distribués et comportent souvent des états impliquant des caractéristiques de concurrence, de parallélisme et mê e de temps réel. En plus, et du fait de l'évolution rapide des systèmes informatiques que ce soit dans le domaine des systèmes embarqués et/ou répartis, de tels systèmes logiciels sont présents presque partout dans les domaines de notre vie quotidienne, tels que l'aéronautique, les centrales électriques, les transports et la santé. Par conséquent, leurs échecs peuvent avoir des conséquences catastrophiques pouvant influer sur (ou mettre en danger) la vie humaine et son environnement socio-économique.

Ce type de systèmes informatique représente un défi pour les concepteurs puisqu'ils implique différents niveaux de qualité qui sont directement proportionnels au besoin de sécurité et de fiabilité du système en cours de conception. Par conséquent, le principal intérêt durant le développement de tels systèmes logiciels ne se limite pas à la définition claire des fonctions, propriétés du système et l'interaction entre ses composants, mais surtout à identifier et/ou éliminer les erreurs dans le code proposé du système pour assurer une implémentation réussie.

Dans cette thèse, nous allons contribuer à la conception d'architecture logicielle sûre pour les systèmes critiques à base d'agents par la proposition d'une approche de développement formelle combinant à la fois la vérification des modèles et des techniques de test basées sur les propriétés. En particulier, on a choisi d'utiliser les réseaux de Petri dans la phase de modélisation. Ensuite, on a proposé et implémenté un algorithme de génération automatique de spécification basée sur la logique de réécriture pour les modèles de réseaux de Petri proposés afin de minimiser les érreurs dans la phase de spécification. Enfin, les techniques de "modèle-checking" et celle de test basé sur les propriétés ont été impliquées dans les phase de vérification et de test.

Mots Clés : Architectures logicielles, Méthodes formelles, Réseaux de Petri, Systèmes logicielles Critiques, Systèmes Multi-Agents

الملخص

عادة ما تكون الأنظمة البرمجية الحرجة معقدة وموزعة وغالبًا ما تتضمن حالات تنطوي على خصائص التزامن والتوازي وحتى وقت الإستجابة الحقيقي. بالإضافة إلى ذلك ، وبسبب التطور السريع لأنظمة الكمبيوتر سواء في مجال الأنظمة المدمجة أو الموزعة ، فإن أنظمة البرامج هذه أصبحت موجودة في كل مجالات حياتنا اليومية ، مثل الطيران ، محطات الطاقة، النقل والصحة. وبالتالي ، فإن العطب في مثل حالاتها يمكن أن تكون لها عواقب كارثية قد تؤثر (أو تعرض للخطر) الحياة البشرية وبيئتها الاجتماعية و/أو الاقتصادية.

يمثل هذا النوع من أنظمة الكمبيوتر تحديًا للمصممين لأنه يشمل مستويات مختلفة من الجودة تتناسب بشكل مباشر مع الحاجة إلى الأمان والموثوقية للنظام الذي يتم تصميمه.
ولذلك، فإن التركيز الرئيسي خلال تطوير النظم والبرمجيات لا يقتصر فقط على تعريف أو تحديد واضح للنظام وخصائصه والتفاعل بين مكوناته، وإنما يتعداه إلى تحديد/إزالة الأخطاء في شفرة الكود المقترحة لضمان التجسيد والتنفيذ الناجح.

في هذه الأطروحة ، سوف نساهم في تصميم بنية برمجية آمنة للأنظمة الحرجة القائمة على الوكيل من خلال اقتراح منهج تطوير نظامي يجمع بين كل من تقنيتي فحص النماذج والإختبار المبني على الخصائص لشفرة الكود المقترحة. على وجه الخصوص، اخترنا استخدام شبكات بيتري في مرحلة النمذجة. بعد ذلك ، طوّرنا خوارزمية توليد المواصفات التلقائية على أساس منطق إعادة كتابة لنماذج شبكة بيتري المقترحة للتقليل من الأخطاء في مرحلة المواصفات. وأخيراً، يتم استخدام كل من التقنيات المذكورة للتحقق من صحة النموذج المقترح وكذا شفرة الكود.
الكلمات المفتاحية

الأنظمة البرمجية الحرجة، الأنظمة متعددة الأعوان، بنية البرامج، شبكات بيتري، طرق التحقق النظامية.

Publications

1. Ammar Boucherit, Laura M Castro, Abdallah Khababa, and Osman Hasan. Towards the formal development of software based systems : Access control system as a case study. *Information Technology And Control*, 47(3) :393–405, 2018
2. Ammar Boucherit, Abdallah Khababa, and Laura M Castro. Automatic generating algorithm of rewriting logic specification for multi-agent system models based on petri nets. *Multiagent and Grid Systems*, 14(4) :1–16, 2018 In Press
3. Boucherit Ammar and Khababa Abdallah. Contribution for the formal analysis of agent based critical systems properties. *Wulfenia Journal*, 20(1) :217–230, 2013
4. Boucherit Ammar and Khababa Abdallah. Towards the formal specification and verification of multi-agent based systems. *IJCSI*, 2011
5. Boucherit Ammar, Khababa Abdallah, and Belala Faiza. Rewriting logic based approach for the formalization of critical systems based on multi-agent system. *International Journal of Computer Applications*, 13(2), 2011
6. Ammar Boucherit, Kamel M Barkaoui, Osman Hasan, and Abdallah Khababa. An enhanced rewriting logic based semantics for petri nets. *Journal of Logical and Algebraic Methods in Programming*, IN REVIEW AFTER REVISION

Contribution Scientifique

Dans les papier III, IV, V, nous avons concentré notre attention principalement sur la proposition d'une approche basée sur la logique de réécriture pour l'analyse formelle des systèmes multi-agents. Ensuite, le papier I est une amélioration des travaux précedents et présente une approche dans laquelle nous avons combiné à la fois la technique de vérification "modèle checking" et le test basé sur la propriété (PBT) pour garantir l'absence d'erreur dans la conception et l'implémentation. Le papier II présente un algorithme bien défini pour la génération automatique de la spécification Maude de modèles de réseaux de Petri. Enfin, dans le papier VI , nous avons proposé une sémantique améliorée des réseaux de Petri basée sur la logique de réécriture, qui est l'un des formalismes les plus utilisés pour formaliser les systèmes multi-agents afin de pallier certaines limites dans la spécification de quelques extensions des réseaux de Petri.

Sommaire

Table des figures

Liste des tableaux

Introduction générale

'expansion technologique dans le monde numérique n'a pas seulement facilité et rendu nos activités quotidiennes très faciles et confortables, mais il devient tellement clair que notre vie est façonnée et de plus en plus contrôlée par des logiciels ; jusqu'à ce que nous ne puissions pas imaginer notre monde tel qu'il fonctionne aujourd'hui, sans de tels systèmes logiciels. Ils se retrouvent presque partout dans tout les domaines de notre vie quotidienne, tels que l'électroménager, téléphones mobiles, véhicules, transports, la santé, centrales electriques et nucleaires et la tendance ne cesse de s'accroître surtout avec l'Internet des choses (IoT).

En même temps, bien que les systèmes logiciels sont devenus une partie intégrante de notre vie, ils sont aussi responsables d'un nombre toujours croissant d'erreurs plus ou moins graves. *À* titre d'exemple, les conséquences de l'échec d'un robot domestique qui ne parvient pas à nettoyer toute la surface d'une chambre ne sont pas aussi désastreuses que si le système anti-collision aérien entrait dans un état provoquant la génération d'un chemin de vol d'évitement erroné.

Pour mieux s'en convaincre, citons notamment quelques unes des erreurs logicielles les plus spectaculaires. En 1996, des erreurs logiciels sont à l'origine de l'explosion du premier vol d'Ariane 5, après 40 secondes de son décollage. Ce qui résultait en une perte exorbitante, estimée à 8,5 milliards de dollars. Plus récemment, le plantage des systèmes informatiques de contrôle de l'aéroport international de Los Angeles en avril 2014. Heureusement, aucun accident ni blessure n'a été signalé, bien que de nombreux vols ont été retardés ou annulés. Finalement, une erreur dans le système "Decor" relié à météo France, à l'aéroport d'Orly en novembre 2015, a provoqué l'interruption du trafic durant une demi-heure.

Ces dernières sont des exemples de ce qu'on appelle des "systèmes critiques" car le cas de leurs échecs peut avoir des conséquences catastrophiques pouvant influer sur (ou mettre en danger) la vie humaine et son environnement socio-économique. En conséquence, les logiciels qui exploitent, gèrent ou contrôlent des systèmes critiques sont souvent complexes et leurs mauvais fonctionnements pouvant avoir des conséquences dramatiques ou inacceptables. Autrement dit, la variété d'applications de systèmes logiciels implique différents niveaux de qualité directement proportionnels au besoin de sécurité et de fiabilité du système à concevoir et du domaine d'application.
C'est dans ce contexte, celui des systèmes critiques, systèmes multi-agent et des méthodes formelles, que cette thèse s'inscrit. En effet, les systèmes critiques ainsi que les logiciels dédiés sont délicats à concevoir, à vérifier et à réaliser, ils continuent à être bridés par les fortes contraintes, en termes de performances, sûreté et d'autonomie, qui pèsent sur eux du fait de leur complexité et de leur nature non tolérante aux fautes. Par conséquent, la question primordiale est alors celle de savoir si l'on peut faire confiance à ces systèmes logiciels.

Problématique de Recherche

Dans nos jours, les systèmes logiciels sont devenues de plus en plus complexes et critiques. Ils sont généralement composés d'entités hétérogènes et distribuées, qui doivent coopérer et se coordonner de manière "intelligente" pour échanger et partager des connaissances afin de résoudre un problème ou accomplir une mission. La complexité susmentionnée des architectures de systèmes modernes et la délocalisation du traitement appellent de plus en plus sur l'utilisation des approches de conception adéquates pour soutenir le traitement décentralisé ; ainsi que des techniques rigoureuses d'analyse pour gérer la complexité et satisfaire les exigences de performance et de sécurité dans sa description architecturale.
Parallèlement, il est absolument connu que tout système logiciel un peu complexe est susceptible de comporter des erreurs qui ont échappé à la sagacité des meilleurs programmeurs et des procédures d'assurance qualité traditionnelles. Par conséquent, la complexité des systèmes logiciels modernes — en terme de taille du code, le nombre gigantesque d'états (cas de test) qu'il faut prendre en compte pour être sûr de leurs comportements en toutes situations — met aussi en évidence qu'une méthode de conception rigoureuse de tels syst mes logiciels, basée sur l'utilisation des méthodes formelles et assistée par des techniques et/ou outils de test adéquats, est devenue plus nécessaire que jamais.

En conséquence, et selon notre point de vue, les sous-problèmes auxquels on doit faire face pour augmenter la confiance dans nos systèmes logiciels critiques sont les suivants :

– Le choix du formalisme le plus approprié pour modéliser et assurer une bonne description de l'architecure comportementale d'un système critiques.
– Le choix d'une technique de vérification rigoureuse pour identifier le maximum possible des erreurs de spécification et de conception avant de passer à la phase de programmation.
– Le choix de la technique de test la plus performante pour assurer l'absence d'erreurs dans le code avant sa réalisation.

Objectifs de la thèse

Les approches de modélisation basées agents sont devenues très prometteuses, efficaces et apportent des avantages non négligeables dans la simulation et la mise en place des systèmes distribués et complexes. En effet, leur éfficacité est dû à cause de l'aptitude des agents de représenter directement les entités du système étudié, leurs comportements, interactions ainsi que la modularité qu'apporte les systèmes multi-agents (SMA). Néanmoins, l'amélioration de la qualité et de la fiabilité des SMAs à l'aide de méthodes formelles est un sujet récent, encore peu exploré. Parallèlement, et vu l'absence d'une méthode formelle standardisée pour le développement des logiciels basés agents pour les systèmes critiques, l'ingénierie logicielle pour les systèmes critiques basés agents est devenue une tâche significativement difficile du fait de la nature complexe, distribuée et critiques de la plupart de ses applications.

Par conséquent, le but de cette thèse est principalement de proposer une nouvelle approche pour le développement formel de l'architecture logicielle comportementale des systèmes critiques basés agents. L'approche proposée doit dépasser le stade de la description de l'architecture globale du systèmes en terme de ses composants et couvre les autres phases de développement telles que la vérification et le test. En particulier, les objectifs du travail rapporté dans cette thèse sont principalement :

1. Proposer et implémenter un algorithme pour la génération automatique de spécification Maude à partir du modèle du système.
2. Proposer une spécification améliorée basée sur la logique de réécriture pour les réseaux de Petri qui sont un formalisme puissant pour la modélisation des architectures basées agents.
3. Exploiter les outils d'analyse formels du système Maude pour vérifier les propriétés comportementales les plus importantes du système.
4. Utiliser une technique de test automatique afin d'améliorer la qualité du code proposé pour l'implémentation du système.

Approche, motivations et contributions

Pour faire face à la complexité, difficultés de vérification et aux exigences de qualité des applications critiques, les approches formelles sont progressivement impliquées dans le cycle de developpement des systèmes critiques pour initier toute volonté de sécurité. Généralement, une approche formelle utilise un langage formel bien défini syntaxiquement et sémantiquement pour écarter toute ambiguïté dans la spécification et permettre une compréhension claire et approfondie de ce que doit faire un système. Une telle approche permet aussi l'utilisation d'outils de vérification et de preuve, la possibilité de raffiner jusqu'à l'obtention d'un model formellement vérifié et même de générer automatiquement une implémentation abstraite qui sera par la suite testé par des outils de génération automatiquement des jeux de tests.

Dans un premier temps, nous avons choisi le formalisme des réseaux de Petri pour élaborer le modèle du système multi-agents. D'une part, ce choix peut être simplement justifié par leur sémantique simple, notation graphique, efficacité et aptitude de représenter la conccurence et le parallélisme, qui sont des caractéristiques principales des SMAs. D'autre part, les réseaux de Petri sont aussi caractérisés par la riche théorie sous-jacente développée autour d'eux, équipés par des méthodes d'analyse permettant d'exprimer et d'analyser un nombre important des propriétés générales (sûreté, vivacité, équité ...etc) et/ou spécifiques au système étudié.

Puis, une spécification algèbrique basée sur la logique de réécriture sera manuellement préparée pour le modèle de réseau de Petri proposé avant de pouvoir utiliser les outils de vérification associés au système Maude. Néanmoins, la préparation manuelle de cette spécification demande un temps considérable — surtout dans le cas de réseau de Petri de grande taille — et peut être sujette à des erreurs. Par conséquent, l'automatisation de cette phase dans le domaine des systèmes critiques sera très avantageuse car elle permettra de minimiser le taux d'erreurs dans la spécification et réduir le temps de la préparation de la spécification surtout dans le cas du modèle de réseau de Petri de taille considérable. Dans ce cadre, on a proposé et implémenté un algorithme pour la génération automatique de spécification Maude à partir de la présentation algèbrique d'un réseau de Petri. D'autre part, la sémantique existante des réseaux de Petri dans le cadre de la logique de réécriture souffre de quelques limites telles que la difficulté d'exprimer les réseaux de Petri avec des arcs inhibiteurs sans l'ajout d'autres opérateurs ainsi que l'impossibilité d'exprimer les réseaux de Petri avec des arcs à poids variable. En raison de ces carences et déficits, on a proposé une nouvelle sémantique améliorée basée sur la logique de réécriture pour les réseaux de Petri.
Ensuite, et parce que le modèle développé du système contient des exigences de sécurité telles que l'absence de blocages et/ou d'autres états critiques similaires susceptibles de

provoquer le crash du système, ce modèle doit être rigoureuse vérifié afin d'assurer une certaine confiance dans le comportement des agents et du système. En conséquence, il nous a semblé intéressant d'utiliser la technique du « model-checking » pour pouvoir vérifier le modèle du système de manière exhaustive et automatique pour assurer sa conformité aux spécifications.

Enfin, et même si la spécification a été automatiquement générée et le modèle a aussi été vérifié à l'aide d'une technique formelle, ça ne poura pas assurer à cent pour cent que le système multi-agents n'émergera pas de comportements incohérents lors de sa mise en place. Car, un système peut comporter des erreurs potentielles dans son implémentation. De ce fait, il nous semble aussi indispensable d'utiliser une autre technique, que l'on peut qualifier de complémentaire à la techniques de vérification précédente, permettant autant que possible l'élimination des erreurs pouvant apparaître dans l'implémentation du système. Cette technique est celle de test basé sur les propriétés (Property based testing), qui consiste à vérifier qu'un programme ou un psoeudo-code satisfait les propriétés spécifiées, à travers une grande variété d'entrées (tests) aléatoires fournies avec un générateur automatique. Cette manière de test a donné le couvert à notre technique comme si c'était une technique de test formelle, ou au moins d'être considéré comme étant une technique de test dans un cadre formel.

Organisation de la thèse

Ce document est composé de deux parties principales. La première partie est consacrée à la présentation des fondements scientifiques et structurée en deux chapitres. Au chapitre 1, nous présentons les notions théoriques les plus importantes de la logique de réécriture, langage de programmation "Maude" ainsi que les concepts élémentaires des réseaux de Petri avec sa sémantiques dans la logique de réécriture. Ensuite, nous présentons dans le chapitre 2 un récapitulatif des techniques de vérification et de test. Le chapitre 3 présente les concepts de base des systèmes multi-agents, nécessaires à notre étude, ainsi qu'un résumé des travaux de formalisation des systèmes multi-agents. La deuxième partie comprend trois chapitres et est réservée à la présentation de nos contributions. Le chapitre 3 présente en détail notre approche proposée pour le développement formel de systèmes critiques basés sur agents. Par la suite, le chapitre 4 présente notre algorithme pour la génération automatique de la spécification Maude de modèles de réseaux de Petri, suivi de la sémantique améliorée que nous avons proposé pour la spécification des réseaux de Petri dans le contexte de la logique de réécriture. Enfin, une conclusion de notre point de vue sur le futur de ce travail est présentée.

Partie I

Fondements Scientifiques

CHAPITRE 1

Le système Maude et réseaux de Petri

Ce chapitre introduit la logique de réécriture avec son langage Maude et met en évidence les concepts importants des réseaux de Petri.

Sommaire

1 Introduction

es méthodes formelles reposent sur l'utilisation de modèles, langages de spécification et des techniques d'analyses formelles afin offrir un cadre très rigoureux pour spécifier, concevoir et valider des systèmes à développer d'une manière systématique et rigoureuse en offrant des garanties fortes de leur fiabilité [7].
L'utilisation d'une méthode formelle revient d'une part à construire le(s) modèle(s) mathématique(s) correspondant(s) aux spécifications informelles d'un système. D'autre part, à utiliser de langages formels pour donner une spécification du système que l'on souhaite développer à un niveau de détail désiré. Une fois le modèle et la spécification sont prêts ; des techniques de vérification rigoureuses peuvent être utilisées pour raisonner sur le modèle afin d'analyser les propriétés pertinentes du système à développer. Enfin, une méthode formelle assure, entre autres, la vérification syntaxique des modèles, la génération automatique de théorèmes et leurs preuves, ou encore la traduction automatique des modèles construites en langages de spécification ou de programmation pour minimiser le taux d'erreurs humain surtout dans le cas des systèmes critiques.
Dans notre cas, nous nous sommes intéressés aux réseaux de Petri et la logique de réécriture qui offrent vraiment un cadre approprié pour la modélisation, la spécification et la vérification des architectures logicielles des systèmes critiques basés agents.
Les sections suivantes présentent l'essentiel des concepts nécessaires — de la logique de réécriture et des réseaux de Petri — pour la compréhension du présent travail. Néanmoins, en nous référant à [8–10] pour plus de détails.

2 La logique de réécriture

La logique de réécriture [11], dont les prémisses apparaissent dans [12], permet de décrire une sémantique opérationnelle relativement simple et d'offrir une expressivité puissante. La logique de réécriture est un modèle formel doté d'une sémantique bien définie. Elle est apparue au début des années 1990 juste après les travaux de José Meseguer sur les logiques générales pour décrire les systèmes concurrents. En effet, la logique de réécriture offre une base solide et rigoureuse pour raisonner d'une manière claire et correcte sur le changement (l'évolution) dans les systèmes concurrents non-déterministes ayant des états et évoluant en termes de transitions. La logique de réécriture est largement utilisée pour la simulation et la vérification des systèmes complexes [13–19] en utilisant son système Maude [20]. En plus, cette logique fournit un

cadre sémantique très expressif à travers duquel différents types de modèles concurrents (logiques, formalismes et langages) peuvent être exprimés et spécifiés de façon naturelle [8, 10, 21, 22].
Le formalisme de la logique de réécriture s'appuie sur la logique équationnelle élargie par des règles de réécriture conditionnelles pour la spécification du comportement des systèmes concurrents. Dans ce qui suit, nous allons présenter les définitions des notions principales de la théorie de réécriture.

2.1 Théorie de réécriture

Une *signature* dans la logique de réécriture est une théorie équationnelle (Σ,E) où Σ est une signature équationnelle formée par un ensemble de sortes et d'opérateurs et E est un ensemble de Σ-équations décrivant les propriétés structurelles d'un système donné. La théorie équationnelle (Σ,E) détermine une relation de congruence engendrée par E et un ensemble quotient des termes modulo l'ensemble des équations E (classes d'équivalence des termes $\mathcal{T}_{\Sigma,E}$). Une telle théorie équationnelle est entièrement définie par l'utilisateur, ce qui procure à la logique de réécriture l'expressivité et la flexibilité de définir des structures complexes et variés décrivant les états du système étudié.
En plus, les formules de la logique de réécriture, appelés dorénavant des *règles de réécriture*, opèrent sur les classes d'équivalence de termes $\mathcal{T}_{\Sigma,E}$ modulo E.

Définition 2.1.
Une règle de réécriture est une paire de termes orientée, notée $[t] \rightarrow [t']$*, où* t *est le membre gauche de la règle et* t' *son membre droit.*

Une règle de réécriture est utilisée pour décrire la possibilité de réécriture de la classe d'équivalence contenant le terme t en la classe d'équivalence contenant le terme t'.
Une *théorie* de la logique de réécriture est donnée par une signature (Σ,E) et par un ensemble de règles de réécriture R représentant des axiomes.

Définition 2.2.
une théorie de réécriture $\mathcal{R}$ *comprend 4 tuples :* (Σ, E, L, R) *tel que :*
- Σ *: est un ensemble d'opérations (symboles de fonctions) et de sortes.*
- E *: est un ensemble de* Σ*-équations.*
- L *: est un ensemble d'étiquettes.*
- R *: est un ensemble de règles de réécriture étiquetées pouvant être conditionnelles où une règle* r *a la forme suivante :* $crl[etiquette] : [t] \rightarrow [t']$ *if* C [i] .

i. C est une condition. Si C est vide, les règles sont appelées inconditionnelles et seront notées : $rl[etiquette] : [t] \rightarrow [t']$.

Dans le contexte de la logique de réécriture, un système concurrent peut être naturellement représenté par une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$ où sa signature décrit la structure statique du système (sortes, opérations et propriétés structurelles) et l'ensemble de ces règles de réécriture étiquetées (L, R) décrit la partie dynamique (les actions pouvant se produire dans tous les cas possibles). Chaque règle de réécriture correspond à une action élémentaire pouvant se produire en concurrence avec d'autres. En plus, une théorie de réécriture peut être enrichie par un ensemble de stratégies [23] décrivant des calculs et contrôlant l'application des règles de réécriture. Autrement dit, elles sont utilisées pour décrire la manière de déroulement des preuves menant à un résultat, pour contrôler le processus de réécriture lorsque les règles peuvent être hautement non déterministes, et aussi pour restreindre l'espace de recherche.

2.1.1 Règles de déduction

Dans un système concurrent, la séquence des transitions exécutées à partir d'un état initial donné, constitue ce que nous appelons le calcul qui correspond à une preuve ou à une déduction dans la logique de réécriture. Les règles formalisant l'opération de réécriture des termes sont appelées des règles de déductions. Autrement dit, pour une théorie de réécriture $\mathcal{R}$, on dit que $[t] \to [t']$ est prouvable dans $\mathcal{R}$ et on écrit $\mathcal{R} \vdash [t] \to [t'] ssi [t] \to [t']$ est obtenue par une application finie des règles suivantes :

1. *Réflexivité :* pour chaque $[t] \in T_{\Sigma,E}(X)$,
$$\frac{}{[t] \to [t']}$$
2. *Congruence :* Pour chaque symbole de fonction $f \in \Sigma_n$, $n \in$,
$$\frac{[t_1] \to [t'_1] \quad \dots \quad [t_n] \to [t'_n]}{[f(t_1 \dots t_n)] \to [f(t'_1 \dots t'_n)]}$$
3. *Remplacement :* Pour chaque règle de réécriture
$r : [t(\overline{x})] \to [t'(\overline{x})]$ *if* $[u_1(\overline{x})] \to [v_1(\overline{x})] \wedge \dots \wedge [u_k(\overline{x})] \to [v_k(\overline{x})]$ in R,
$$\frac{[w_1] \to [w'_1] \quad \dots \quad [w_n] \to [w'_n] \qquad [u_1(\overline{w}/\overline{x})] \to [v_1(\overline{w}/\overline{x})] \dots [u_k(\overline{w}/\overline{x})] \to [v_k(\overline{w}/\overline{x})]}{[t(\overline{w}/\overline{x})] \to [t'(\overline{w}/\overline{x})]}$$
où $\overline{w}/\overline{x}$ est une substitution de $x_i \mapsto w_i, 1 \leq i \leq n$.
4. *Transitivité :*
$$\frac{[t_1] \to [t_2] \qquad [t_2] \to [t_3]}{[t_1] \to [t_3]}$$

En effet, la réécriture d'un terme consiste à trouver une relation de réécrivabilité entre les classes d'équivalences de ce terme. Pratiquement, il est souvent utile de considérer les termes modulo une certaine théorie, comme la commutativité, l'associativité, l'idempotence, les éléments neutres, la distributivité de certains symboles par rapport à d'autres ou des équations spécifiques au domaine [11]. En plus, les règles de réécriture ne pourront être appliquées qu'aux formes normales [ii] des termes [24]. Par conséquent, la normalisation de celui-ci sera effectuée après chaque opération de réécriture. Cette stratégie de normalisation garantit de ne manquer aucune réécriture lorsque les règles sont cohérentes par rapport aux équations.

De manière générale, la déduction dans la logique de réécriture peut être vue comme une itération des étapes suivantes :
- Etape 1 : la règle de *remplacement* identifie toutes les règles de réécriture applicables à l'état global courant. En même temps, la règle de *réflexivité* sera appliquée aux sous termes non identifiés puis elle les transforme en eux-mêmes.
- Etape 2 : les règles de réécriture identifiées dans l'étape (1) sont exécutées indépendamment et en concurrence. Ensuite, la règle de *congruence* prend le role pour construire le nouveau terme global. Ces deux étapes (1) et (2) seront répétées jusqu'à ce qu'il n'y aura plus de règle applicable.
- Etape 3 : le terme final est construit par l'application de la règle de transitivité aux opérations de réécritures faites dès le début.

2.1.2 Réécriture Concurrente

La déduction dans la logique de réécriture correspond à la réécriture concurrente, c'est-à-dire à l'application simultanée de plusieurs règles de réécriture à différentes positions disjointes dans le même terme. En plus, les règles de réécriture définies dans une théorie de réécriture sont indépendantes les unes des autres, c-à-d., il n'y a aucun ordre d'exécution entre elles et à chaque étape de déduction, toutes celles dont la partie gauche correspond à un sous-terme de l'expression actuelle seront identifiées et pourront être appliquées en concurrence.

Définition 2.3.
Étant donnée une théorie de réécriture $\mathcal{R} = (\Sigma, E, L, R)$,
un séquent $[t] \rightarrow [t']$ *est appelé* $\mathcal{R}$*-réécriture concurrente si et seulement si, il peut être dérivé à partir de* $\mathcal{R}$ *par l'application finie des règles de déduction (1-4).*

ii. Un terme est dit en forme normale s'il est irréductible, c'est à dire si aucun de ses sous-termes n'est un redex (expression réductible,i.e, accepte une réduction)

2.2 Langages basés sur la logique de réécriture

Les langages de programmation basés sur la logique de réécriture sont nés suite aux travaux préliminaires sur le langage OBJ [25]. Dans cette sous-section, nous présentons en bref ELAN [26] et CafeOBJ [27]qui sont deux langages basées sur la logique de réécriture, alors que Maude [28] sera présenté dans la section suivante.

ELAN : Le système ELAN [29, 30] prend en charge la spécification de stratégies sophistiquées pour fournir un environnement de spécification, prototypage des déductions et guider le processus de réécriture pour réaliser des tâches complexes. Il offre un cadre logique normal et simple pour soutenir la conception des preuves de théorèmes. ELAN prend de la programmation fonctionnelle le concept des types abstraits de données et le principe d'évaluation de fonction basé sur la réécriture. En particulier, depuis le début du langage, les chercheurs d'ELAN ont développé de nombreuses applications de la logique de réécriture en tant que cadre logique bénéficiant grandement de l'utilisation de stratégies. Une originalité principale du langage est qu'il fournit des constructeurs de stratégies pour indiquer si un appel de fonction renvoie plusieurs résultats. Cette manipulation déclarative du non déterminisme fait partie d'un langage de stratégie permettant au programmeur d'indiquer le contrôle sur l'application des règles.

CafeOBJ : CafeOBJ [31] est un langage de spécifications algébriques exécutable qui a été développé au Japon. Il contient essentiellement OBJ [32] en tant que sous-langage fonctionnel. CafeOBJ dispose de puissantes fonctionnalités de composition de modules via des hiérarchies, des paramètres et des expressions de module. CafeOBJ prend également en charge les modules orientés objet et les systèmes de transition d'observation (OTS), un type spécial de spécifications comportementales idéales pour spécifier des systèmes de transition tels que les protocoles réseau et d'autres systèmes répartis. Les spécifications de CafeOBJ peuvent être formellement analysées de différentes manières.

CafeOBJ est prévu pour être principalement employé pour des spécifications des systèmes, la vérification formelle des caractéristiques, prototypage rapide, ou même de programmation. Du point de vue de la logique de réécriture, la fonction de recherche de CafeOBJ présente un intérêt particulier : elle prend en charge la recherche en largeur d'abord modulo un prédicat d'égalité spécifié par l'utilisateur [33], une forme très utile de vérification de modèle basée sur l'abstraction.

3 Le système Maude

Maude est un système performant et un langage de programmation déclaratif avec des performances compétitives. Il est caratérisé par sa simplicité, expressivité et par sa performance. Autrement dit, Maude est un langage qui offre peu de constructions syntaxiques et une sémantique bien définie. Maude a été influencé d'une manière importante par OBJ3. En particulier, le sous langage de la logique équationnelle de Maude contient essentiellement OBJ3 comme sous langage. Les buts du projet de Maude etaient de supporter la spécification formelle exécutable, et d'élargir le spectre d'utilisation de la programmation déclarative et des méthodes formelles pour spécifier, réaliser et analyser des systèmes de haute qualité dans des secteurs comme : réseaux de communication, bioinformatique et l'informatique répartie, ... etc [34].
L'unité de base pour le développement sous Maude est le module. Par conséquent, il faut écrire un ensemble de modules pour spécifier un système en utilisant Maude. En effet, il est important de noter que Maude permet de programmer à deux niveaux différents. Le premier est "Core Maude", implémenté en C++ et consiste en deux parties : un moteur de réécriture et un interpréteur. Core Maude qui accepte une hiérarchie de modules fonctionnels ou systèmes. Le deuxième c'est le "Full Maude" qui est l'extension du Core Maude en un langage plus riche permettant, notamment, de définir des modules orientés objet. En plus, le langage Maude définit deux niveaux de spécification. Un niveau concerne la spécification du système tandis que l'autre est lié à la spécification des propriétés.
En outre, Maude offre une collection puissante d'outils formels supportant différentes formes de raisonnement logique pour vérifier des propriétés de programme comprenant : Un model-checker, un prouveur de théorème, un outil d'atteignabilité (accessibilité) et un Analyseur de cohérence ...etc.

3.1 La syntaxe du langage Maude

Les notions syntaxiques les plus intéressantes du langage Maude seront présentées en brêve dans cette section.

3.1.1 Déclaration des types

Le langage Maude dispose d'un nombre impotant de types prédéfinis tels `Nat`, `Int`, `Bool`, `String` ... etc. En plus, le développeur peut créer ses propres types à partir de

constructions syntaxiques simples. Pour pouvoir utiliser ces nouveaux types il faut d'abord les déclarer dans une ou plusieurs parties déclaratives. Un type doit être déclaré en utilisant le mot clé **sort** suivi par le nom du type de données comme suit :

```
sort Nom_Du_Type .
```

Le cas où on a plusieurs sortes, on peut utiliser le mot clé **sorts** comme suit :

```
sorts Type_1 Type_2 Type_3 .
```

Maude supporte aussi la relation d'inclusion des types (sous types) en utilisant la déclaration suivante :

```
subsort Type_1 < Type_2 < Type_3 .
```

Cette déclaration indique que **Type_1** est un sous type du **Type_2** qui est lui même un sous type du **Type_3**. Par exemple, la déclaration :

```
subsort NZNat < Nat < Int .
```

où **NZNat** dénote le type des nombre naturels, **Nat** est pour les nombres naturels et **Int** pour les nombres entiers.

Après avoir vu quelques types prédéfinis, ainsi que la manière de déclarer d'autres nouveaux, il nous faut maintenant de voir la syntaxe pour déclarer les variables.

Déclaration des variables :

Pour déclarer une variable, on utilise le mot-clé **var** suivi du (**Nom_de_la_variable**), suivi de deux points avec un espace avant et après, suivi du (**Type_de_la_variable**), suivi d'un espace et d'un point. Par exemple :

```
var M : Int .
```

Vu que Maude est un langage très sensible, on note les remarques suivantes :

Remarque 3.1.

1. *Il faut être attentif qu'aprés chaque déclaration, un espace suivi d'un point doit être mis pour terminer la déclaration et que leur manque peut produire un erreur ou un comportement inattendu.*
2. *Dans le cas où on a plusieurs objets (types de variables, définitions de sous types ou plusieurs variables), on peut les déclarer en une seule fois en ajoutant un (s) au mot clé utilisé (**sorts**, **subsorts** ou **vars**) et en séparant chaque deux objets successives par un espace.*

3.1.2 Déclaration des opérateurs

Un opérateur est déclaré avec le mot clé **op** suivi de son nom, suivi de deux points, suivi de la liste des types de ses arguments (domaine), suivi de ->, suivi du type de son résultat (co-domaine) et éventuellement suivi d'une déclaration d'attribut. Par exemple :

```
op Op_Name : Sort_1 ... Sort_n -> Sort [Attribut_1 ... Attribut_k] .
```

En plus, Maude offre — même pour les opérateur — la possibilité de déclarer plusieurs opérateurs grâce au mot clé **ops**, à condition que ces opérations doivent possèder le même domaine et co-domaine. Nous présentons quelques déclarations d'opérateurs dans l'exemple suivant.

```
op 0 : -> Nat .

ops succ pred : Nat -> Nat .
op _+_ : Nat Nat -> Nat [assoc comm id:0] .
```

Dans cet exemple, l'opérateur `+` est déclaré avec deux attributs **"assoc"** (resp, **"comm"**), ce qui signifie que l'opérateur est associatif (resp, commutatif). En plus, l'attribut **"id"** définir l'élément d'identité (élément neutre) de l'opérateur.

Remarque 3.2. *1. Si la liste des arguments est vide, l'opération est une constante (le zero dans l'exemple précédent).*

2. Maude supporte le surcharge des opérateurs. c-à-d, qu'un même opérateur peut avoir plusieurs déclarations avec des domaines et des co-domaines différents.

3.1.3 Déclaration des équations

Nous rappelons qu'un terme est une constante, une variable ou bien le résultat de l'application d'une opération sur un ensemble de termes. Maude supporte deux types d'équations :

- **Equation inconditionnelle** :
  Une équation inconditionnelle est déclarée à l'aide du mot-clé **eq**, suivi d'un terme (sa partie gauche), du signe d'égalité **=**, puis d'un terme (sa partie droit) en terminant par un espace et un point. La déclaration sera comme suit :

  ```
  eq Terme_1 = Terme_2 [Attribut_1 ... Attribut_k] .
  ```
- **Equation conditionnelle** :
  La déclaration d'une équation inconditionnelle est fait à l'aide du mot-clé **ceq**, et on

ajoutant à la déclaration des équations incoditionnelles une condition. La syntaxe de cette déclaration sera comme suit :

```
ceq Terme_1 = Terme_2 if Condition .
```

3.1.4 Déclaration des règles de réécriture

Maude distingue deux types de règles de réécriture :

- **Règle inconditionnelle** :
  Une règle de réécriture inconditionnelle est déclarée à l'aide du mot-clé `rl`, suivi d'un terme (sa partie gauche), du symbole `=>`, puis d'un terme (sa partie droit)en terminant par un espace et un point. La déclaration sera comme suit :

  ```
  rl Terme_1 => Terme_2 .
  ```

- **Règle conditionnelle** :
  La déclaration d'une règle de réécriture conditionnelle est fait à l'aide du mot-clé `crl`, et on ajoutant à la déclaration des règles de réécriture inconditionnelles, une condition. La syntaxe de cette déclaration sera comme suit :

  ```
  crl Terme_1 => Terme_2 if Condition .
  ```

3.2 Les modules dans Maude

Dans Maude, les spécifications doivent être décrites dans des modules. Maude distingue trois types de modules : les modules fonctionnels, les modules systèmes et les modules orientés objets. Bien que les deux premiers s'existent dans Core Maude, le troisième s'existe dans Full Maude.

3.2.1 Modes d'importation des modules

Maude supporte la réutilisation des modules, en offrant trois modes pour leurs importations.

1. **Protecting** : Ce mode d'importation n'autorise aucune modification des types et des opérations du module importé. C-à-d, les types et les opérations ne peuvent qu'être employés strictement tels quels sont déclarés dans le module importé.

2. **Extending** : Ce mode d'importation est un peu souple et le développeur peut ajouter de nouvelles données ou bien des règles pour enrichir la syntaxe du module importé sans autoriser la confluence. C-à-d, ajouter sans modifier la signification des ceux qui sont déclarés dans le module importé.
3. **Including** : C'est le mode le plus souple et le développeur peut changer librement la signification des éléments déclarés dans le module importé.

3.2.2 Déclaration des modules

• ***Module fonctionnel*** :
Un module fonctionnel est une théorie équationnelle définissant la partie statique d'un système. La déclaration d'un module fonctionnel est donnée somme suit :

fmod nom_module **is**
< *Importation_d'autres_modules* >
< *Déclaration_des_types* >
< *Déclaration_des_opérateurs* >
< *Définition_des_équations* >
endfm
Le module suivant est un exemple :

```
fmod BASIC-NAT is
   sort Nat .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op _+_ : Nat Nat -> Nat .
   vars N M : Nat .
   eq 0 + N = N .
   eq s(M) + N = s(M + N) .
endfm
```

• ***Module système*** :
Un module système est utilisé pour spécifier la partie dynamique d'un système grâce aux règles de réécriture. Il permet de définir un ensemble de règles de calcul sur les éléments spécifiés dans le module fonctionnelle (ou bien, la partie fonctionnelle[iii]). La déclaration d'un module système est donnée généralement somme suit :

iii. il est possible de définir la partie fonctionnelle dans le module système lui même

mod nom_module **is**
< *Importation_d'autres_modules* >
< *Définition_des_règles_de_réécriture* >
endm

Voici un exemple illustrant l'utilisation de module système [35] :

```
mod COMPTER-CIGARETTES is
   protecting Nat .
   sort Etat .
   op c : Nat -> Etat [ctor] .
   op b : Nat -> Etat [ctor] .
   op _ _ : Etat Etat -> Etat [ctor assoc comm] .
   vars W X Y Z : Nat .
   rl[fumer] c (X) => b(X + 1) .
   rl[nouvelle] b(W) b(X) b(Y) b(Z) => c(W + X + Y + Z) .
endm
```

• ***Module orienté objet*** :
Maude offre aux utilisateurs la possibilité de spécifier les systèmes orientés objets concurrents en utilisant des modules orientés objets. Les modules orientés objets incluent implicitement le module **CONFIGURATION** qui définit les concepts de la programmation orientée objet : objets, messages et classes ... etc. Les modules orientés objet sont déclarés avec la syntaxe suivante :

(omod nom_module **is**
< *Importation_d'autres_modules* >
< *Définition_des_règles_de_réécriture* >
endom)

Remarque 3.3.
Bien que les modules orientés objets offrent une syntaxe plus appropriée pour décrire les entités d'un système orienté objet, ils peuvent en réalité être réduits à des modules système pour des fins de simplicité. Dans ce cas, le module ***CONFIGURATION*** *doit être importé.*

3.3 Simulation et vérification des systèmes

3.3.1 Simulation

L'environnement Maude offre un ensemble de commande pour simuler (voir) le comportement d'un système défini à travers une théorie réécriture. On cite ici la command "**`rewrite`**" (en abrégé, "**`rew`**"). L'interpréteur exécute la théorie de réécriture par une application "arbitraire" des règles de réécriture de gauche à droite à partir d'un état initial donné par l'utilisateur. L'utilisateur peut même limiter le nombre d'applications de règles de réécritures en donnant un seuil maximal, sinon, l'interpréteur ne s'arrêtera que lorsqu'aucune règle ne soit applicable ou même l'infini est supposé. En plus, Maude offre aussi la possibilité d'afficher des informations détaillées sur chaque réécriture effectuée et chaque tentative de réécriture conditionnelle en utilisant les commande de traçage.

3.3.2 Vérification

Maude offre pour la vérification deux outils complémentaire :

- **Model-Checker** : Pour analyser le comportement d'un système à travers cet outil, deux niveaux de spécification doivent être définis. Le premier niveau concerne la spécification du système (théorie de réécriture, état initial) tandis que l'autre est lié à la spécification des propriétés qui doit être décrite sous forme d'une formule en logique temporelle linéaire (LTL).
- **Outil d'Atteignabilité**(*Search Tool*) : permettant de vérifier explicitement l'existence/l'absence d'états critiques dans le système étudié à partir d'un état initial donné. En effet, cet outil peut être utilisé pour voir tous les chemins possibles pour atteindre un état et offre un temps de vérification très rapide si le meilleur état initial est utilisé.

4 Les réseaux de Petri

Les réseaux de Petri (RdP) ont été introduits pour la première fois en 1962 par Carl Adam Petri en 1962 dans sa thèse de doctorat à l'université de technologie, Darmstadt, Allemagne [36]. Ils sont encore connus comme l'un des formalismes les plus efficaces en raison de certaines de leurs caractéristiques distinctives, telles que la notation graphique, la sémantique simple et la théorie mathématique riche qui a été développé autour

d'eux [37]. Les réseaux de Petri sont largement utilisés pour la modélisation et l'analyse de systèmes discrets et dynamiques dans des contextes théoriques et industriels [38–42].

Définition 4.1.
Un réseau de Petri est formellement défini comme un 5-tuple $\mathcal{N} = (P, T, Pre, Post, M_0)$ *tel que :*

- P *: est un ensemble fini de places,* $P = \{P_1, P_2, ...\}$,
- T *: est un ensemble fini de transitions,* $T = \{T_1, T_2, ...\} P \cap T = \emptyset$,
- $Pre : P \times T \longrightarrow \mathbb{N}$, *tel que* $Pre(P_i, T)$ *est la fonction d'incidence qui définit le poids des arcs reliant* P_i *à* T *(marquage entrant de la transition* T*),*
- $Post : T \times P \longrightarrow \mathbb{N}$, *tel que* $Post(T, P_j)$ *est la fonction d'incidence qui définit le poids des arcs reliant* T *à* P_j *(marquage sortant de la transition* T*),*
- $M_0 : P \rightarrow$ *est le marquage initial.*

Le modèle de réseau de Petri est extrêmement simple, il consiste essentiellement en un ensemble de places, de transitions et d'arcs dirigés qui relient des places à des transitions et vice-versa mais jamais deux sommets de même nature. (voir Figure 1.1).

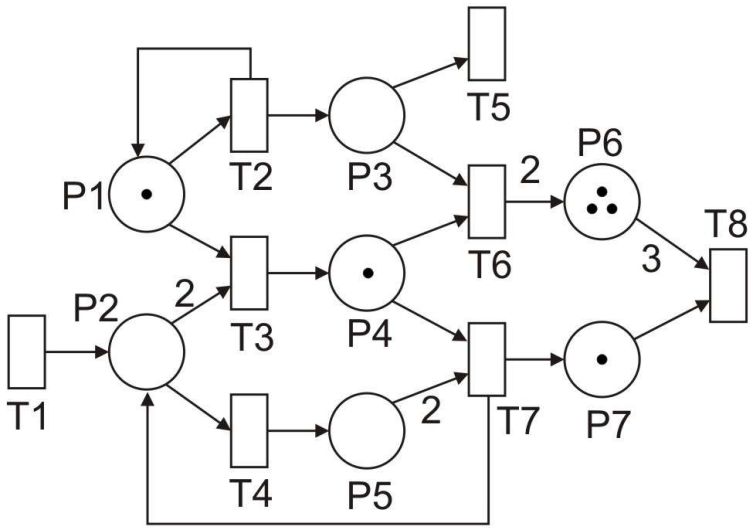

Figure 1.1: Exemple d'un Réseau de Petri

Les places (représentés par des cercles) représentent des états ou des conditions du système et peuvent contenir un nombre quelconque de jetons, qui sont représentés par des points noirs. Les transitions (représentées par des barres) représentent les changements d'état du système ou les événements susceptibles de se produire. Les arcs dirigés (indiqués par des flèches) définissent les conditions préalables et/ou les conditions postérieures à chaque transition en termes de places. Un arc lié à une transition t peut avoir une valeur entière qui définit son poids, c'est-à-dire le nombre

de jetons qui seront consommés ou produits après le franchissement de la transition t.
Un réseau de Petri est dit ***ordinaire*** si tous ses arcs ont un poids k = 1 (et donc non étiqueté). Les places liés aux transitions avec des arcs partant des places sont appelés ***places d'entrées***, et les transitions liées aux places avec arcs commençant de transitions sont appelés ***places de sorties***. Une transition sans place en entrée est une ***transition source***, une transition sans place en sortie est une ***transition puits*** ($T1$ et $T5$ dans la Figure 1.1). De plus, certaines transitions sont considérées comme étant en ***conflit*** si elles ont des ensembles (ou même des sous-ensembles) identiques des places d'entrées (par exemple, $T3$ et $T4$ dans la Figure 1.1).
En plus, un réseau de Petri est dit ***pur*** s'il n'a pas de transitions ayant une place d'entrée qui soit à la fois une place de sortie de cette même transition (boucle élémentaire). Dans le cas contraire, on parle de réseau de Petri ***impur***. Dans notre exemple de la Figure 1.1, la place $P1$ est à la fois une entrée et une sortie de la transition $T1$. Par conséquent, ce réseau de Petri n'est pas un réseau de Petri pur.
L'état global du système modélisé est exprimé par la distribution des jetons sur l'ensemble des places (appelé *marquage* du réseau de Petri). Le marquage M d'un réseau de Petri peut être changée en M' (écrit $M\ rightsquigarrow M'$), selon certaines règles simples (dites règles d'activation et franchissement).
Une transition $t \in T$ est dite active à un marquage M, si chacune de ses places d'entrées P_i est marquée au moins autant que $Pre(P_i, t)$.

$$\forall P_i \in \text{ensemble des places d'entrées de } t : M(P_i) \geq Pre(P_i, t)$$

En suite, la transition activée peut être franchis inconditionnellement. Par conséquent, un nombre de jetons égal à $Pre(P_i, T)$ est supprimé des places d'entrées et un nouveau nombre de jetons, égal à $Post(T, P_j)$, est produit et ajouté aux places de sorties.
Un réseau de Petri est dit "***k-borné***" si le nombre de jetons à ses places n'excède pas un nombre fini k pour tout marquage accessible à partir de M_0.

4.1 Réseaux de Petri et logique de réécriture

Comme c'est déjà mentionné, la logique de réécriture peut naturellement exprimer et unifier divers types des réseaux de Petri. Cette représentation a été proposée dans [11] et a ensuite été généralisée pour une large gamme de réseaux de Petri dans [10].
Pour en avoir une idée de cette représentation [iv], nous considérons notre exemple de la

iv. la sémantique des réseaux de Petri dans la logique de réécriture sera bien détaillée dans la section 2 du chapitre 6

Figure 1.1 en donnant la spécification correspondante comme suit :

```
fmod PN-SIGNATURE is
sorts Place Marking .
subsorts Place < Marking .
ops empty Initial : -> Marking .
ops P1 P2 P3 P4 P5 P6 P7 : -> Place .
op __ : Marking Marking -> Marking [assoc comm id: empty] .
eq Initial = P1 P4 P6 P6 P6 P7 .
endfm

mod EXAMPLE-PETRI-NET is
including PN-SIGNATURE .
var M : -> Marking .
rl [T1] : M => M P2 .
rl [T2] : P1 => P1 P3 .
rl [T3] : P1 P2 P2 => P4 .
rl [T4] : P2 => P5 .
rl [T5] : M P3 => M .
rl [T6] : P3 P4 => P6 P6 .
rl [T7] : P4 P5 P5 => P2 P7 .
rl [T8] : M P6 P6 P6 P7 => M .
endm
```

4.2 Quelques extensions des réseaux de Petri

Les réseaux de Petri ont été largement étudiés et étendus par de nombreux chercheurs. Certaines des extensions liées à notre travail sont décrites dans cette section, et un exemple de chacune de ses exensions est présenté dans la Figure 1.2.

– L'une des extensions les plus utilisées est le réseau de Petri, qui utilise un type particulier d'arcs, appelé arcs inhibiteurs. [43]. Il est représenté graphiquement par une ligne se terminant par un petit cercle attaché à une transition (voir arc reliant $P1$ à $T4$ dans la Figure 1.2 (a)). Cette extension des réseaux de Petri augmente considérablement l'expressivité en permettant un "test à zéro", ce qui représente des priorités et rend ainsi les réseaux de Petri aussi puissants que les automates à compteurs et les machines de Turing [44, 45].
– La deuxième extension est un réseau de Petri avec des poids d'arc variables (dynamiques). Cette extension est proposée pour avoir un outil assez puissant de

modélisation, d'analyse et de simulation pour les systèmes dynamiques complexes [46–48]. Dans ces réseaux de Petri, le poids d'un arc est dynamique (variable) et qui sera spécifié par une variable (voir arc reliant $P3$ avec $T4$ dans la Figure 1.2 (b)) ou par le nombre réel de jetons dans un lieu (voir arc reliant $P1$ avec $T2$ dans la Figure 1.2 (b)).

– La troisième extension est appelée réseaux de Petri colorés (CPN) [49, 50], qui conservent les propriétés utiles des réseaux de Petri standard tout en les enrichissant de structures de données complexes. La principale caractéristique qui rend les modèles des réseaux de Petri colorés plus compacts et pratiques réside dans la définition de jeton. Dans le cas simple, les jetons ont une valeur de données simple (appelée couleur du jeton) qui leur est attachée, puis le jeu de couleurs d'une place (type) est défini par les couleurs des jetons qu'il contienne (voir le réseau de Petri dans Figure 1.2 (c)).

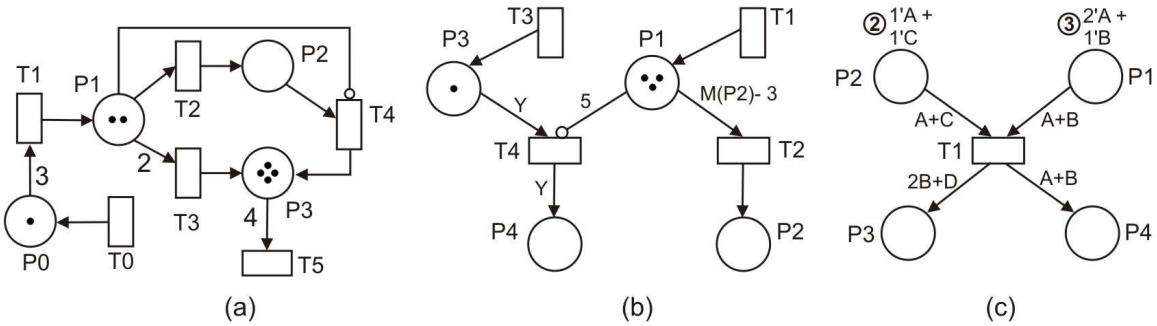

Figure 1.2: Exemple des extensions des réseaux de Petri

5 Conclusion

Dans ce chapitre, nous avons présenté rapidement les notions de base de la logique de réécriture qui est un cadre rigoureux permettant de raisonner de manière correct et définir de manière formelle les systèmes concurrents ayant des états et évoluant en termes de transitions. En plus, on a présenté le formalisme de réseaux de Petri ainsi que sa spécification dans le cadre du langage Maude par une théories de réécriture.
Dans la suite, nous allons présenter les techniques de vérification et de test ainsi que le paradigme agent afin de pouvoir présenter l'approche proposée pour le développement des systèmes critiques basés agent.

CHAPITRE 2

Techniques de Vérification

Dans ce chapitre nous allons présenter les techniques de vérification et de test qui seront utilisées afin de mieux comprendre l'approche de conception proposée.

Sommaire

1 Introduction

'une part, tout informaticien ayant déjà conçu des logiciels informatiques a sûrement rencontré des bugs qui sont généralement causées par une erreur qui n'est pas identifiées malgré une ou plusieurs relectures attentives. Cette difficulté de détection des bugs est dû au fait qu'une erreur peut être dans la plupart des cas une petite ligne de code mal écrite ou même un point manquant, mais quand ces erreurs s'y glissent, les conséquences sont parfois catastrophiques. En plus, il est souvent difficile à comprendre ce que fait vraiment un programme dès que celui-ci devient de taille non négligeable ; d'où la nécessité de chercher des techniques pour éviter ces problèmes.
D'autre part et plutôt avant cela, le passage à la phase de programmation n'est envisageable que si la spécification est bien validée par rapport aux fonctionnalités attendues du système. Autrement dit, lors du développement d'un logiciel, on ne doit pas se limiter à la vérification du produit final pour assurer leur bon fonctionnement, car il arrive que le logiciel est correct par rapport à sa spécification mais cette dernière est à la fois incomplète et inadéquate. Il devient par conséquent crucial d'utiliser des technique rigoureuses qui garantissent la conformité des modèles des systèmes vis-à-vis de leurs requis et de leurs fonctionnalités désirées afin d'identifier leurs disfonctionnements potentiels dès les premières phases de conception.
Dans le contexte des SMA, il est important de combiner plus qu'une technique de vérification et de test déjà existantes ou même de proposer de nouvelles qui permettent d'étudier les propriétés du bon fonctionnement du système dans le but de garantir l'absence de toute erreur ou défaillance dans ces systèmes.
Dans ce chapitre, nous allons présenter brièvement les techniques de vérification et de test, en se focalisant un peu plus sur celles utilisées dans notre approche de conception des architectures logicielles des systèmes critiques basés agent.

2 Techniques de vérification

Avec l'augmentation inévitable de la complexité des systèmes matériels et logiciels, la demande croissante de méthodologies qui peuvent accroître la confiance dans la conception et la construction de ceux-ci s'augmente aussi, puisque l'erreur dans quelques types de systèmes peut être désastreuse. Par conséquent, et du fait que plus une erreur est corrigée tôt, moins elle coûte cher, non seulement en argent, en temps-équipe, mais également en vie humaine. Cette situation a conduit au développement de certaines techniques de vérification pour faciliter la détection précoce des erreurs.

Dans les sections qui suivent, nous allons présenter une description des techniques de vérification et test les plus utilisées lors du développement de produits logiciels.

2.1 Preuve automatique

Les méthodes de preuve [51] sont basées sur des démonstrations mathématiques pour prouver que le programme satisfait sa spécification. Ces preuves sont effectuées à l'aide de l'assistant (outil) de preuve qui permet à partir d'un certain nombre d'axiomes et de règles d'inférence d'asurer la correction d'un programme, en prouvant des résultats intermédiaires au fur et à mesure comme lorsqu'on prouve un résultat en mathématiques.
Bien que les techniques de preuve ont l'avantage de ne pas reposer sur une construction explicite d'un modèle de comportement du type états/transitions, très gourmand en mémoire, ce qui permet de surmonter le problème d'explosion combinatoire, l'utilisation des techniques de preuve est rendue difficile par le fait que le développeur doit écrive lui-même la preuve, qui sera ensuite vérifiée par l'outil de preuve automatique. En plus, les techniques de preuves nécessitent aussi l'intervention de l'utilisateur pour guider le processus de vérification et restent donc un processus manuel.

2.2 Model-checking

la vérification de modèles, plus connue sous son nom anglais original de model-checking est une technique de vérification formelle qui est né au début des années 1980. Cette technique repose sur l'idée de construire le graphe de toutes les états possibles d'un système (ou son modèle), pour puisse s'assurer qu'aucune de ces états n'est en contradiction avec les comportements (propriétés) souhaités. En effet, cette technique est basée sur deux modèles : un modèle du système et un modèle de propriété que le modèle est censé préserver.
Les algorithmes de model-checking font appel à de nombreuses techniques afin de réduire la complexité de l'analyse et lutter contre l'explosion exponentielle du temps de calcul ou de la taille mémoire. Parmi ces techniques, on cite à titre exemple : réduction par symétrie, exploration aléatoire du graphe, représentations symboliques des espaces d'états, calcul à la volée du modèle et des propriétés ... etc [52]. Généralement, les sytèmes qui se prêtent le mieux à la vérification par model-checking sont : les systèmes critiques, les systèmes distribués et les systèmes réactifs.

La Figure 2.1 présente les composants d'une telle technique de vérification.

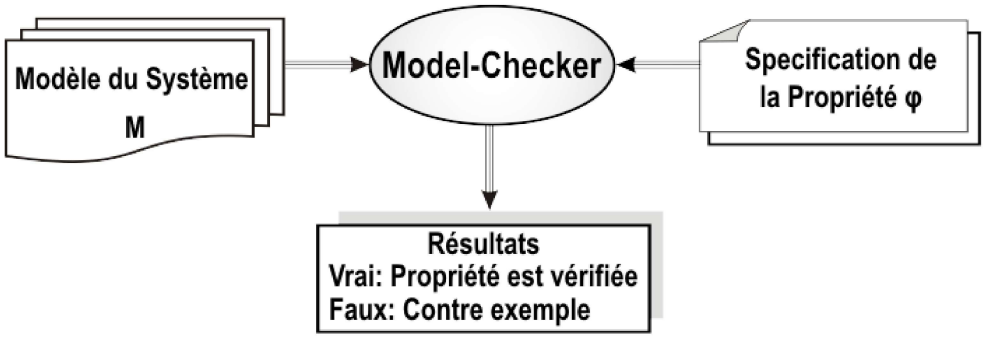

Figure 2.1: Composants de la Technique de Vérification par Modèle-Checking

2.3 Test

En effet, la première méthode utilisée par les développeurs pour traquer les bugs (erreurs) empêchant le bon fonctionnement d'un logiciel est le test. En principe, il repose sur l'exécution ou l'évaluation d'un logiciel ou d'un composant du logiciel, par des moyens généralement manuels, pour vérifier qu'il répond aux besoins et aux exigences recueillis auprès des utilisateurs, ou pour identifier les différences entre les résultats attendus et les résultats obtenus. Pratiquement, le testeur utilise le logiciel et prend en compte les erreurs qu'il rencontre, en essayant de prévoir tous les comportements possible d'un utilisateur.
Dans un premier temps, le test porte ses fruits, mais s'avère vite limitée, puisque cette technique est loin de fournir une vue exhaustive des erreurs eventuelles causés par une erreur de programmation ou à cause de manipulation incorrecte par l'utilisateur provoquant le dysfonctionnement du système.

2.4 Test basé sur les propriétés

Même si le test est la technique la moins coûteuse pour capturer une grande partie des défauts d'implantation (bugs), il est rarement considérés comme passionnants ; et il est souvent perçus comme un processus manuel nécessaire pour aider à augmenter le niveau de confiance ou garantir la fiabilité partielle d'un système. Néanmoins, le test n'offre aucune garantie sur la sureté d'un système mais plutôt des probabilités, et par conséquent, plus de tests seront automatisés, meilleure sera la situation. Les tests basés sur les propriétés (PBT) changent cela. Il s'agit de l'une des pratiques

les plus passionnantes du développement logiciel de ces dernières années. Il promet de meilleurs tests, plus solides et très concurrents aux autres techniques existantes, avec très peu de code. En plus, il offre beaucoup d'automatisation pour gagner du temps comme il couvre au mieux les cas de test suivant le nombre demandé par le développeur.

2.4.1 Principe de test

Le test basé sur les propriétés est une technique de test dynamique en boîte noire. Comme le montre la Figure 2.2, dans la technique PBT, le développeur doit écrire une expression quantifiée (formule) de manière universelle caractérisant le comportement du système à tester. Par conséquent, au lieu de choisir des données d'entrée spécifiques, l'outil PBT utilise des générateurs de données pour générer de manière aléatoire des données d'entrée acceptables. Ensuite, en exécutant autant de scénarios concrets que le développeur le souhaite et en diagnostiquant chacun de ces scénarios et en offrant des contre-exemples pour les échecs de test.

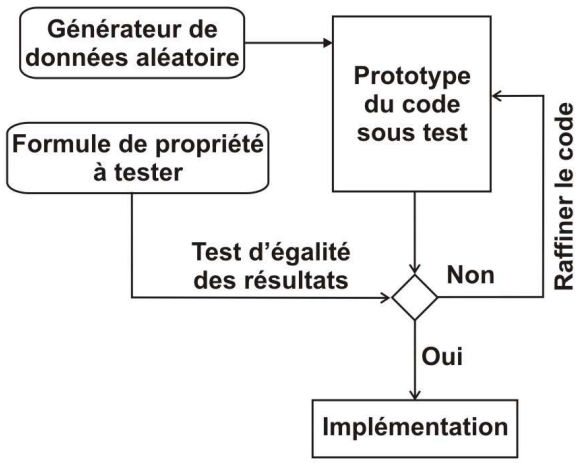

Figure 2.2: Principe de la technique PBT

2.4.2 Intérêts et applications de la PBT

D'une part, dans le domaine des systèmes distribués, parallèles et concurrents, les langages fonctionnels en général, et Erlang [53] en particulier, attirent de plus en plus d'attention en raison de nombreuses réussites [54]. Le niveau d'abstraction plus élevé qu'ils fournissent, associé à un certain nombre de propriétés telles que les fonctions d'ordre supérieur et le filtrage par motif, révèle ce paradigme de programmation comme

un choix technologique d'implémentation très puissant, qui aide les développeurs à mieux traiter les complexités inhérentes à la distribution, le parallélisme et la concurrence. En plus, Erlang est un langage fonctionnel en temps réel souple qui prend en charge la concurrence, la distribution et la tolérance aux pannes. Il permet également l'injection de mises à jour de code sans avoir à arrêter l'application.
D'autre part, l'intérêt pour la technique PBT et ses outils augmente rapidement, car les efforts requis par le développeur, comparés au nombre de tests pouvant être effectués simultanément, font de la technique PBT une option très attrayante et rentable. À ce jour, et à notre connaissance, Quviq QuickCheck [55] est l'outil de PBT le plus avancé et le plus puissant du marché. Il a été utilisé avec succès dans la recherche et l'industrie pour tester des systèmes complexes, critiques distribués et concurrents, mis en œuvre à Erlang et ailleurs [56–62].
De plus, il n'est pas étonnant que des outils PBT tels que QuickCheck soient nés et mis en œuvre en tirant parti des propriétés de langages fonctionnels tels que Haskell et Erlang. Ils représentent non seulement une alternative très performante pour les systèmes logiciels de test implémentés dans ces langages, mais ils peuvent également être utilisés pour tester des composants et des systèmes implémentés dans d'autres langages.

3 l'Outil LTL model-checker de Maude

L'outil LTL model-checker de Maude est un outil de vérification assez puissant qui a été conçu dans le but de vérifier des propriétés, exprimées en logique linéaire (LTL), relatives aux systèmes modélisés en logique de réécriture [63]. Le principe du model-checker de Maude est celui des méthodes de vérification explicites (basée sur la théorie des automates) utilisant la technique de calcul "à la volée" (on-the-fly) pour minimiser au maximum l'espace des états possible du système. Une étude comparative montrant la capacité du model-checker de Maude par rapport d'autre outils en terme de la taille des systèmes, temps d'exécution et taille mémoire nécessaire pour la vérification, a été donnée dans [63, 64].

3.1 Principe de vérification

Etant donnée un modèle M du système étudié et une formule φ représentant la propriété à vérifier, le problème pour le model-checker se résume en la vérification de la formule suivante : $M \models \varphi$

En effet, l'idée principale de la technique de vérification du model-checker de Maude — en tan que approche basée sur la théorie des automates — consiste à construire une structure *Kripke* K équivalente au modèle M et un automate de *Büchi* équivalent à la négation de la propriété à vérifier $B_{\neg\varphi}$. Autrement dit, un automate reconnaissant exactement les exécutions qui ne satisfont pas la formule φ.
Ensuite il synchronise fortement K et $B_{\neg\varphi}$ (de sorte que les deux avancent simultanément) pour obtenir un autre automate de *Büchi* B' tel que :

$$L(B') = L(K) \cap L(B_{\neg\varphi})$$

On aura par conséquent, un automate B' qui ne comporte que les exécutions de K qui ne vérifient pas la formule φ.
Donc, le problème de début du model-checker $k \models \varphi$ se ramène donc à une nouvelle question :
Est ce que le langage reconnu par l'automate B' est vide ?
Deux cas sont donc possibles :

- Si $L(B') = \emptyset$, then $M \models \varphi$. (la propriété φ est correcte dans M).
- Sinon, un countre-exemple sera donné pour montrer à quel point (état), la propriété φ n'est pas vérifiée dans M.

Ce principe peut être exprimé brièvement comme suit :
$M \models \varphi \iff K \models \varphi \iff L(K) \subseteq L(\varphi) \iff L(K) \cap \overline{L(\varphi)} = \emptyset \iff L(K) \cap L(\neg\varphi) = \emptyset$
Ce principe (démarche) peut être aussi illustré graphiquement comme c'est présenté dans la Figure 2.3.

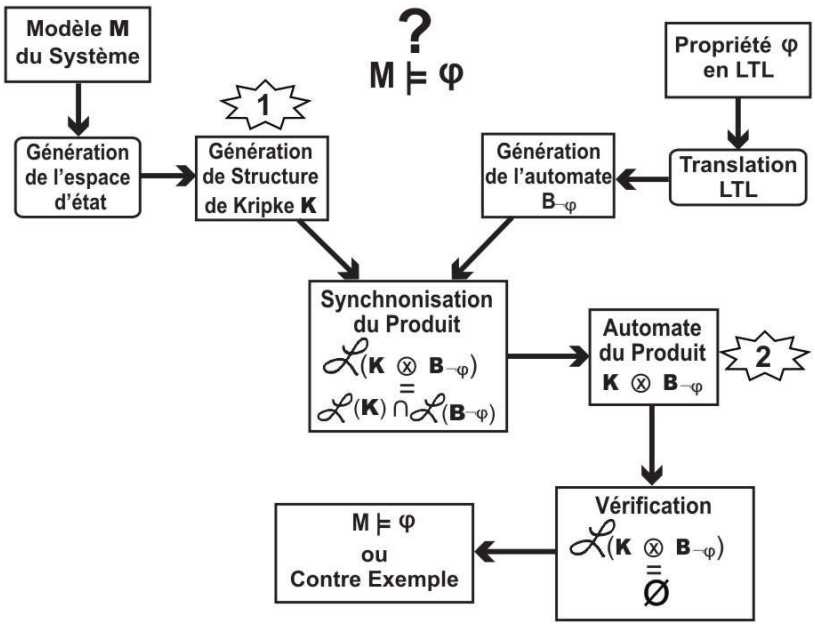

Figure 2.3: Démarche de Vérification par Modèle-Checking

Les numéros 1 et 2 encerclés par des étoiles dans la Figure 2.3 indiquent les étapes durant lesquelles se provoque l'explosion combinatoire et à ces points où les techniques d'optimisation ("à la volée" , par exemple) jouent leur rôles.

3.2 Composants du LTL model-checker de Maude

Le model-checker de Maude regroupe cinq modules principaux comme illustré dans la Figure 2.4 et qui sont :

LTL : formalise les définitions syntaxiques et sémantiques de la logique temporelle linéaire (LTL).

LTL SIMPLIFIER : réduit la taille de l'automate de Büchi qui correspond à la propriété à vérifier et par conséquent, minimiser le temps de calcul et l'espace mémoire nécessaire.

SAT-SOLVAR : permet de vérifier la satisfiabilité et la tautologie d'une formule LTL.

SATISFACTION : définit la syntaxe utilisée pour exprimer les propriétés du système.

MODEL-CHECKER : le module principal dans le processus d'analyse.

Les trois autres modules sont créés par l'utilisateur, et leurs rôles sont définis comme suit :

M-SYSTEM : définit la théorie de réécriture spécifiant le système.

M-PROP : définit les propriétés vérifiées par le système et qui seront considérés comme référence par le model-checker.

M-CHECK : C'est au niveau de ce module qu'on doit donner les formule des propriétés à vérifier en definissant un état initial pour démarrer l'opération de vérification.

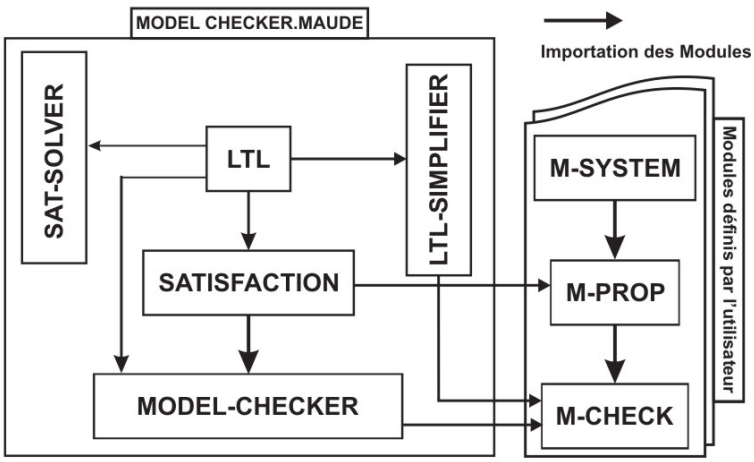

Figure 2.4: Modules Principaux du LTL Model-Checker de Maude

4 Conclusion

Nous avons présenté dans ce chapitre une description brève des techniques de vérification et de test les plus utilisées lors du développement de produits logiciels. En effet, nous ne seront pas intéressés que par la technique de vérification de model-checking et la technique de test basé sur les propriétés. La première technique sera utilisée pour vérifier la confirmité du modèle du système étudié vis-à-vis à sa spécification fonctionnelle, alors que la deuxième a pour role de tester automatiquement — avec le nombre nécessaire des cas de test — l'implémentation proposé pour le système étudié ou pour un de ses composants. Ces deux techniques seront par la suite exploitées dans notre approche pour le développment des systèmes critiques basés agent.

CHAPITRE 3

Formalisation des systèmes multi-agents

Dans ce chapitre nous allons présenter brièvement le contexte du domaine de recherche dans lequel cette thèse se situe. Notre but sera d'attirer l'attention du lecteur sur le problème de formalisation des systèmes multi-agents ainsi que les nombreux travaux réalisés sur ce sujet afin de mieux choisir le formalisme et la technique les plus adaptés à notre point de vue pour le développement des systèmes multi-agents.

Sommaire

1 Introduction

es systèmes modernes sont devenus de plus en plus complexes (interconnexion de matériels et de logiciels, délocalisation des traitements et mise en réseau d'organisations, ...) et font souvent appel à de nombreuses disciplines technologiques (automatique, informatique, électronique,...) pour résoudre les problèmes associés au développement de notre monde actuel. D'un point de vue informatique, les Systèmes Multi-Agents (SMA) se sont imposés comme étant le paradigme le plus approprié pour résoudre ces problèmes que ce soit dans le domaine des systèmes distribués, dans la robotique et de l'intelligence artificielle, ... etc [65]. Néanmoins, la bonne maîtrise du développement de tels systèmes impose, entre autres de savoir correctement les construire et les valider. Ce chapitre va aborder brièvement les formalismes et les techniques les plus utilisées en pratique dans le contexte des SMA pour assurer que ces systèmes sont correctement conçus, réalisés et répondent pleinement aux attentes fonctionnelles et de sécurité des utilisateurs finaux, surtout du fait de leur application dans des domaines critiques. Pour plus de détails sur les concepts liés à ce domaine, nous référons à [65, 66].

2 Concepts de base et applications

2.1 Notion d'Agent

Il n'existe pas encore un consensus sur la définition d'un agent et plusieurs définitions de ce paradigme peuvent être trouvés dans la littérature. À notre propos, nous retenons celle de wooldridge et Jennings [67] qui est illustré dans la Figure 3.1.

Définition 2.1.
un agent est un système informatique situé dans un environnement, et qui agit d'une façon autonome et flexible afin de répondre à ses objectifs de conception.

Partant de cette définition et/ou d'autres proposées pour le paradigme agent, on peut distinguer quelques propriétés clés telles que :

- **Intelligence** : en tant qu'un système informatique, l'agent peut avoir des capacités de raisonnement et/ou de représentation symboliques sur son environnement. En effet, l'intelligence d'un agent se manifeste dans sa capacité d'interagir d'une manière autonome et de prendre l'initiative d'un comportement tendant vers ses buts [68].

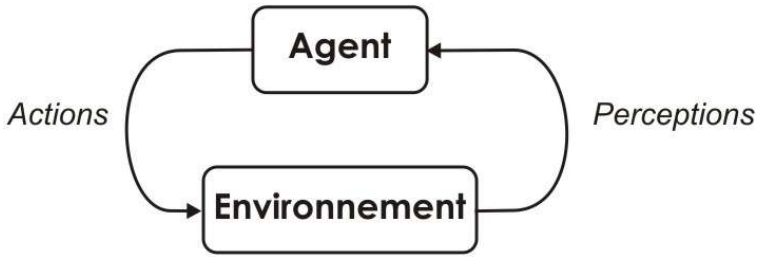

Figure 3.1: Agent et Environnement

- **Autonomie** : l'agent est capable d'agir et de contrôler ses actions et son état interne sans l'intervention d'un tiers. Il doit être aussi capable d'interagir au bon moment et prendre des décisions pour effectuer leurs tâches.
- **Sociabilité** : c'est la capacité d'un agent d'interagir, entretenir des échanges d'informations ou partage de connaissances avec d'autres agents de son environnement pour accomplir ses tâches, l'aider à accomplir les leurs tâches ou de collaborer afin de réaliser une opération commune.

2.2 Les systèmes multi-agents

La définition que nous adoptons pour le concept des SMAs est similaire à celle de J. Ferber et C. Draa [65, 69] où :

Définition 2.2.
Un système multi-agents est un système distribué composé d'un ensemble d'Agents qui sont conçus et implantés idéalement pour interagir dnas un environnement, le plus souvent, selon des modes de coopération, de concurrence ou de coexistence afin de résoudre une tâche donnée.

Cette définition met l'accent sur deux points fondamentaux caractérisant un système multi-agents. Le premier point est que le système soit composé d'un ensemble d'agents autonomes fonctionnant en parallèle et ayant un but. Le second point est que les agents disposent d'un moyen et/ou composant permettant la manipulation de l'environnement et l'interaction avec d'autres agents, comme un protocole de communication.
Selon [69], un SMA est généralement caractérisé par :

- chaque agent a des informations ou des capacités de résolution de problèmes limitées, ainsi chaque agent a un point de vue partiel ;
- il n'y a aucun contrôle global du système multiagent ;
- les donnés sont décentralisées ;
- le calcul est asynchrone.

2.3 Potentialités, applications et défis

Actuellement, la modélisation à base d'agents est la méthode la plus puissante et la plus connue pour modéliser tous types de systèmes articulés. Cette potentialité est due à :

1. L'aptitude d'un ensemble d'agents à résoudre des problèmes difficiles ou impossibles pour un agent individuel. Dans ce contexte, il est bien connu dans le domaine des systèmes multi-agents qu'une seule fourmi ou abeille n'est pas intelligente, mais que leurs colonies le sont. Ex : *trouver le chemin le plus court vers la source de nourriture*, ... etc.
2. La modélisation à base d'agents permet au concepteur de rechercher et de prédire le comportement dans des systèmes complexes (systèmes biologiques, cellules cancéreuses, fourmilière, ...).
3. La structure du système multi-agents se comporte comme un système ouvert, dans lequel différents objets ayant différentes propriétés peuvent être combinés dans un ensemble de relations complexes et évolutives.

Par conséquent, et du fait de leur potentialité de modélisation et leur capacité de résoudre des problèmes complexes, les systèmes multi-agents ont des applications variés dans des domaines différents tels que :
1) la modélisation et le contrôle de systèmes complexes, robotiques et critiques [70–77].
2) l'analyse et la simulation des environnements dynamiques, processus industriels, comportements socials et pour la résolution des problèmes par émergence [78–87].

Bien que le paradigme Agent et SMA offrent de nombreux avantages potentiels et devenu l'une des approches de pointe, cette potentialité ne doit pas cacher les difficultés liées à leur conception et qui peuvent discréditer et remettre en cause leur pertinence et leur crédibilité scientifique [88–90].
Avec une telle situation, une nouvelle génération de méthodes de développement des systèmes multi-agents a vu le jour pour guider les développeurs et leur faciliter les phases de modélisation, d'implémentation et de test [67, 91–97]. Une bonne comparaison entre ces méthodes selon différents critères peut être trouvé dans [98, 99]. Une telle méthode peut couvrir tout le processus de développement ou juste une partie. Néanmoins, ces méthodes souffrent généralement de l'absence d'une sémantique rigoureuse appropriée à la vérification formelle de propriétés du fait qu'elles se base sur des techniques et des formalismes eux-même manquent d'une sémantique formelle [100]. En plus, elles présentent des différences déterminantes pour le type de système à concevoir, de sorte qu'elles ne sont pas toutes adaptées à tout type de problème [101]. Ceci

est dû parce qu'il est pratiquement difficile — voire impossible — de développer des bibliothèques universelles et de concevoir des modèles sécurisés génériques pour les systèmes multi-agents en raison de ses vaste domaines d'application. Par conséquent, les concepteurs doivent utiliser des notations formelles lors de la phase de modélisation et des approches rigoureuse pour évaluer les propriétés clés et simuler le comportement du système multi-agents. Autrement dit, une telle approche pourrait être utilisée pour fournir une description formelle du comportement de l'agent et/ou des fonctions du système, fournir des directives pour la conception du système, faciliter leur analyse préliminaire, garantir la conformité du système à un ensemble de propriétés requises, puis prendre une décision à propos de sa faisabilité [3, 102].

3 Formalisation des systèmes multi-agents

La formalisation est une démarche qui commence par la construction d'un système formel (une description précise de la structure, des services, des relations et des taches) du système visé dont l'objectif est d'aider le concepteur de prouver la conformité du système avec sa spécification. En effet, la formalisation est une étape indispensable surtout quand il s'agit d'un système comportant des propriétés qui nécessitent une analyse approfondie et/ou complexe. En plus, puisque la compréhension de fonctionnement d'un système multi-agents ainsi que l'analyse de son comportement à partir de son prototype ou son implémentation n'est pas si simple et présente plusieurs inconvénients, il existe plusieurs tentatives de formalisation de ce type de systèmes.

Selon Ferber [65], cette étape s'avère très importante pour représenter :
- les actions des agents et de leurs conséquences dans l'environnement.
- le comportement interne et externe (observable) d'un agent.
- les interactions entre les agents ainsi que les diffiérents modes de communication (coopération, coordination et résolution de conflits)
- l'évolution du système multi-agents.

Dans ce contexte, un grand nombre d'approches, langages et de formalismes graphiques (et textuels) ont été proposés pour décrire tous les aspects des systèmes multi-agents. Néanmoins, il serait difficile de dresser une liste exhaustive de tous ces travaux qui ont été réalisés pendant plus de 20 ans, et nous ne rappelons donc que quelques travaux nous les considérons très importants du point de vue de leur pertinence et leur large utilisation. Par conséquent, nous classons ces travaux selon l'approche de spécification et la technique de vérification utilisées en deux grandes catégories :

3.1 Approches de spécification

- **UML et langages** : Parmi le grand nombre de langages de spécification des systèmes multi-agents que l'on trouve dans la littérature, tels que : CASL [103–106], SLABS [107–109], XABSL [110–112], COOL [113], AgentSpeak [114], ELMS [115], SDL [116], UML — qui est un langage standard unifié regroupant les principes de conception d'un ensemble de méthodologies orientées objet — reste le formalisme le plus utilisé pour décrire les différents aspects des systèmes multi-agents [76, 117–122]. Néanmoins, et du fait de sa caractère orienté objet, le langage UML n'est pas assez adéquat pour représenter les connaissances des agents ou les raisonnements logiques dans un SMA. Par conséquent, certaines extensions comme AML [123, 124], AUML [125–129] et Agent UML [130–133] ont été principalement conçues pour combler ce manque et capturer les aspects des systèmes multi-agents. En plus, UML n'est pas initialement basée sur une sémantique formelle permetant de fonder le raisonnement et l'analyse de propriétés d'un système multi-agents. Cette situation a conduit à la l'apparition de nombreux travaux pour la formalisation de UML elle même [134–139], et de développer un langage OCL [140] permettant de réduire l'ambiguïté et l'incompréhension des modèles en spécifiant des contraintes supplémentaires sur le comportement des composants du système étudié.
  D'autre part, nous pouvons trouver dans la littérature d'autres langages puissants tels que les langages de description d'architecture (ADL). En effet, Un ADL est un type particulier de langage qui fournit clairement une syntaxe concrète pour spécifier les architectures de système de manière plus abstraits, flexibles et robustes que les tentatives traditionnelles comprenant UML et les réseaux de Petri. Par conséquent, il n'est pas surprenant de trouver un grand nombre de travaux de recherche qui concerne la description des architectures de systèmes multi-agents [141–147]. Cependant, le problème majeur de la plupart des ADL est qu'ils manquent de sémantique formelle, du support explicite pour l'exécution d'une description de l'architecture, garantissant la conformité entre un système et une architecture spécifiée par l'utilisateur.
- **Réseaux de Petri** : sont toujours considérés comme l'un des formalismes de modélisation les plus utilisés pour aider les concepteurs aux premiers stades de la formalisation et ensuite la simulation des modèles multi-agents [148–150]. En effet, ce formalisme est fondé sur une représentation à la fois graphique et mathématique. L'aspect graphique permet de faciliter la conception et de visualiser le processus de communication dans un SMA, alors que l'aspect mathématique assure à ce formalisme des fondements théoriques rigoureux.

Par conséquent, les réseaux de Petri ont été largement utilisés pour décrire le comportement interne, externe et social des agents [151–154], les protocoles d'interaction [155–157] ainsi que la coopération et la coordinatioon dans les systèmes multi-agents [158–161]. De plus, les réseaux de Petri ont été aussi favorisés pour l'analyse de systèmes multi-agents en raison de la définition mathématique exacte de leur sémantique d'exécution ainsi que la théorie développée autour d'eux. Ensuite, les modèles de réseaux de Petri correspondants des systèmes multi-agents seront évalués en utilisant les méthodologies d'analyse et les techniques de simulation existantes pour les réseaux de Petri [162–166]. Par exemple, dans [167], les modèles de réseaux de Petri sont générés à partir des diagrammes UML multi-agents afin de simuler le comportement d'un agent. Par la suite, les techniques d'analyse de réseau de Petri sont appliquées pour le test automatique du comportement des agents. Enfin, de nombreuses extensions des réseaux de Petri ont été proposées pour bien décrire les agents et d'autres aspects dans les systèmes multi-agents [168–173].

- **Logique de réécriture** : Le système Maude se caractérise principalement par sa force expressive, ainsi que par le grand nombre d'outils d'analyse formels. Maude a été préconisé dans plusieurs travaux pour la spécification formelle des systèmes multi-agents [3–5, 174–179]. En plus, dans le papier [180], Maude a été utilisé pour présenter une implémentation du (langage simplifié du) langage de programmation d'agent cognitif 3APL. Dans une telle implémentation, les configurations 3APL sont représentées sous forme de termes et les règles de transition sont mappées en règles de réécriture. Dans la version améliorée [181], Maude est utilisé pour le prototypage du langage de programmation d'agent BUpL (langage de programmation Belief Update) et pour l'exécution de programmes d'agent. Maude est utilisé aussi dans [182] pour obtenir une description formelle des modèles d'organisation basés sur AGR (Agent-Group-Role), qui sont ensuite simulés et vérifiés à l'aide des outils associés. Enfin, le langage Maude est utilisé pour formaliser le comportement des agents du modèle DIMA [183]. Dans ce travail, les aspects inhérents aux modèles DIMA sont capturés et décrit dans la logique de réécriture. Ce travail a été étendu pour prendre en charge à la fois la description formelle et la vérification des modèles multi-agents DIMA [184]. Cependant, l'inconvénient principal et commun à tous ces travaux est que la traduction des modèles des systèmes multi-agents vers une spécification Maude est préparée manuellement et de manière ad hoc. Il peut donc être très difficile de s'assurer que cette spécification est une description fidèle des modèles des systèmes multi-agents étudiés surtout dans le cas des modèles complexes.

3.2 Approches de vérification

Du fait de leur complexité et leur large applications, la vérification des systèmes multi-agents prend de plus en plus d'importance où plusieurs travaux peuvent être trouvés dans ce domaine [175, 185–191]. D'une part, un nombre considérable de travaux ont été réalisés pour vérifier le comportement des agents [192–195], analyser des protocoles d'interaction et des langages de communication [196–199] ainsi que pour simuler et contrôler la sécurité dans des systèmes multi-agents [81, 84–86, 200–202].
D'autre part, peu de travaux de recherche sont orientés vers la création d'outils de vérification des systèmes multi-agent et la majorité sont basés sur des outils de modèle checking existants. Nous présentons dans la Table 3.1 un résumé des travaux les plus renommés dans ce contexte selon l'outil utilisé.

Travail	**Formalisme de spécification du système**	**Langage de spécification des propriétés**	**Outil de Vérification**	**Année**
Wooldridge et al. [203]	MABLE	LORA	SPIN	2002
Benerecetti et al. [204]	Diagrammes de décision binaires ordonnés (OBDD)	MATL	NuMAS	2002
Bordini et al. [205, 206]	AgentSpeak(F)	LTL	SPIN	2003
Gammie et al. [207]	Diagrammes de décision binaires ordonnés (OBDD)	CTL / LTL	MCK	2004
Lomuscio et al. [208]	Systèmes interprétés	CTL et logique épistémique	MCMAS	2006
Belala et al. [174]	Multi-formalismes (RdP, théories de réécriture, ... etc)	LTL	Maude Model-Checker	2006
Riemsdijk et al. [180]	3APL	LTL	Maude Model-Checker	2006
Lomuscio et al. [209]	Systèmes interprétés	CTLK	NuSMV	2007
Boudiaf et al. [184]	Théories de réécriture	LTL	Maude Model-Checker	2008
Nabialek et al. [210]	Multi-formalismes (réseaux d'automates temporisés, RdP temporisés, Estelle, ... etc)	CTLKD	VerICS	2008
Astefanoaei et al. [177]	Langage normatif pour SMA	LTL	Maude Model-Checker	2009
D'Souza et al. [211]	Machine (automate) à états finis	CTL / LTL	NuSMV2	2012
Laouadi et al. [182]	Agent UML	LTL	Maude Model-Checker	2017

Table 3.1: Récapitulatif des travaux utilisant des model-checkers dans le contexte SMA

3.3 Synthèse et discussion

La spécification d'un système multi-agents SMA implique l'identification d'un grand nombre d'entités et leurs relations. Pour cela, elle consiste à construire le modèle système avec le formalisme le plus approprié pour vérifier les propriétés en question et ne pas se limiter — s'il le faut — sur l'utilisation d'un seul formalisme afin de gerer les différentes perspectives du système [90, 212]. Parce que, comme c'est indiqué dans [213], si nous prenons l'exemple du paradigme des systèmes multi-agents ; la spécification de la structure du système peut être réalisée à l'aide des diagrammes UML, tandis que la dynamique des agents peut être spécifiée à l'aide de réseaux de Petri ou de règles d'inférence. Ensuite le modèle sera analysé par l'utilisation des outils externes de vérification. Ce qui aidera le concepteur à découvrir les erreurs dans le modèle du système multi-agents.

Dans ce contexte, nous constatons que UML et les langages de description d'architecture (ADL) sont principalement utilisés pour fournir une représentation claire des architectures des systèmes multi-agents avant leur développement. Cependant, ils reposent sur des méthodes relativement informelles, telles que les notations textuelles et/ou graphiques. En conséquence, de nombreuses tentatives ont été faites pour pallier le manque de sémantique formelle pour les descriptions architecturales.

Puisque nous somme intéressés dans notre étude par les architectectures comportementales, les réseaux de Petri ont été également utilisés pour décrire les architectures de systèmes en raison de leur potentiel de modélisation à un niveau d'abstraction élevé [214–218]. Par la suite, les travaux de recherche ont été concentrés soit sur la traduction (resp, la combinaison) du langage UML [219–227] et ADL [228–230] vers (resp, avec) les réseaux de Petri, ou bien, sur la définition des ADLs et des descriptions architecturales basés sur les réseaux de Petri [231–234] afin d'obtenir une description formelle et bénéficier de leurs outils d'analyse formelle existants. Parmi ces travaux, on trouve que quelques-uns [144, 235, 236] sont basés sur les réseaux de Petri et orienté vers les systèmes multi-agents.

On peut conclure ce point en affirmant que l'utilisation des RdPs pour la description des SMAs est une idée intrigante dans le sens où la symbolique assez simple des RdPs permet une compréhension rapide de l'architecture globale d'un SMA ou bien celles des agents composant le système. En plus, on ne peut pas dire que leur utilisation dans ce contexte peut paraître obsolète sous certains aspects et du fait de l'existence d'autres formalismes de représentation aussi puissants et intuitives. Car, il n'existe pas un seul modèle de RdP, mais toute une famille permettant de modéliser et de simuler les différents aspects (mécanismes et comportements) des systèmes multi-agents que

cela s'avère nécessaire tels que : RdP colorés [163], RdP orientés objets[169, 237, 238], RdP à prédicats [239], RdP imbriqués [240], RdP hiérarchi ues [241], RdP flous [242] et RdP à agent [172].
Dès lors, il existe prèsque toujours un modèle des RdPs qui peut être utilisé comme base pour la représentation et l'analyse des systèmes multi agents.

Similairement, on trouve que certains auteurs ont tenté de traduire les diagrammes UML [243–246] et les modèles de description d'architecture logicielle [247–252] vers la logique de réécriture, tandis que d'autres ont proposé des ADLs basés sur la logique de réécriture [253–255].
Par conséquent, cette situation des travaux de recherche nous a conduit à conclure facilement qu'en raison de la sémantique formelle de la logique de réécriture ainsi que l'aspect graphique et le potentiel de modélisation des résaux de Petri, le résultat sera très prometteur si on les utilises (combines) dans une même méthode pour décrire formellement les architectures logicielles des systèmes multi-agents.
D'autre part et d'un de vue de vérification, il est évident que la technique de modèle checking est la technique la plus présente dans la littérature. Par conséquent et après notre choix d'utiliser la logique de réécriture dans la phase de spécification, nous allons bénéficier du model-checker de Maude. En même temps, cela ne signifie pas que la technique de modèle checking est la seule technique nécessaire pour la vérification et la validation des systèmes multi-agents et d'autres techniques peuvent être utilisées.

4 Conclusion

Nous avons vu, tout au long de ce chapitre que le paradigme agent a été largement exploité dans les laboratoires de recherche et dans des applications rééls. Cependant, la conception des systèmes multi-agents sûres reste une tâche qui nécessite plus de maîtrise. Pour cela, plusieurs formalismes ont été appliqués pour résoudre ce problème, et simplifier les tâches de spécification et de vérification en garantissant la surêté des SMAs développés.
Dans la deuxième partie de cette thèse, nous présenterons en détail le cadre applicatif de notre thèse. Nous contribuerons à la spécification et la vérification formelles des SMAs par l'enrechissement de la sémantique basée sur la logique de réécriture pour le formalisme des réseaux de Petri, la génération automatique de la spécification et par l'application de l'outil model-checker du système Maude.

Partie II

Approche et Contributions

CHAPITRE 4

Une approche formelle de développement de systèmes logiciels

Ce chapitre introduit l'approche proposée pour le développement formelle des systèmes logiciels.

Sommaire

1 Introduction

Dans le monde actuel, les systèmes logiciels occupent de plus en plus de nos vies et ils deviennent difficile et étrange de trouver une personne ou une entreprise qui n'utilise pas d'ordinateurs ni de logiciels dans ses activités quotidiennes. Ces systèmes logiciels ont rendu notre vie quotidienne plus facile et plus confortable, au point que nous ne pouvions plus imaginer notre monde tel qu'il est maintenant ; fonctionne sans des systèmes logiciels. Par exemple, de tels systèmes sont largement présents dans notre environnement pour de nombreuses utilisations à la maison, à la banque, à l'hôpital et même pour les systèmes dits critiques. Par conséquent, cette variété de systèmes logiciels implique différents niveaux de qualité, directement proportionnels au besoin de sécurité et de fiabilité de chaque système ou domaine d'application. Par conséquent, l'utilisation d'approches de conception adéquates ainsi que de techniques d'analyse rigoureuses revêt de plus en plus d'importance.
Selon notre point de vue, l'objectif fondamental d'une approche de développement de tels systèmes logiciels surtout critique peut être résumé dans les trois points suivants :

1. Définir clairement la structure et le comportement du système en utilisant les formalismes de modélisation et/ou les langages de spécification formelles les biens adaptés..
2. Identifier les dangers, les risques et les erreurs du système — qui peuvent coûter cher — pour les prendre en charge et mieux vérifier ses propriétés d'intérêt en utilisant des techniques de vérification formelles.
3. S'assurer que l'implémentation du système satisfait à la spécification et que le la phase de codage ne portera pas des erreurs en utilisant des techniques de test avancées.

Dans ce chapitre, nous proposons une nouvelle approche hybride combinant à la fois la technique de vérification formelle "model-checking" avec la technique de test basé sur la propriété (PBT) pour pouvoir vérifier à la fois le modèle du système et son implémentation logiciels. L'approche proposée utilise les réseaux de Petri[256], qui est un formalisme puissant pour la modélisation et la réécriture de la logique [11], qui représente une logique très expressive pour donner une sémantique formelle lors de la spécification des modèles de réseaux de Petri. Ensuite, le model-checker de Maude [63] est utilisé pour vérifier les propriétés importantes et critiques du système, et enfin le test basé sur la propriété (PBT) [257, 258] en tant que technique de test automatique basée sur la génération aléatoire de séquences de tests (scénarios de test) pour élever le niveau de confiance sur la réalisation du système en cours de développement. En

fait, cette proposition est l'amélioration, l'extension et la combinaison de nos travaux antérieurs [4, 259], où nous avons utilisé ces techniques séparément.

2 Travaux en rapport

De nombreux travaux dans le domaine de la formalisation et d'analyse des architectures logicielles peuvent être trouvés dans la littérature [260–262]. Dans cette section, nous résumons brièvement les aspects les plus liés à notre travail.
Tout d'abord, sur la base des formalismes de modélisation, nous constatons que UML et les langages de description d'architecture (ADL) sont principalement utilisés pour fournir une repr sentation claire des architectures. Cependant, ils reposent sur des méthodes relativement informelles et de nombreuses tentatives ont été faites pour pallier ce manque de sémantique formelle.
Par exemple, certains travaux ont porté sur la combinaison ou la traduction de modèles UML [219, 221, 263, 264] et ADLs [228–230, 265–267] vers les réseaux de Petri pour profiter de ses outils d'analyses liées. Beaucoup d'autres ont essayé de traduire les diagrammes UML [243–246] ainsi que les descriptions des architectures logicielles [248–251] vers la logique de réécriture. En outre, certains auteurs ont proposé des ADLs basés sur la logique de réécriture [253–255, 268].

Les réseaux de Petri ont également été largement utilisés pour décrire les architectures de systèmes en raison de leur potentiel de modélisation et du niveau d'abstraction élevé [220, 234, 269, 270]. Dans ce contexte, d'autres auteurs ont cherché à définir des (resp, utiliser) ADLs basés sur les réseaux de Petri (resp, les réseaux de Petri comme ADL) [232, 235, 271] afin d'obtenir une description formelle et un mécanisme de simulation pour les architectures logicielles.
D'après tout ce que nous avons présenté, on peut donc facilement conclure qu'en raison de la manque d'une sémantique formelle de la plupart de ces formalismes, de nombreux travaux se sont concentrés sur la recherche d'une sémantique bien définie pour fournir une description formelle des architectures logicielles.
Deuxièmement, en ce qui concerne les techniques d'analyse des architectures logicielles, la logique de réécriture est largement intégrée dans le processus de développement afin de décrire les modèles d'architecture proposés, par exemple [4, 5, 272–274] et par la suite, l'ensemble d'outils d'analyse formels proposés par le système Maude est utilisé. En plus, la technique de vérification "modèle-checking" est l'une des techniques les plus appliquées pour la vérification des architectures logicielles [275–277]. Les lecteurs intéressés sont référés à [278], où une classification et une comparaison très complètes

des techniques de vérification "modèle-checking" pour les architectures logicielles sont présentées. De plus, cette technique est également associée à des tests pour assurer la fiabilité des architectures logicielles, on cite par exemple [279–282].
Troisièmement, un nombre considérable de travaux ont été réalisés dans le cadre de la combinaison des réseaux de Petri avec la technique de vérification "modèle-checking". Par exemple, dans [283], des réseaux de Petri colorés hiérarchiques (HCPN) sont utilisés pour la modélisation et la simulation des modèles de composants COMPOR, et la technique de vérification "modèle-checking" est aussi utilisée à prouver leur exactitude. Dans une telle approche, il est impossible d'affirmer que les modèles d'interaction CMS (Component Model Specification) sont corrects en considérant toutes les architectures de composants et séquences d'interaction possibles. C'est pourquoi les auteurs espéraient définir une méthode générique pour la modélisation et la vérification des modèles de composants utilisant HCPN et la technique de vérification "modèle-checking" . Dans un deuxième papier, [284], une méthodologie de vérification formelle utilisant le réseau de Petri et des techniques de vérification "modèle-checking" est proposée. Le réseau de Petri est utilisé pour modéliser le comportement du système et la technique de vérification "modèle-checking" est utilisée pour vérifier certaines propriétés essentielles dans le système étudié, notamment la sécurité, la convivialité et l'équité. Certains problèmes, tels que l'analyse de couverture et d'équivalence, n'ont pas été étudiés et les auteurs s'attendent à trouver une solution pour ces problèmes par l'utilisant de la technique de vérification "modèle-checking" à l'avenir.

3 Caractéristiques d'une approche formelle pour SMA

En raison de la nature complexe et de l'hétérogénéité des composants du système multi-agents, les concepteurs doivent souvent simuler le comportement du système au cours du processus de développement et vérifier la conformité entre le modèle conçu et le comportement attendu du système étudié ; afin de prendre une décision quant à sa faisabilité avant son implémentation réelle [164, 285–287].
En fait, de nombreux travaux ont été réalisés pour la formalisation de MAS et peuvent être trouvés dans la littérature telle que [187, 288–291]. Précisément, à la phase de modélisation, plusieurs formalismes ont été proposés et utilisés afin de passer de la description du système informel ou moins formel à sa spécification formelle, puis les agents individuels avec leurs relations sont décrits en termes d'activités au moyen de réseaux de Petri, machine à états, règles logiques ou instructions conditionnelles (si-alors) [292, 293]. De même, au niveau de la vérification, le vérificateur de modèle est l'une des techniques les plus utilisées pour la vérification des propriétés MAS

[203, 289, 294–296].
En plus et afin d'assurer au maximum la correction du code proposé pour l'implémentation du système à développer, il est judicieux d'utiliser la technique de test basé sur les propriété à la phase de test. Cette dernière peut être considéré comme si c'était la technique de test dans un cadre formel du fait quelle est basée sur la génération automatique des sénarios pour tester le code et elle génère un contre exemple dans le cas d'erreur.

4 Approche de développement formelle proposée

Le processus de conception logicielle est souvent considéré comme une séquence de phases qui transforme un ensemble de spécifications globale et informelles en une spécification technique détaillée pouvant être utilisée pour le développement. Autrement dit, elle parte de l'architecture du logicielle pour transformer une description plus abstraite en une description plus détaillée.
Notre approche pour le développement de systèmes logiciels critiques envisage les phases suivantes du développement logiciel : modélisation, spécification, validation et tests. Nous partons d'une description abstraite du système en utilisant le formalisme approprié et avançons en séquence d'une phase à l'autre jusqu'à l'obtention d'une squelette valide de la réalisation du système. Notant que notre intention n'est pas de proposer un cycle de développement méthodologique, mais une approche systématique pour l'analyse du système (modèle et code). Notre approche pourrait potentiellement correspondre à n'importe quel cycle de développement logiciel.
Les sous-sections suivantes décrivent les étapes de notre approche, qui couvre le processus de développement logiciel, de la spécification abstraite à l'implémentation.

4.1 La phase de modélisation

L'objectif de cette première phase est de créer la première version du modèle de l'architecture du système. En fonction du système étudié, un système peut comporter plusieurs phases opérationnelles, généralement : démarrage, initialisation, traitement normal et arrêt. Habituellement, chaque phase opérationnelle a sa propre architecture. De plus, au sein de chaque composant ou sous-système du système, une autre architecture peut décrire et orchestrer son comportement spécifique.
Généralement, le modèle système est composé de deux types de descriptions : statique

et dynamique. La description statique concerne les différents éléments composant l'architecture, alors que la description dynamique concerne leur configuration, c'est-à-dire la définition des règles régissant le comportement du système en fonction des actions possibles autorisées.

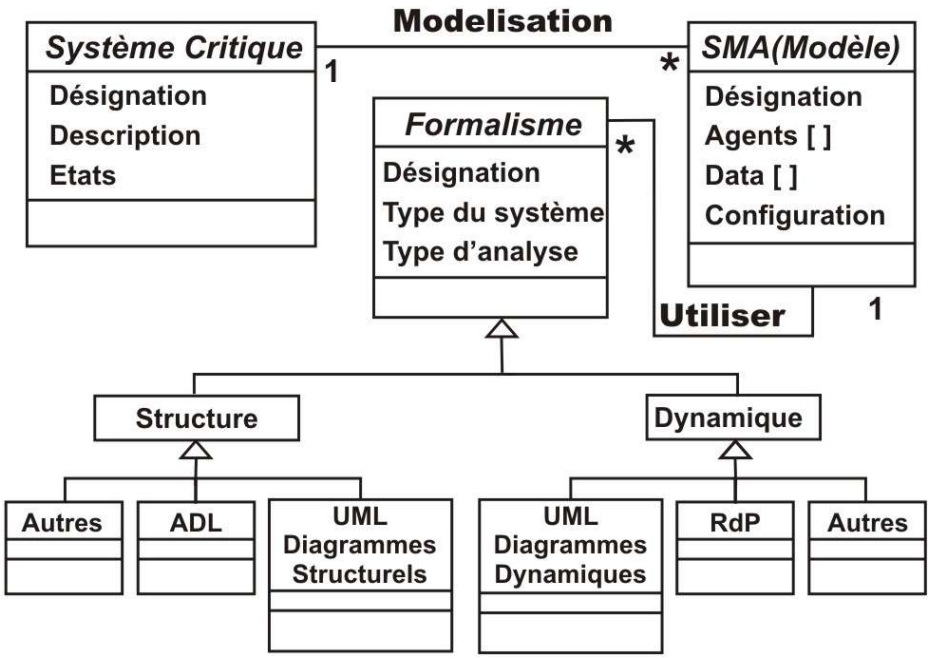

Figure 4.1: La phase de modélisation

Une fois que les éléments architecturaux pertinents (composants et relations entre eux) ont été identifiés, le modèle d'architecture informel de chaque propriété importante du système doit être défini. Ensuite, en fonction de la propriété ou de l'aspect à vérifier, l'architecture du système doit être formalisée à l'aide du formalisme le plus approprié (langage de description d'architecture (ADL), réseaux de Petri, UML, ... etc). Cette étape peut être répétée plusieurs fois jusqu'à ce que toutes les actions intéressantes du système soient bien représentées. A la fin de cette phase, un modèle bien défini pour chaque propriété d'intéressante du système est créé, comme illustré dans la Figure 4.1. Même si qu'on préfère l'utilisation des réseaux de Petri dans cette phase, il est au même temps judicieux de considérer plusieurs modèles d'un même système s'il est nécessaire, afin de bénificier des avantages de chacun d'eux et de vérifier un nombre important de ses propriétés (plus de motivation peut être trouvé dans la section 3.3 du chapitre 3).

4.2 La phase de spécification

Au cours de cette seconde phase, les descriptions d'architecture logicielle — le modèle correspondant à chaque propriété importante du système — obtenues lors de la phase précédente sont soumises à une spécification basée sur la logique de réécriture. Cette traduction est effectuée en utilisant les règles de transformation (mapping) du formalisme utilisé vers la logique de réécriture comme illustré dans la Figure 4.2.

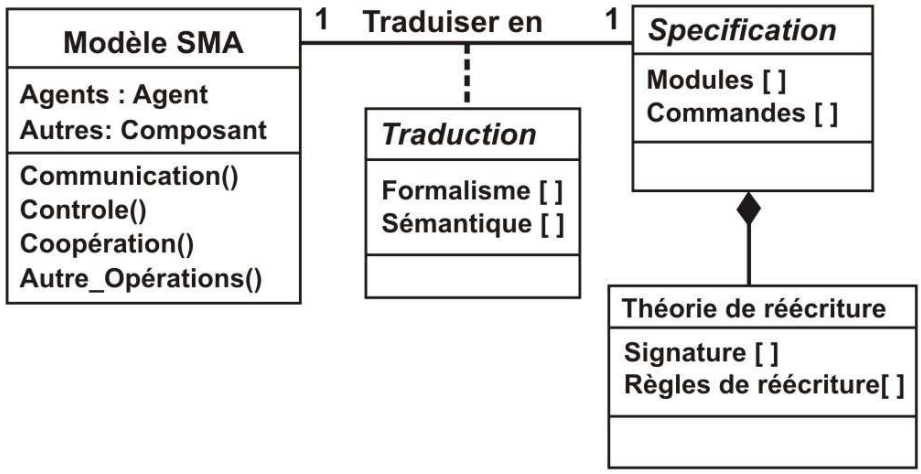

Figure 4.2: La phase de spécification

Afin de faciliter cette phase, le concepteur peut trouver des détails sur ce type de traduction dans [10, 249, 272].
En effet, notre choix de préférer l'utilisation des réseaux de Petri dans la phase de modélisation nous apporte l'avantage de l'automatisation de cette phase de spécification en utilisation l'algorithme et le prototype donnés dans ce papier [2]. D'un point de vue purement technique (programmation), le fait d'avoir choisi d'utiliser la logique de réécriture est très avantageux car cela offre aux concepteurs des techniques facilitant la représentation de l'état mental de l'agent et de rajouter de nouvelles règles de réécriture pour faire évoluer l'agent et le SMA, en bénéficiant des caractéristiques des langages de programmation logique. En plus, Maude supporte l'importation de modules et les techniques de programmation paramétrées de OBJ [32] et comme les spécifications doivent être structurées en modules, une architecture complète (ou un composant) peut être affectée à l'interface d'une autre architecture en tant que paramètre. Cette caractéristique importante permet au concepteur de représenter facilement des architectures composées de manière hiérarchique, facilitant la réutilisation et la construction de nouvelles spécifications à partir de d'autres existantes [34].

4.3 La phase de vérification

Étant donné que nous mettons l'accent sur la conception de systèmes critiques, l'intégration d'une phase de vérification est evidemment essentielle à notre approche afin de garantir la correction des propriétés de sécurité souhaitées du système. Cette troisième phase peut aider le concepteur à déterminer que le système avec ses composants doivent être construits correctement, en guidant dans la validation que l'architecture répond aux exigences définies.
Dans notre approche, la vérification sera effectuée à l'aide de l'outil model-checker de Maude. Cet outil est généralement basé sur l'utilisation de deux niveaux de spécification :

1. *Spécification du système* : à ce niveau, le comportement du système est décrit à l'aide d'une théorie de réécriture. En affet, cette spécification est le résultat de la phase précédente de notre approche.
2. *Spécification des propriétés* : à ce niveau, on doit vérifier l'ensemble des propriétés souhaitées (fonctions) du système ainsi que d'autres propriétés liées au formalisme utilisé. En plus, et vu qu'on utilise le model-checker durant cette phase, il est nécessaire de définir un ensemble des propriétés sur le modèle (en utilisant sa théorie de réécrit re), qui vont être utilisées comme référence dans le processus de vérification.

Le résultat du processus de vérification peut aboutir à un résultat positif, ce qui signifie que le modèle du système répond aux exigences et propriétés souhaitées, ou à un contre-exemple, signalant les incohérences trouvées. Un contre-exemple est une trace d'exécution qui amène le modèle d'état fini de son état initial à un état dans lequel la violation — propriété incorrecte ou un état non autorisé est atteint — se produit.

Comme présenté dans la Figure 4.3, notre approche supporte l'utilisation de l'outil d'atteignabilité (accessibilité) de Maude "***Search Tool***" pour vérifier l'existence/absence des états critiques dans le système. L'ensemble de ces états critiques est généralement définie par les experts du domaine d'étude. Son avantage principal est de montrer tous les chemins possibles pour atteindre un état donné et offre un temps de vérification très rapide si le meilleur état initial est utilisé.
Finalement, et dans le cas de systèmes à états infinis non décidables, Maude supporte la technique de vérification "modèle checking" avec une bibliothèque de stratégies de recherche. De plus, en raison de la nature réflexive de la logique de réécriture et de son implémentation, une telle bibliothèque peut être facilement étendue par l'utilisateur avec de nouvelles stratégies de vérification [297, 298].

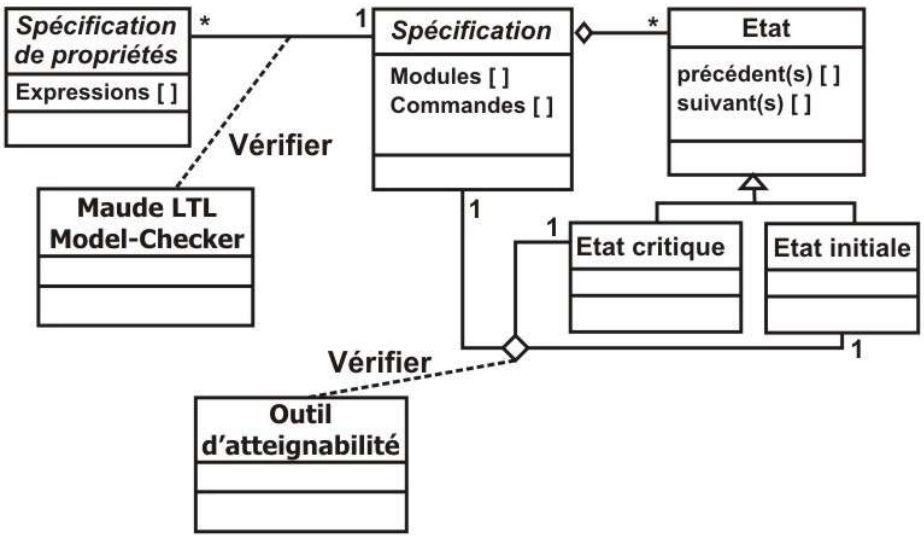

Figure 4.3: La phase de Vérification

4.4 La phase de test

Contrairement à la vérification formelle, les tests n'offrent aucune garantie absolue. Les tests sur des cas unitaires ou des scénarios laissent souvent des bogues cachés qui finissent par apparaître lors du fonctionnement du système. Naturellement, le glissement des défaillances est le pire des scénarios pour les systèmes critiques et nous devons faire mieux. C'est pourquoi nous proposons d'utiliser le test basé sur propriété (PBT) comme stratégie dans la phase de test (voir Figure 4.4).

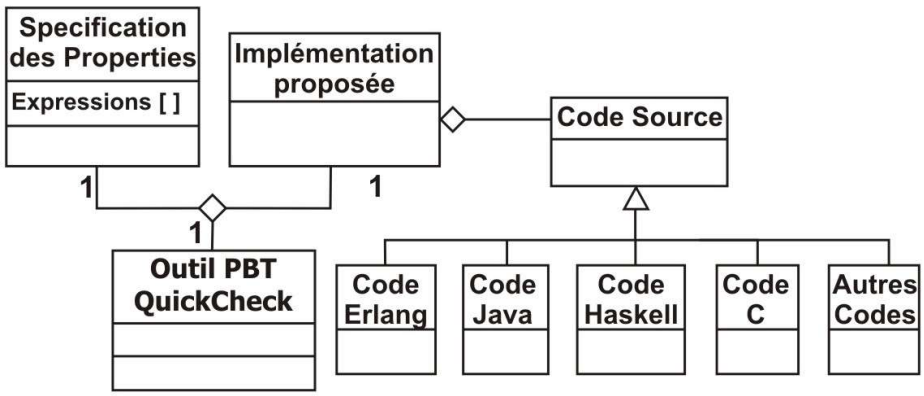

Figure 4.4: La phase de test

En suivant cette approche, nous définissons toutes les propriétés souhaitées qui n'ont pas été vérifiées lors des phases précédentes et celles qui sont liées au code proposé pour l'implémentation du système. Puis, nous consacrons autant de temps que possible pour

générer, exécuter et évaluer automatiquement l'ensemble nécessaire des cas de test. Nous allons profiter des capacités de QuickCheck de Erlang pour produire des contre-exemples lorsqu'un test spécifique est trouvé pour montrer comment la réalisation du système viole une propriété donnée.

Enfin, nous donnons un aperçu de toutes les étapes de notre approche dans la Figure 4.5.

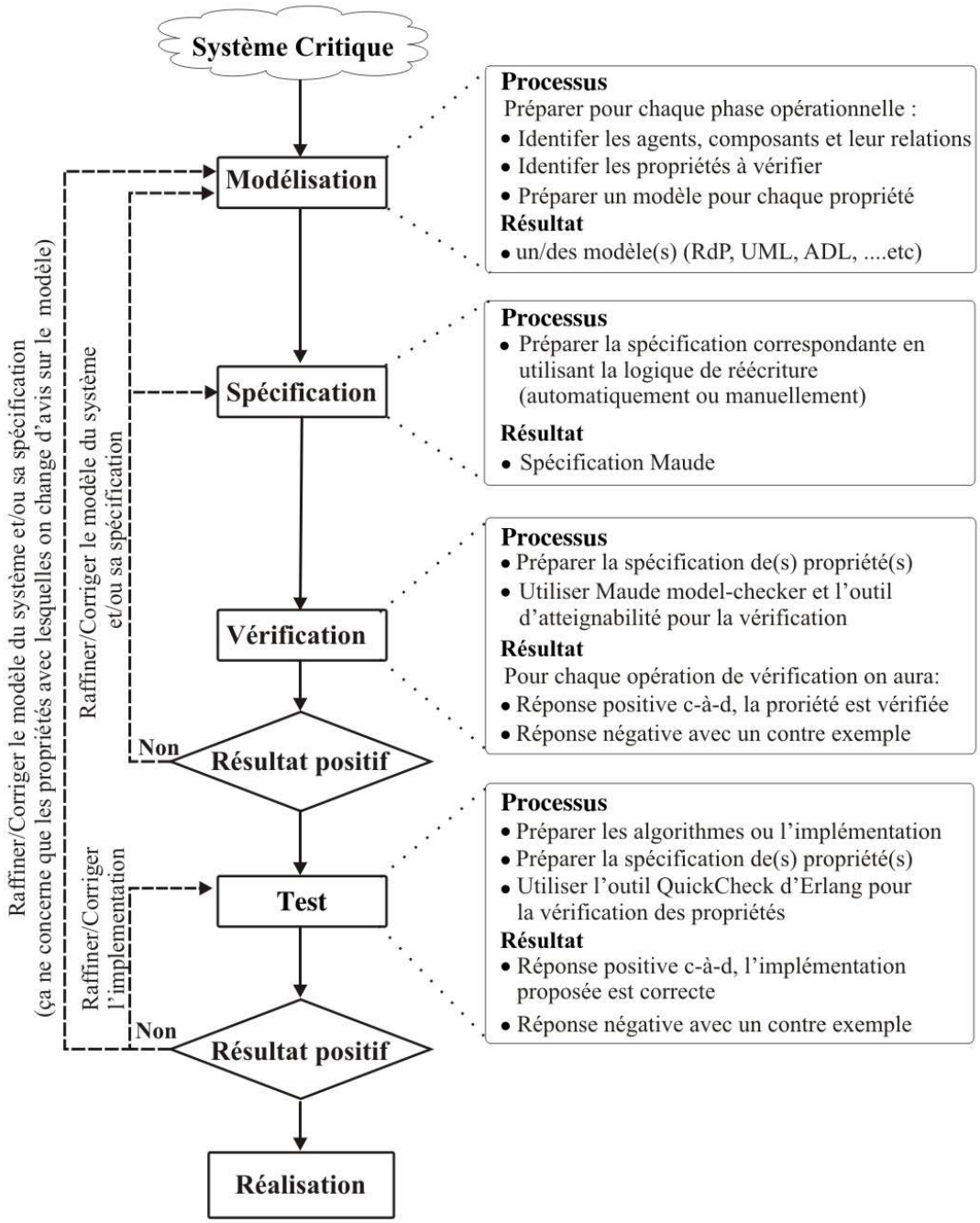

Figure 4.5: La description générale de l'approche proposée

5 Conclusion

Nous avons présenté dans ce chaiptre une nouvelle approche pour l'anlyse formelle des architectures logicielles comportementale combinant la technique de vérification "modèle-checking" et la technique de test basé sur les propriétés (PBT) afin d'assurer un niveau de confiance suffisant surtout pour les systèmes critiques.
Les objectifs ultimes de notre approche étaient :

- Utilisation du formalisme de réseau de Petri — en première place — dans la phase de modélisiation pour préparer un modèle approprié de l'architecture du système à développer. Ensuite, la logique de réécriture est utilisée dans la phase de spécification en profitant de la sémantique améliorée pour les réseaux de Petri, présentée dans la section 3.
- Combinaison de technique de vérification "modèle-checking" et PBT pour garantir l'absence d'erreur ou de comportement inattendu dans le modèle et l'implémentation proposés du système en cours de developpement.

Notre objectif final était de fournir une approche conviviale pouvant aider les concepteurs à définir et générer automatiquement les spécifications du système. Puis, vérifier et tester les modèles d'architecture du système et leurs implémentations. La méthode proposée a été testée sur une étude de cas et ensuite plus dans le papier [1].
À l'avenir, nous envisageons de travailler à la traduction automatique d'autres formalismes — afin élargir l'espace d'utilisation de notre approche — et aussi d'autres extensions des réseaux et/ou de leur proposer une sémantique orientée objet basée sur la logique de réécriture.

CHAPITRE 5

Algorithme de génération automatique des spécifications Maude des SMAs

Ce chapitre introduit l'algorithme proposé pour la génération automatique des spécifications Maude à partir des modèles des SMAs basés sur les réseaux de Petri.

Sommaire

1 Introduction

C'est bien évident qu'au cours des deux dernières décennies, la modélisation à base d'agents est devenue l'une des approches prometteuse qui a conduit à l'étude d'un large éventail de problèmes dans différents domaines [299–303]. Cependant, cette potentialité de modélisation des SMAs s'est également accompagnée de défis et de difficultés considérables lors de leur conception [88, 304–306] en raison de la complexité croissante des systèmes modernes. En outre, nous devons garder à l'esprit que la grande part de responsabilité dans le processus de conception risque de tomber - avec un pourcentage considérable - à la phase de modélisation, car nous ne pourrons jamais obtenir un système répondant aux exigences requises à partir d'un modèle défectueux. Il est donc impératif de rechercher des techniques de modélisation et des formalismes efficaces permettant de gérer ces difficultés. Par conséquent, plusieurs travaux ont été proposés pour la spécification formelle des systèmes multi-agents (MAS), qui tendent à décrire le comportement des agents à l'aide des diagrammes UML, des modèles de réseaux de Petri, des règles logiques et des expressions formelles, ... etc.
Dans notre cas, une spécification algèbrique basée sur la logique de réécriture sera par la suite préparée pour le modèle de réseau de Petri avant de pouvoir utiliser les outils de vérification associés au système Maude. Néansmoins, la préparation manuelle de cette spécification demande un temps considérable — surtout dans le cas de réseau de Petri de grand taille — et peut être sujette à des erreurs. Par conséquent, l'automatisation de cette phase dans le domaine des systèmes critiques sera très avantageuse car elle va nous permettre de minimiser le taux d'erreurs dans la spécification, réduire le temps de développement et de faciliter (accélérer) la vérification de la correction de la spécification générée par rapport au modèle de réseau de Petri. Dans ce chapitre, nous allons présenter l'algorithme proposé pour la génération automatique de la spécification Maude correspondante à un modèle de réseau de petri.

2 Motivation et contribution

Nous allons présenter dans cette section quelques travaux précédents pour montrer brièvement l'efficacité, puis motiver l'utilité des réseaux de Petri et de la logique de réécriture dans le contexte des SMAs.

2.1 Les réseaux de Petri et SMA

Les réseaux de Petri sont toujours considéré comme l'un des formalismes les plus utilisés pour aider les concepteurs aux premiers phases du développement et de simulation des systèmes multi-agents [307]. Par exemple, les réseaux de Petri et plusieures de ses extensions ont été largement utilisés pour la modélisation des SMAs, décrire le comportement des agents et d'autres aspects dans les SMAs [159, 168, 169, 172]. En plus, les réseaux de Petri ont été largement privilégiés pour l'analyse des SMAs en raison de la définition mathématique exacte de leur sémantique d'exécution, la théorie sous-jacente développée autour d'eux ainsi que les méthodes de simulation et d'analyse permettant de vérifier et d'analyser de leur propriétés. Par conséquent, les modèles de réseaux de Petri correspondants des SMAs seront bien évalués.[162, 165, 287, 308]. Par exemple, les modèles de réseau de Petri sont générés à partir des diagrammes UML multi-agents afin de simuler le comportement d'un agent. Par la suite, les techniques d'analyse du modèle de réseau de Petri sont appliquées aux tests de comportement des agents [167].

2.2 Le système Maude

La logique de réécriture et Maude ont été préconisé dans plusieurs travaux pour la spécification et la vérification des SMAs. Le système Maude se caractérise principalement par sa force expressive, ainsi que par le grand nombre d'outils d'analyse formels associés. Par exemple, Riemsdijk et al. citevan2006prototyping présente une implémentation basée sur Maude pour une version simplifiée du langage de programmation d'agent cognitif 3APL. Dans une telle implémentation, les configurations 3APL sont représentées sous forme de termes et les règles de transition sont mappées en règles de réécriture. Dans la version améliorée [181], Maude est utilisé pour le langage de programmation d'agent de prototypage BUpL (Belief Update programming Language) et pour l'exécution des programmes d'agent. Ensuite, l'outil LTL-model checker et les outils d'analyse du système Maude sont également utilisés pour la vérification et le test. Les auteurs de [182] ont utilisé Maude pour obtenir une description formelle des modèles d'organisation basés sur Agent-Group-Role (AGR), qui sont par la suite simulés et vérifiés à l'aide de ses outils du système Maude.
Enfin, le langage Maude est utilisé pour formaliser le comportement du modèle [183] de l'agent DIMA. Dans ce travail, les aspects inhérents aux modèles DIMA sont capturés, puis les descriptions correspondantes de Maude ont été validées à l'aide des outils d'analyse de Maude. Ce travail a été étendu pour prendre en charge à la

fois la description formelle et la vérification des modèles multi-agents DIMA [184]. Cependant, l'inconvénient principal commun est que la transformation des modèles multi-agents en spécification Maude est préparée manuellement de manière ad hoc. Il peut donc être très difficile de s'assurer que cette spécification est une description fidèle du modèle d'agents proposé.

2.3 Synthèse

D'après ce que nous avons exposé, il s'avère évident que l'utilisation du formalisme de réseau de Petri dans la modélisation des SMA est très avantageuse. Ce formalisme permet de visualiser le parallélisme, la synchronisation, le partage de ressources et les conflits dans des systèmes distribués. De plus, il permet de modéliser le comportement (interne et externe) des agents, la coopération et l'interactions entre les agents. Néanmoins, les pratiquants des réseaux de Petri doivent utiliser des techniques de vérification formelles qui identifient et atténuent les risques dans les modèles proposés. D'autre part, il paraît aussi que le système Maude est un cadre formel pour une large gamme de réseaux de Petri ; qui offre des outils d'analyse avancés permettant la vérification automatique des propriétés des modèles basés sur les réseaux de Petri. En particulier, le LTL model-checker est un outil puissant qui est basé sur la technique de vérification à la volée. De plus, Maude permet d'utiliser une autre technique complémentaire, c'est-à-dire un outil d'atteignabilité (accessibilité) permettant de vérifier explicitement l'existence/l'absence d'états critiques dans le système étudié à partir d'un état initial donné. En effet, cet outil peut être utilisé pour voir tous les chemins possibles pour atteindre un état et offre un temps de vérification très rapide si le meilleur état initial est utilisé.
Par conséquent, nous nous sommes concentrés dans ce chapitre sur la présentation d'un algorithme de traduction automatique d'un modèle basé sur les réseaux de Petri à sa spécifications Maude. Cet algorithme permet d'éviter les erreurs humaines et diminuer le temps nécessaire pour la préparation des spécifications Maude par rapport à la traduction manuelle ; surtout dans le cas de modèles des réseaux de Petri volumineux et complexes. Enfin, Un petit outil implémentant notre algorithme a été aussi développé et peut être téléchargé et utilisé librement [i].

i. Disponible à : https ://drive.google.com/open ?id=12PmlZHrJfhniEseM-uBRuFU3PGisgxPm

3 Travaux en rapport

Un certain nombre de travaux et d'outils ont été proposés pour la génération automatique de code à partir de réseaux de Petri, tels que C [309], C ++ [310], Java [311, 312], VHDL [313] et PROMELA [314]. Cependant, notre intérêt pour ce papier réside exclusivement dans ceux liés à Maude.

Bien que la spécification formelle des réseaux de Petri dans la logique de réécriture remonte à il y a quelques décennies, [10, 11], seul un petit nombre d'œuvres pour la génération automatique ont été publiées. Par exemple, le premier outil basé sur la logique de réécriture est celui présenté dans [315] pour l'édition, la simulation et l'analyse de réseaux de termes algébriques simultanés étendus (ECATNets) via une collection quasi complète d'outils d'analyse formelle. La seconde est l'approche proposée dans [316] qui combine métamodélisation et grammaires graphiques pour générer automatiquement la spécification Maude afin de simuler et d'analyser les modèles ECATNets via ATOM3 (A Tool for Multi-formalism and Meta-Modelling). Dans ce travail, un diagramme de classes UML (Unified Modeling Language) est utilisé pour définir un méta-modèle d'ECATNets. Ensuite, une grammaire graphique est proposée pour générer la spécification Maude des modèles ECATNets spécifiés pour effectuer la simulation. Enfin, et à base de notre connaissance, le travail le plus étroitement lié à notre présent travail est celui présenté dans [317], où les auteurs proposent un outil graphique permettant une traduction bidirectionnelle des réseaux de Petri colorés vers Maude et inversement.

Néanmoins, il est à noter que tous les outils précités ont été présentés sous forme de boîte noire sans outil disponible concrétisant leurs approches, ni d'algorithme simple illustrant le fonctionnement de la génération de spécification Maude. En outre, même si le présent document ne peut être comparé strictement aux travaux précités, car ils ne poursuivent pas le même objectif, le principal avantage — et donc la contribution — de notre travail est qu'il est pleinement automatique et basé sur une présentation algébrique bien définie des réseaux de Petri (matrices d'incidence) qui fournit une base théorique à leur simulation et à leur analyse formelle.

4 Algorithme de génération automatique

L'algorithme proposé est principalement basé sur la méthode de spécification donnée dans [10, 11] et sa description générale est illustrée par la figure suivante Figure 5.1.

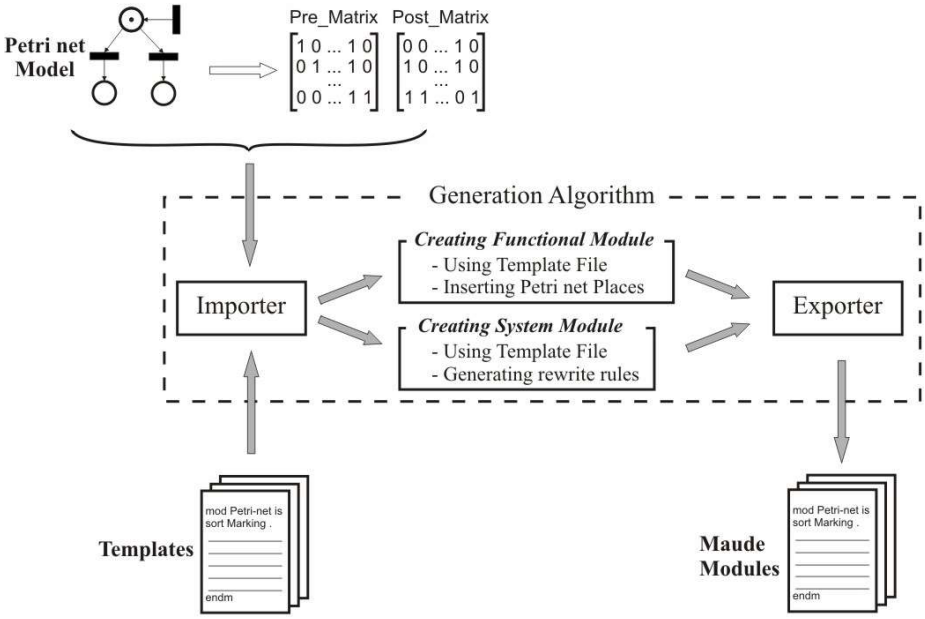

Figure 5.1: Description générale de l'algorithme de génération automatique

4.1 Entrées de l'algorithme

L'algorithme proposé repose principalement sur l'utilisation de la matrice d'incidence et des modèles types (gabarits) pour les modules fonctionnels et systèmes.

– **Matrice d'incidence** :
La représentation algébrique d'un réseau de Petri $\mathcal{PN}$ est définie par la matrice d'incidence composée des matrices Pré-incidence PRE et Post-incidence $POST$ définies par N lignes (nombre de places).) et M colonnes (nombre de transitions). Ces deux matrices peuvent être directement obtenues du réseau de Petri en fonction des fonctions pré-incidence et post-incidence. Par conséquent, la matrice PRE pourrait être remplis à partir du réseau de Petri $\mathcal{PN}$ comme indiqué dans Listing 5.1.

Input: Petri Net : N places, M transitions
1: **procedure** FILL_PRE_MATRIX(PRE : Matrix of N × M of integer)
2: **Ouput : PRE Matrix** of N × M of integer
3: **for** $i \leftarrow 1$ *to* N **do**
4: **for** $j \leftarrow 1$ *to* M **do**
5: **if** (an arc leads from place i to transition j) **then**
6: **PRE**[i,j] $\leftarrow Pre(P_i, T_j)$
7: **else**
8: **PRE**[i,j] $\leftarrow 0$
9: **end if**
10: **end for**
11: **end for**
12: **end procedure**

Listing 5.1: Pseudo code pour remplir la matrice de pre-incidence (PRE)

De même, la procédure FILL_POST_MATRIX est définie dans le Listing 5.2.

Input: Petri Net : N places, M transitions

```
1: procedure FILL_POST_MATRIX(POST : Matrix of N × M of integer)
2:     Ouput : POST Matrix of N × M of integer
3:     for i ← 1  to  N do
4:         for j ← 1  to  M do
5:             if (an arc leads from transition j to place i) then
6:                 POST[i,j] ← Post(P_i, T_j)
7:             else
8:                 POST[i,j] ← 0
9:             end if
10:        end for
11:    end for
12: end procedure
```

Listing 5.2: Pseudo code pour remplir la matrice de post-incidence (POST)

– **Modèles-type (Gabarit)** :

Il est clair que certaines parties dans les modules de spécifications Maude — pour les réseau de Petri — ne sont pas modifiables et que seuls les noms des places (resp. L'ensemble des règles de réécriture) doivent être modifiés dans le module fonctionnel (resp, le module système) pour chaque nouveau réseau de Petri. Par conséquent, l'utilisation d'un fichier de modèle-type de spécification contenant les éléments de spécification de constante sera très utile pour un algorithme de génération automatique. Ces modèles-type de spécification FTemplate (FUNC_PETRINET) et STemplate (SYSTEM_PETRINET) sont donnés dans le Listing 5.3.

```
1. fmod FUNC_PETRINET is
2.   sorts Place Marking .
3.   subsorts Place < Marking .
4.   op empty : -> Marking .
5.   ops PLACES_NAMES  : -> Place .
6.   op __ : Marking Marking -> Marking
               [assoc comm id: empty] .
7. endfm
```

```
1. mod SYSTEM_PETRINET is
2.   including FUNC_PETRINET .
3.    ... Other Statements ...
4.    rl [Trans-Name] :  lhs => rhs .
      ... Other Rules ...
5. endm
```

Listing 5.3: Modèle-type des modules de spécification

4.2 Eléments de l'algorithme

L'algorithme proposé comprend deux parties. La première concerne la génération du module fonctionnel, tandis que la seconde concerne la génération du module système.

• ***Création des modules de spécification*** :
La procédure suivante décrit le processus de création des modules de spécification à l'aide des fichiers de modèles-type (Listing 5.4). En fait, cette procédure copiera le contenu du fichier de modèle-type dans le module de spécification à créer.

Input: Template file
1: **procedure** CREATE_SPECIFICATION_FILE(Specificationfilename : String, Templatefilename : String)
2: **Ouput : Specification file** : Maude file
3: $Open(Templatefilename)$
4: $Createemptyfile(Specificationfilename)$
5: $Copyfile(Templatefilename, Specificationfilename)$ ▷ copy "Template file" to "Specification file"
6: $Close(Specificationfilename)$
7: $Close(Templatefilename)$
8: **end procedure**

Listing 5.4: La procédure de création des fichiers de spécification

• ***La génération des modules de spécification*** :
Afin de faciliter le processus de génération de la spécification Maude, nous supposons que les lieux sont étiquetés comme : P1, P2, ... Pn et, transitions comme : T1, T2, ... Tm. De plus, nous utiliserons principalement deux procédures : `PLACE_NAMES` et `REWRITE_RULES`.
La première procédure extrait (relève) le nombre de places de la matrice pré-incidence PRE afin de générer la ligne de spécification correspondante dans une variable chaîne de caractère `"placenames"` qui sera insérée à la place appropriée dans la module fonctionnel comme indiqué dans Listing 5.5.

Input: N : rows number of PRE Matrix
1: **procedure** PLACE_NAMES(placenames : String)
2: **Ouput : placenames** : String ▷ this will replace line 5 in functional module template
3: **placenames** ← "ops" + " "
4: **for** $i \leftarrow 1$ *to* N **do**
5: **placenames** ← **placenames** + "P" + Asstring(i) + " " ▷ Asstring(i) : converts i into string
6: **end for**
7: **placenames** ← **placenames** + " : − > Place ."
8: **end procedure**

Listing 5.5: Pseudo code generating the place names for the functional module

La fonction suivante IS_SOURCE(resp, IS_SINK) est une fonction intermédiaire qui détermine les transitions sources (resp, puits) dans le réseau de Petri. Elle est donnée dans Listing 5.6 (resp, Listing 5.7).

En fait, ces deux fonctions sont nécessaires car leurs règles de réécriture correspondantes seront préparées avec un traitement spécial.

```
Input: PRE Matrix
 1: function Is_Source(x : integer)                    ▷ x : order of transition in PRE
 2:     Ouput : result : boolean
 3:     result ← True
 4:     i ← 1
 5:     while (i <= N)and(result) do
 6:         if (PRE[i,x] > 0) then
 7:             result ← False                          ▷ The transition x : is not a source
 8:         end if
 9:         i ← i + 1
10:     end while
11:     Is_Source ← result
12: end function
```

Listing 5.6: Pseudo code de vérification des transitions sources

De même, la fonction suivante Is_Sink est donnée comme suit :

```
Input: POST Matrix
 1: function Is_Sink(x : integer)                      ▷ x : order of transition in POST
 2:     Ouput : result : boolean
 3:     result ← True
 4:     i ← 1
 5:     while (i <= N)and(result) do
 6:         if (POST[i,x] > 0) then
 7:             result ← False                          ▷ t The transition x : is not sink
 8:         end if
 9:         i ← i + 1
10:     end while
11:     Is_Sink ← result
12: end function
```

Listing 5.7: Pseudo code de vérification des transitions puits

La seconde procédure génère la règle de réécriture correspondante pour chaque transition dans la matrice d'incidence IM, puis enregistre le résultat dans une variable de type chaîne de caractère `"RewriteRule"`. La valeur de cette chaîne de caractère sera insérée dans le module système.
Certainement, cette procédure sera répétée (appelée) pour chaque transition dans le réseau de Petri. Le pseudo code de cette procédure est illustrée dans Listing 5.8.

Input: Matrix PRE,POST

```
1: procedure REWRITE_RULE(x : integer, RewriteRule : String)  ▷ x : represents row/transition number
2:     Ouput : RewriteRule : String
3:     Variables : Label, lhs, rhs, arrow : String
4:     arrow ← "=>" + "  "
5:     Label ← "T" + Asstring(x)                              ▷ to generate label for each transition
6:     RewriteRule ← "rl" + "  " + "[" + label + "]" + "  " + " :" + "  " ▷ prepare the rewrite rule
7:     lhs ← ""                                               ▷ initialization of lhs of each rule to empty
8:     rhs ← ""                                               ▷ rhs is also initialized to empty for each rule
9:     for i ← 1  to  N do
10:        if (PRE[i, x] > 0) then
11:            for j ← 1  to  PRE[i, x] do
12:                lhs ← lhs + "P" + Asstring(i) + "  "       ▷ preparing lhs
13:            end for
14:        end if
15:        if (POST[i, x] > 0) then
16:            for j ← 1  to  (POST[i, x]) do
17:                rhs ← rhs + "P" + Asstring(i) + "  "       ▷ preparing rhs
18:            end for
19:        end if
20:    end for
21:    if (Is_Sink(x)orIs_Source(x)) then
22:        lhs ← "M" + "  " + lhs                            ▷ M is a variable of sort : Marking
23:        rhs ← "M" + "  " + rhs
24:    end if
25:    RewriteRule ← RewriteRule + lhs + arrow + rhs + "."
26:                                  ▷ RewriteRule will be written at line 3 of system_module
27: end procedure
```

Listing 5.8: Pseudo code de génération de règle de réécriture pour chaque transition

De plus, il nous paraît qu'on a besoin d'une autre procédure complémentaire (voir Listing 5.9) pour écrire **placenames** et **RewriteRule** dans les fichiers de spécification. Pour une telle procédure, nous devons donner le nom du fichier de spécification et le numéro de ligne où les chaînes de caractère **placenames** ou **RewriteRule** doivent être écrits.

Input: Specification file

```
1: procedure INSERT(Specificationfilename : String, Linetoinsert : String, linenumber : integer)
2:     Open(Specificationfilename)
3:     writenewline(Linetoinsert, linenumber)
4:     ▷ write "Linetoinsert" at the line : (linenumber).
5:     Close(Specificationfilename)
6: end procedure
```

Listing 5.9: Procedure d'insertion du code dans le fichier de specification

4.3 l'Algorithme complet

L'algorithme complet proposé pour la génération automatique de la spécification Maude à partir du modèle de réseau de Petri est donné comme suit :

Algorithm 1 Automatic Generation of Maude Specification

```
Input: Petri Net : N places, M transitions
       FTemplate : Functional template file
       STemplate : System template file
Output: Specification Files : Functional module, System Module
   Begin
 1: S : integer
 2: S ← 0                                   ▷ M : is equal 0 when there is (are) not sink or source transition(s)
 3: Sink_or_Source : boolean
 4: Sink_or_Source ← False
 5: Fill_PRE_Matrix(PRE)
 6: Fill_POST_Matrix(POST)
 7: Create_specification_file(functional_module, FTemplate)
 8: Place_Names(placenames)
 9: Insert(functional_module, placenames, 5)
10: Create_specification_file(system_module, STemplate)
11: j ← 1
12: while (j <= M)and not(Sink_or_Source) do        ▷ Test the existence of source or sink transitions
13:     if (Is_Sink(j) or (Is_Source(j)) then
14:         Sink_or_Source ← True
15:     end if
16:     j ← j + 1
17: end while
18: if (Sink_or_Source) then
19:     Insert(system_module, "op M :-> Marking .", 3 + S)  ▷ Add the variable M declaration to the specification
20:     S ← S + 1                              ▷ Incrementing S to insert the next rule in the following line
21: end if
22: for j ← 1 to M do
23:     Rewrite_Rule(j, RewriteRule)
24:     Insert(system_module, RewriteRule, 3 + S)
25:     S ← S + 1                              ▷ Incrementing S to insert the next rule in the following line
26: end for
   End
```

Listing 5.10: Pseudo code de l'algorithme proposé

5 Conclusion

Dans ce chapitre, nous avons présenté un algorithme bien défini pour la génération automatique de spécifications Maude à partir d'un modèle de réseaux de Petri, basé sur leur représentation algébrique. Cet algorithme (prototype) sera très utile aux pratiquants qui construisent des modèles des réseaux de Petri et aux concepteurs même peu expérimentés avec la logique réécriture pour gagner du temps, éviter les erreurs humaines lors du processus de traduction généralement manuel et leur donner la possibilité d'exploiter tous les outils d'analyse avancés de Maude pour leurs modèles. Nos travaux ultérieurs consistent à créer un outil complet ou un plug-in d'Eclipse implémentant cet algorithme et fournissant une interface utilisateur graphique

conviviale permettant aux utilisateurs finaux d'éditer le réseau de Petri, puis de générer les matrices d'incidence et la spécification Maude à partir du modèle de réseaux de Petri. De plus, des recherches sont en cours pour élargir l'applicabilité de notre algorithme à d'autres catégories de réseaux de Petri en utilisant les structures de données les plus appropriées. Enfin, on doit noter que ce chapitre fait l'objet du papier [2] qui est en cours de révision.

CHAPITRE 6

Nouvelle sémantique pour les réseaux de Petri

Ce chapitre introduit la sémantique améliorée des réseaux de Petri basée sur la logique de réécriture.

Sommaire

1 Introduction

D'une part, les réseaux de Petri se caractérisent par de nombreuses caractéristiques distinctives en tant que paradigme de modélisation et d'analyse. Celles-ci ont permi aux réseaux de Petri d'être un formalisme prometteur utilisé dans de très nombreux domaines. On cite à titre d'exemple les problèmes de partage des ressources, la synchronisation des tâches, les protocoles de transmission et la gestion de la production [318]. En même temps, il existe une demande croissante d'utilisation d'outils formels conviviaux [319–321] et de techniques d'analyse [322, 323] pour vérifier et valider le comportement de modèles complexes basés sur un réseau de Petri.
D'autre part, Maude [20] est un langage de programmation exécutable qui a été adopté dans de nombreux travaux récents pour spécifier, exécuter et analyser des réseaux de Petri [272, 324–327], grâce à l'expressivité de sa logique [11] et ses outils d'analyse associés. Cependant, la sémantique existante basée sur la logique de réécriture pour les réseaux de Petri [11], généralisée pour une large gamme de dérivés de réseaux de Petri dans [10], présente certaines lacunes, notamment la difficulté à manipuler la multiplicité des arcs, nombre de jetons dans les places en plus de l'ambiguïté sémantique entre les places et leurs jetons. Ce dernier donne l'impression que le changement de marquage d'un réseau de Petri est un changement sur les jetons plutôt qu'un changement d'états des places contenant ces jetons. De plus, la sémantique existante rend également la représentation des arcs avec des poids variables, la priorité ou le test si un RdP est borné ou non si difficile, voire impossible, même avec quelques équations supplémentaires et des modifications radicales.
Dans ce chapitre, nous proposons une sémantique améliorée du réseau de Petri qui distingue clairement les places de leurs jetons afin de surmonter naturellement les limites susmentionnées, puis de présenter ses avantages par rapport à la sémantique existante.

2 Sémantique existante des réseaux de Petri

La sémantique existante basée sur la logique de réécriture pour les réseaux de Petri standard a été proposée dans [11] et a ensuite été généralisée pour un large gamme de réseaux de Petri dans [10].
Pour illustrer cette sémantique, considérons le réseau de Petri présenté dans la Figure 6.1(a), décrivant le comportement du distributeur automatique classique [328] utilisé pour acheter des gâteaux et des pommes ; un gâteau coûte 1 dollar et une pomme 3

quarters (pièce de 25 centimes). Pour des fins de simplification, nous supposons que cette machine accepte uniquement les dollars, mais peut convertir quatre quarters en dollars pour outre passer ce problème de conception.

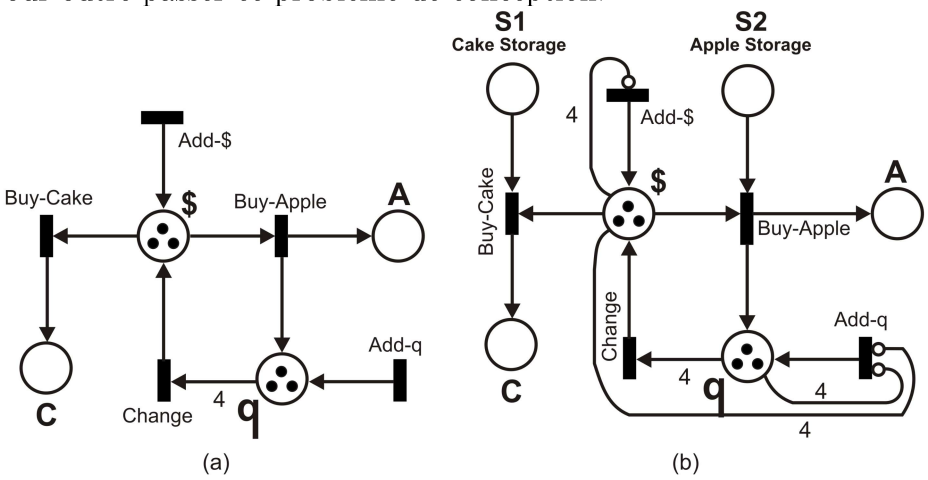

Figure 6.1: Réseaux de Petri décrivant le comportement du distributeur automatique

2.1 Aspects structurels

Les types abstraits de données (appelés dans Maude : `sorts`) nécessaires pour décrire un réseau de Petri sont les suivants : ***place*** et ***marking***. Un marquage sur un réseau de Petri est en fait un multiset sur son ensemble de places, représentant (un instantané de) l'état du réseau de Petri et indiquant les ressources disponibles dans chaque place. Ces types sont définis comme `sort` et des `subsort` comme suit :

```
sorts Place Marking .
subsort Place < Marking .
```

Ensuite, les noms des places du réseau de Petri sont déclarés en tant qu'opérateurs comme suit :

```
ops C A $ q :-> Place .
```

Puis, l'état actuel (marquage) d'un réseau de Petri peut ensuite être défini avec un opérateur d'union multi-ensembles (multiset) fini, comme suit :

```
op _ _ : Marking Marking -> Marking [assoc comm id:null].
op null : -> Marking .
```

Selon cette déclaration, nous pouvons voir qu'un marquage est représenté comme un élément du type `Marking` et que l'union de deux marquages est un nouveau marquage. De plus, l'attribut `"assoc"` (resp, `"comm"`) est utilisé pour déclarer que l'opérateur est associatif (resp, commutatif). Le marquage vide est représenté par la constante `null`.

Enfin, le marquage initial peut également être déclaré en tant qu'opérateur, puis défini par une équation :

```
op initial :-> Marking .
eq initial = $ $ $ q q q .
```

Dans une telle déclaration, seuls les noms des places contenant des jetons apparaissent dans l'état initial avec une occurrence égale au nombre de jetons qu'ils contiennent.

2.2 Aspects comportementaux

L'évolution d'un réseau de Petri est liée aux tirs (franchissements) des transitions. Pour cela, la spécification de l'opération de franchissement d'une transition consiste de deux termes (multisets), où le premier (marquage représentant les ressources qu'il va consommer (pré-condition)) peut être remplacé par le second (marquage représentant les ressources qu'il va produire (postcondition)).
Par conséquent, chaque transition `t` est décrite par une règle de réécriture étiquetée avec la syntaxe suivante :

`rl [`⟨`Transition-Label`⟩`] : ⟨`Termset-1`⟩ `=>` ⟨`Termset-2`⟩ .`

Dans une telle règle, le terme `Termset-1` (resp, `Termset-2`) ne contient que l'ensemble des places d'entrées (resp, sorties) de la transition correspondante, où le nom de chaque place (d'entrée) (resp, sortie) est répété dans la partie gauche (resp, droite) de la règle de réécriture, autant de fois que le poids de l'arc liant la place à la transition. Dans ce cas, à chaque fois le terme `Termset-1` est trouvé dans le marquage il peut être transformé au terme correspondant décrit dans la partie droite de la règle de réécriture. De plus, une transition puit (resp, source) est un cas particulier et sa règle de réécriture correspondante reste la même et une variable (`M`, par exemple) de type `Marking` est ajoutée aux `Termset-2` et `Termset-1`.
Enfin, en suivant les étapes pré-décrites, nous donnons la spécification complète du réseau de Petri, présentée à la Figure 6.1 (a), comme suit :

```
fmod PETRI-NET-SIGNATURE is
  sorts Place Marking .
  subsorts Place < Marking .
  op null : -> Marking .
  ops C A $ q : -> Place .
  op __ : Marking Marking -> Marking
                  [assoc comm id: null] .
  op initial : -> Marking .
  eq initial = $ $ $ q q q .
endfm
```

```
mod VENDING-MACHINE is
protecting PETRI-NET-SIGNATURE .
  var M : Marking .

  rl [add-$] : M => M $ .
  rl [add-q] : M => M q .
  rl [buy-C] : $ => C .
  rl [buy-A] : $ => A q .
  rl [change] : q q q q => $ .
endm
```

Comme nous pouvons le voir, dans cette spécification, la partie statique (la signature) de la machine est donnée dans un module fonctionnel `PETRI-NET-SIGNATURE`. Ce module

a été importé dans le module système `VENDING-MACHINE`, qui ajoute ensuite les règles de réécriture (règle pour chaque transition) afin de compléter la description de la partie dynamique du distributeur automatique. Notant que nous avons ajouté les deux règles `"[add-$]"` et `"[add-q]"` pour décrire les transitions ***sources*** dans le réseau de Petri.

2.3 Limites de cette sémantique

Bien que la sémantique existante est très compacte en termes de la taille du code source produit pour la spécification de réseau de Petri, elle présente également quelques inconvénients majeurs (voir la section 4 pour plus de détails) :

- Il existe une confusion sémantique quant à l'utilisation du type `Place`, qui est censé représenter des places mais il sert à indiquer le jeton d'une place. Par conséquent, la règle de réécriture décrivant le franchissement d'une transition est considérée comme une description de changement de l'état des jetons plutôt que d'une description de de changement de l'état des places dans lesquels les changements se produisent.
- La sémantique existante n'est pas capable de compter naturellement (ou afficher) le nombre de jetons existants dans une place sans équations supplémentaires car elle gère la multiplicité des arcs avec une manière si triviale (répétition des noms des places). Par conéquent, il n'est pas possible de tester naturellement les places de réseau de Petri pour zéro (c'est-à-dire de vérifier si une place P_i a zéro jeton). En plus, le test si un réseau est borné et la spécification des réseaux de Petri avec des arcs inhibiteurs ne sont pas naturellement possible. Enfin, la spécification de réseaux de Petri à poids d'arc variables n'est pas aussi possible avec la sémantique existante.
- La spécification de réseaux de Petri colorés basée sur cette approche comme celle donnée dans [10, 317] hérite également tous les problèmes mentionnés.

3 Sémantique améliorée pour les réseaux de Petri

Dans la sémantique proposée pour les réseaux de Petri, nous nous sommes principalement concentrés à surmonter les inconvénients mentionnés afin de rendre la spécification des réseaux de Petri plus naturelle. Pour illustrer l'idée proposée, nous utilisons le réseau de Petri présenté dans la Figure 6.1(b) pour bien remarquer la différence.

3.1 Aspects structurels

Nous proposons de définir une place de réseau de Petri par son nom (sort : `Placename`) suivi du nombre de jetons qu'il détient (sort : `Int`). Cependant, le type de base `Marking` — qui est formée par l'union multi-ensemble (multiset) — est conservé pour modéliser le comportement dynamique d'un réseau de Petri. Par conséquent, la nouvelle signature correspondante est donnée comme suit :

```
sorts PlaceName Place Marking .
subsort Place < Marking .
ops C A $ q :-> Placename .
op <_,_> : PlaceName Int -> Place [ctor] .
op _ _ : Marking Marking -> Marking [assoc comm id:null].
```

Le premier avantage de la spécification proposée est que le test du nombre de jetons dans une place ainsi que la spécification des arcs inhibiteurs sont maintenant possibles. En outre, la déclaration du marquage initial dans notre proposition décrit le réseau de Petri global, y compris toutes ses places, et ne se limite pas aux places contenant des jetons. Par conséquent, cette déclaration donne un vue (instantané) claire du marquage initial du système et permet aux utilisateurs (observateurs ou developpeur) de connaître tous les noms des lieux ainsi que les jetons qu'ils détiennent directement depuis le marquage initial du réseau de Petri.

```
op initial :-> Marking .
eq initial = < $,3 > < q,3 > < C,0 > < A,0 > .
```

3.2 Aspects comportementaux

Chaque fois un réseau de Petri est marqué avec M, son nouveau marquage M' après le franchissement d'une transition T est définie comme suit :

$$M' = M - Pre(P_i, T) + Post(T, P_j)$$

Plus précisément, le changement dans l'état du réseau de Petri lors du franchissement de la transition T se produit en effet au niveau de l'ensemble des places d'entrées et de sorties en supprimant des jetons du premier ensemble et en ajoutant d'autres au second ensemble. Par conséquent, dans notre proposition, nous énumérons l'ensemble des places de d'entrées et de sorties de la transition T sur les côtés gauche et droit de la règle de réécriture correspondante. En plus, une règle de réécriture décrivant

le franchissement d'une transition peut souvent être conditionnelle pour inclure la condition d'activation d'une telle transition. Par conséquent, elle est donné comme suit :

```
crl [⟨Transition-Label⟩] : ⟨Left-Hand-Side⟩ => ⟨Right-Hand-Side⟩ if Cond .
```

Dans une telle règle, le `Left-Hand-Side` (resp, `Right-Hand-Side`) d'une règle de réécriture décrit l'état des places d'entrées et de sorties avant (resp, après) le franchissement en donnant le nombre de jetons de chaque place. De plus, `Cond`[i] est l'expression qui représente la condition d'activation de la transition.
Par conséquent, une règle de réécriture pourrait être un peu plus grande, mais elle est considérablement plus claire — en termes de la présentation — que celle de la sémantique existante.
Enfin, nous présentons maintenant la spécification complète du réseau de Petri (voir la Figure 6.1 (b)) conformément à la sémantique proposée.

```
fmod NEW-PETRI-NET-SIGNATURE-1 is
protecting INT .
sorts PlaceName Place Marking .
subsort Place < Marking .
ops C A $ q : -> PlaceName .
op <_,_> : PlaceName Int -> Place [ctor] .
op __ : Marking Marking -> Marking [ctor assoc comm id: null] .
op null : -> Marking .
op initial : -> Marking .
eq initial = < $,3 > < q,3 > < C,0 > < A,0 > .
endfm

mod NEW-VENDING-MACHINE-1 is
inc NEW-PETRI-NET-SIGNATURE-1 .
vars x y z : Int .
rl [add-$] : < $,x >  => < $,x + 1 > .
rl [add-q] : < q,x >  => < q,x + 1 > .
crl [buy-c] : < $,x > < C,y > => < $,x - 1 > < C,y + 1 > if (x > 0) .
crl [buy-a] : < $,x > < A,y > < q,z > => < $,x - 1 > < A,y + 1 > < q,z + 1 > if (x > 0) .
crl [change] : < $,x > < q,z > => < $,x + 1 > < q,z - 4 > if (z >= 4) .
endm
```

i. cette expression n'est pas nécessaire dans le cas d'une transition source, car elle est toujours activée et, par conséquent, la règle de réécriture n'est pas conditionnelle

4 Contributions et comparaison

Dans cette section, nous présentons l'efficacité de la sémantique proposée qui est basée sur la logique de réécriture pour les réseaux de Petri. À cette fin, des résultats de comparaison expérimentale ont été présentés afin d'évaluer ses avantages et performances par rapport à la sémantique existante.

4.1 Distinction entre places et jetons

Comme décrit dans la section 2.3, il y a une certaine ambiguïté entre les places et les jetons lorsqu'on utilise le type `"Place"`. Cela se voit clairement dans la déclaration de l'état initial ou même dans les règles de réécriture. Par exemple, considérons la règle de réécriture décrivant la transition `"change"` du réseau de Petri donné dans la Figure 6.1 (a).

```
rl [change] : q q q q => $ .
```

Il est clair que cette règle signifie qu'à chaque fois que l'état complet du système comporte quatre jetons dans la place `q`, ils seront remplacés par un jeton dans la place `$` après le franchissement de la transition. Ainsi, `q` signifie un jeton de la place `q`, mais pas la place `q` elle-même comme indique le déclaration des types.
À l'opposé, la règle de réécriture décrivant la même transition `"change"` dans la sémantique proposée est donnée comme suit :

```
crl [change] : < $,x > < q,z > => < $,x + 1 > < q,z - 4 > if (z >= 4) .
```

Comme on peut le constater dans une telle règle, il existe une différence claire entre une place et ses jetons. De plus, il est clair que le franchissement de cette transition apportera des modifications à deux places ($ and q). Plus précisément, le nombre de jetons (x) à la place $ sera augmenté par un (x+1) et le nombre de jetons (z) à la place q sera réduit par quatre (z-4) pour exprimer le changement de quatre jetons de q (s'il existe) en un dollar.

4.2 Clarté dans l'exploration du marquage

Pour explorer le marquage (comportement) du réseau de Petri à partir du marquage initial donné, nous utilisons la commande `"rewrite"`. Cependant, il est nécessaire de déterminer la limite supérieure autorisée pour le nombre d'applications de règles de réécriture lors de l'utilisation de cette commande. Sinon, l'infini est supposé.

Par exemple, considérons la commande "rew [5]" (resp, "rew [100]") pour voir le marquage obtenu après 5 (resp, 100) fois d'application de règle de réécriture en partant du marquage initial, qui a 3 dollars et 3 quarters.

Existing Semantics	**Proposed Semantics**
Maude> rewrite [5] in VENDING-MACHINE : initial . rewrites: 6 in 541371105387ms cpu (0ms real) (0 rewrites/second) result Marking: $ $ $ $ $ $ q q q	Maude> rewrite [5] in NEW-VENDING-MACHINE-1 : initial . rewrites: 23 in 541371105385ms cpu (0ms real) (0 rewrites/second) result Marking: < q,3 > < $,0 > < C,2 > < A,2 >
Maude> rewrite [100] in VENDING-MACHINE : initial . rewrites: 101 in 541371105435ms cpu (0ms real) (0 rewrites/second) result Marking: $ q	Maude> rewrite [100] in NEW-VENDING-MACHINE-1 : initial . rewrites: 481 in 541371105397ms cpu (0ms real) (0 rewrites/second) result Marking: < q,2 > < $,0 > < C,18 > < A,2 >

Comme on peut le constater, les résultats d'exécution de la commande "rewrite" dans la sémantique existante peuvent devenir impraticables et incompréhensibles, en particulier dans le cas de systèmes comportant un nombre considérable de jetons. Cependant, ce n'est pas le cas pour la sémantique proposée où le nombre de jetons à chaque place est affiché sans qu'il soit nécessaire de les compter manuellement.

4.3 Description des arcs inhibiteurs

Afin d'examiner l'habilité de décrire les arcs inhibiteurs à la fois sur la sémantique existante et sur celle proposée, nous considérons la deuxième machine, présentée dans la Figure 6.1(b), dans laquelle nous impliquons les utilisateurs du distributeur d'insérer plus de 4 dollars (resp, 4 quarters) dans la boîte à monnaie avant d'acheter quelque chose (resp, échanger la monnaie). Cependant, les quarters retournés après l'achat d'une pomme sont ajoutés à la boîte à monnaie sans contrôle.

4.3.1 La sémantique proposée

La représentation des places — dans la sémantique proposée — est basée sur le constructeur :

```
op <_,_> : PlaceName Int -> Place [ctor] .
```

qui permet d'accéder directement au nombre de jetons dans une place donnée du réseau de Petri. Par conséquent, la spécification de tous les types d'arcs (arcs ordinaires et inhibiteurs) seront naturellement possible. En utilisant la sémantique proposée, on on la spécification du distributeur présenté dans la Figure 6.1(b) comme suit :

```
fmod NEW-PETRI-NET-SIGNATURE-2 is
protecting INT .
sorts PlaceName Place Marking .
subsort Place < Marking .
ops C A $ q S1 S2 : -> PlaceName .
op <_,_> : PlaceName Int -> Place [ctor] .
op __ : Marking Marking -> Marking [ctor assoc comm id: null] .
ops null initial : -> Marking .
eq initial = <$,3> <q,3> <C,0> <A,0> <S1,50> <S2,50> .
endfm
mod NEW-VENDING-MACHINE-2 is
inc NEW-PETRI-NET-SIGNATURE-2 .
vars x y z t : Int .
crl [add-$] : < $,x > => < $,x + 1 > if (x < 4) .
crl [add-q] : < $,x > < q,z > => < $,x > < q,z + 1 > if (z < 4) and (x < 4) .
crl [buy-c] : < $,x > < C,y > < S1,z > => < $,x - 1 > < C,y + 1 > < S1,z - 1 >
                                                    if (x >= 1) and (z >= 1) .
crl [buy-a] : < $,x > < A,y > < q,z > < S2,t > => < $,x - 1 > < A,y + 1 > < q,z + 1 >
                                                  < S2,t - 1 > if (x >= 1) and (t >= 1) .
crl [change] : < $,x > < q,z > => < $,x + 1 > < q,z - 4 > if (z >= 4) .
endm
```

Dans cette spécification, les transitions avec les arcs inhibiteurs `add-$` et `add-q` ont été aisément décrites en utilisant des règles de réécriture conditionnelles avec le test nécessaire pour le nombre limite de l'arc.

4.3.2 La sémantique existante

Dans la sémantique existante, le nombre de jetons dans une place ne peut pas être obtenu naturellement et, pour l'obtenir, on peut utiliser un opérateur supplémentaire pour capturer le nombre de jetons comme suit :

```
op {_} : Marking -> PN .
```

englobant la totalité du marquage , on pourrait définir de la même manière un deuxième opérateur, comme suit :

```
op number : PN Place -> Int .
```

qui est simplement définie comme suit :

```
eq number({M M'},M) = 1 + number({M'},M) .
eq number({M'},M) = 0 [owise] .
```

Avec ce nouvel opérateur, on pourrait décrire les transitions avec des arcs inhibiteurs.

Par exemple, les règles `add-$` et `add-q` pourraient maintenant être écrites comme suit :

```
crl [add-$] : {M} => {M $} if number({M},$) < 4 .
crl [add-q] :  {M} => {M q} if number({M},$) < 4 /\ number({M},q) < 4 .
```

4.4 Test de bornitude

Le test de bornitude d'un réseau de Petri ou d'une de ses places est possible avec la sémantique proposée s'il s'agit d'un système terminant (ne présente pas de séquences de réécriture infinies). De plus, nous savons déjà que le lieu $ est borné par 17 car le système étudié comporte un nombre limité (50) de gâteaux et de pommes. Nous démontrerons maintenant l'aptitude de la sémantique proposée à vérifier l'exactitude d'une telle caractéristique. Pour ce faire, il est possible d'utiliser l'outil d'accessibilité [ii] ou le model-checker de Maude pour rechercher l'existence d'un marquage où la place $ va contenir 18 jetons en utilisant la commande suivante :

```
Maude> search in NEW-VENDING-MACHINE-2 : initial =>* M < $,18 > .
No solution.
states: 714867  rewrites: 15490529 in 4281906278ms cpu (105312ms real)
(3 rewrites/second)
```

Dans ce résultat, nous pouvons voir que l'outil d'accessibilité de Maude n'a pas trouvé de solution, ce qui signifie que la place $ ne va pas contenir 18 jetons durant l'évolution du système. Donc, cette place peut être bornée et pour confirmer cette supposition, nous devons vérifier s'il existe des marquages où le nombre de jetons à cet endroit est égal à 17 [iii].

```
Maude> search in NEW-VENDING-MACHINE-2 : initial =>! M < $,17 > .

Solution 1 (state 714857)
states: 714864  rewrites: 15490426 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
M --> < q,1 > < C,50 > < A,50 > < S1,0 > < S2,0 >

Solution 2 (state 714863)
states: 714866  rewrites: 15490496 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
M --> < q,2 > < C,50 > < A,50 > < S1,0 > < S2,0 >
```

ii. le model-checker de Maude ne peut pas être utilisé pour la sémantique existante sans l'opérateur supplémentaire indiqué dans la section 4.3. De plus, l'outil d'accessibilité ne peut pas être utilisé de manière absolue dans le cas de la sémantique existante car le nombre de jetons ne peut pas être indiqué (présenté) dans l'expression définissant le marquage

iii. Nous avons utilisé le paramètre `" =>! "` pour la commande *search* de Maude afin de réduire l'ensemble des solutions aux états finaux canoniques, c'est-à-dire les états qui ne peuvent pas être réécrits davantage. Sinon, le paramètre `" =>* "` peut alors être utilisé (voir [328] pour plus de détails).

```
Solution 3 (state 714866)
states: 714867  rewrites: 15490529 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
M --> < q,0 > < C,50 > < A,50 > < S1,0 > < S2,0 >

No more solutions.
states: 714867  rewrites: 15490529 in 4281906278ms cpu (109750ms real)
(3 rewrites/second)
```

Comme on peut le voir, la place $ peut contienir — et ne contient jamais plus de — 17 jetons, et par conséquent, elle borné à 17. On note ici que l'utilisation du model-checker de Maude pour ce test est plus aisée. Enfin, cette procedure de test peut être simplement élargie pour tester le réseau de Petri complet.

4.5 Les réseaux de Petri avec arc à poids variable

Pour mieux comprendre la notion de réseaux de Petri avec "un arc à poids variables", considérons l'exemple donné dans la Figure 6.2(a) qui est inspiré de [329].

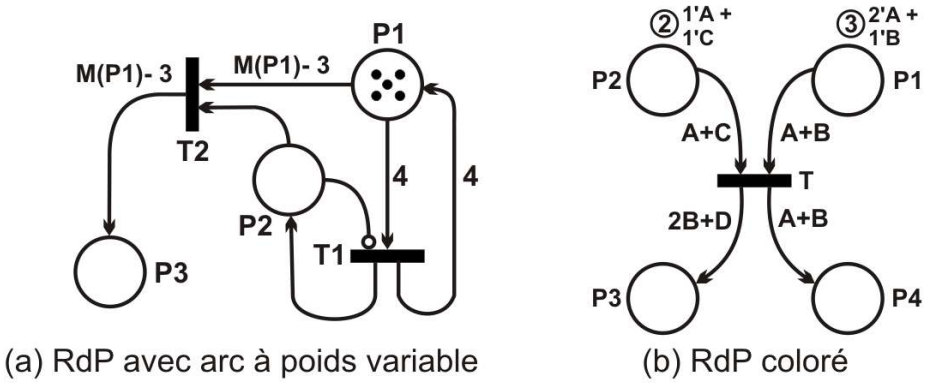

Figure 6.2: Exemples des extensions des réseaux de Petri

Dans cet exemple, les transitions T1 et T2 permettent de contrôler et de maintenir le marquage de la place P1 à un niveau souhaité fixé ici à 3 par exemple. Par conséquent, la transition T1 sera franchie lorsque le marquage de la place P1 est supérieur au niveau souhaité. Ensuite, un jeton à la place P2 et T2 sera activé, puis déclenché pour retirer les jetons en excès de la place P1 à travers le poids (M (P1) –3) de l'arc (P1,T2). Le code suivant représente le marquage initial et les règles de réécriture décrivant les transitions T1 qui a un arc inhibiteur et T2 qui a un arc à poids variable selon la sémantique proposée dans la section 3.

```
eq initial = <P1,5> <P2,0> <P3,0> .
crl [T1] : < P1,x > < P2,0 > => < P1,x - 4 + 4 > < P2,1 > if (x >= 4) .
crl [T2] : < P1,x > < P2,1 > < P3,y > => < P1,3 > < P2,0 > < P3,y + x - 3 > if (x > 3) .
```

Bien sûr, il convient de noter qu'il n'est pas possible de décrire des réseaux de Petri avec des poids d'arc variables dans la sémantique existante, même avec l'opérateur supplémentaire donné dans la section 4.3.

4.6 Les réseaux de Petri colorés

Le réseau de Petri coloré indiqué a quatre place P1, P2, P3 et P4 et quatre couleurs de jetons : Color1 (A), Color2 (B), Color3 (C) et Color4 (D). De plus, il possède trois ensembles de couleurs où P1 et P4 appartenant au même ensemble de couleurs (Color1, Color2), P2 appartient à l'ensemble (Color1, Color3) et P3 à l'ensemble (Color2, Color4). Par conséquent, la spécification correspondante est donnée comme suit :

```
fmod COLOURED-PN-SIGNATURE is
protecting INT .
sorts Color1 Color2 Color3 Color4 .
sorts PlaceNameset1 PlaceNameset2 PlaceNameset3  .
sorts Place Marking .
subsort Place < Marking .
op A : Int -> Color1 . op B : Int -> Color2 .
op C : Int -> Color3 . op D : Int -> Color4 .
ops P1 P4 : -> PlaceNameset1 .
op P2 : -> PlaceNameset2 .
op P3 : -> PlaceNameset3 .
op <_,_,_> : PlaceNameset1 Color1 Color2 -> Place [ctor] .
op <_,_,_> : PlaceNameset2 Color1 Color3 -> Place [ctor] .
op <_,_,_> : PlaceNameset3 Color2 Color4 -> Place [ctor] .
ops null initial : -> Marking .
op __ : Marking Marking -> Marking [ctor assoc comm id: null] .
eq initial = < P1,A(2),B(1) > < P2,A(1),C(1) > < P3,B(0),D(0) > < P4,A(0),B(0) > .
endfm

mod COLOURED-PN is
inc COLOURED-PN-SIGNATURE .
vars x1 x2 x3 y1 y2 z t : Int .
crl [T] : < P1,A(x1),B(y1) > < P2,A(x2),C(z) > < P3,B(y2),D(t) > < P4,A(x3),B(y2) > =>
        < P1,A(x1 - 1),B(y1 - 1) > < P2,A(x2 - 1),C(z - 1) > < P3,B(y2 + 2),D(t + 1) >
        < P4,A(x3 + 1),B(y2 + 1) > if (x1 >= 1) and (y1 >= 1) and (x2 >= 1) and (z >= 1) .
endm
```

Utilisons maintenant la commande `rewrite` pour explorer le comportement de ce réseau de Petri coloré. Le résultat de l'exécution de cette spécification est donné comme suit :

```
rewrite [1] in COLOURED-PN : initial .
rewrites: 17 in 541555185225ms cpu (0ms real) (0 rewrites/second)
result Marking: < P1,A(1),B(0) > < P4,A(1),B(1) > < P2,A(0),C(0) > < P3,B(2),D(1) >
```

Comme on peut le constater, la spécification de ce réseau selon la sémantique proposée est très claire et qu'il n'existe aucune confusion entre les noms des places avec leur couleurs correspondantes. De plus, dans la spécification précédente, nous avions deux couleurs dans tous les places du réseau de Petri coloré, mais dans le cas où les places ont un nombre de couleurs différent, la spécification peut être déclarée comme suit :

```
op A : Int -> Color1 . op B : Int -> Color2 . op C : Int -> Color3 .
op P1 : -> PlaceNameset1 . op P2 : -> PlaceNameset2 . op P3 : -> PlaceNameset3 .
op <_,_> : PlaceNameset1 Color1 -> Place [ctor] .
op <_,_,_> : PlaceNameset2 Color1 Color3 -> Place [ctor] .
op <_,_,_,_> : PlaceNameset3 Color1 Color2 Color3 -> Place [ctor] .
```

Dans cette spécification, nous supposons que nous avons trois couleurs (Color1 (A), Color2 (B) et Color3 (C)) et trois ensemble de couleurs de sorte que, P1 appartient au premier ensemble qui contient une seule couleur (Color1), P2 à la deuxième ensemble qui contient deux couleurs (Color1, Color3) et P3 appartient au troisième ensemble qui contient trois couleurs (Color1, Color2, Color3).
Enfin, nous présentons un résumé des points forts et des différences les plus importants entre la sémantique existante et celle proposée dans la Table 6.1.

	Sémantique proposée	**Sémantique existante**
Clarté et lisibilité de la specification	Mieux	
Distinction forte entre places et jetons	Mieux	
Sémantique de franchissement	Mieux	
Simplicité et concision des règles		Mieux
Habilité naturelle de compter les jetons	Oui	Non
Habilité de spécifier les arcs inhibiteurs	Oui	Nécessite d'autres opérateurs
Habilité de tester la bornitude	Oui	Nécessite d'autres opérateurs
Habilité de spécifier les arcs à poids variable	Oui	Non

Table 6.1: Comparaison entre les deux sémantiques

5 Conclusion

Dans ce chapitre, nous avons proposé une sémantique basée sur la logique de réécriture pour les réseaux de Petri afin de surmonter les limitations de la sémantique existante. En fait, les types fondamentaux pour définir la signature d'un réseau de Petri, les places et le marquage sont conservés avec quelques modifications de leur sémantique. La nouvelle sémantique de réseaux de Petri nous a permis de décrire facilement les

réseaux de Petri avec des arcs à poids variables, des arcs d'inhibiteurs et de tester leur bornitude. À l'avenir, nous souhaitons explorer la possibilité de spécifier et d'analyser d'autres extensions de réseaux de Petri de haut niveau tout en utilisant la nouvelle sémantique. Enfin, on note que ce chapitre fait l'objet du papier [6] qui est en cours de révision.

Conclusions et perspectives

Dans cette thèse, nous avons vu que l'approche multi-agents offre une alternative très séduisante aux représentations classiques des systèmes complexes en terme de modélisation, simulation et implémentation. Par conséquent, cette approche n'est pas limité à un domaine particulier et qu'elle peut être utilisé pour modéliser, simuler et vérifier les systèmes critique. Néanmoins, et du fait de l'hétérogénéité des modèles considérés dans le même système à développer et l'inéxistence de correspondance claire entre les modèles destinés à être développer et des structures informatiques clairement établies, il est donc crucial de disposer de méthodes de conception qui se basent sur des formalismes puissant et des des techniques de vérification rigoureuses, automatiques et efficaces pour assurer la fiabilité de ces systèmes. En plus, il est aussi très important d'utiliser des techniques de test automatique pour assurer l'absence d'erreur dans l'implémentation de ces systèmes. Autrement dit, ces méthodes et techniques ne doivent pas être limitées à une seule phase, mais couvrir tout le processus de développement des applications basées sur l'approche multi-agents.
En effet, le travail réalisé couvre les phases suivantes : spécification, vérification et implémentation. Il est motivé par trois objectifs principaux :

1. Utiliser le formalisme le plus adapté pour la modélisation des applications basées sur l'approche agent et établir une sémantique riche pour ce formalisme dans le cadre de la logique de réécriture.
2. Être capable de prouver des propriétés intéressantes d'un SMA à partir de sa spécification en utilisant des techniques de vérification formelles.
3. Assurer au maximum possible la correction du code proposé pour le SMA à développer car il s'agit d'un système critique.

Les contributions de ce travail se décomposent en deux thèmes principaux :
En ce qui concerne le formalisme utilisé, on a pu définir un algorithme pour la génération automatique de sa spécification correspondante dans la logique de réécriture afin d'éviter les erreurs de spécification. En plus, on a défini une sémantique améliorée en comparaison de la sémantique existante pour les réseaux de Petri dans le cadre de la logique de réécriture.
En ce qui concerne le modèle et le code du système à développer, on a utilisé des techniques très séduisante pour assurer leurs corrections dans l'aspect fonctionnel et l'aspect implémentatoire.
L'interêt essentiel de notre approche est de pouvoir modéliser un large spectre des systèmes multi-agents du fait du potentiel des réseaux de Petri et de la sémantique améliorée. En plus, le pouvoir de vérifier la correction du modèle et son implémentation.

Notre travail souffre néanmoins de quelques inconvénients du fait qu'il se limite sur l'utilisation de réseaux de Petri ordinaires — et quelques de ses extensions — qui ne permettent pas d'associer des informations aux jetons ou de décrirent les scénarios complexes des SMAs et des systèmes critiques. C'est pour ça, que nous envisageons de poursuivre nos recherches suivant plusieurs axes :

- Proposer une sémantique orientée objets (améliorée) pour les réseaux de Pétri afin de pouvoir inclure les ECATNets, les réseaux de Petri à agent et les réseaux de Petri imbriqués qui sont des formalismes prometteux pour la modélisation des SMAs.
- Utiliser la version RT-Maude pour adapter l'approche aux applications temps réels qui s'adaptant dynamiquement en fonction du contexte.
- Utiliser le langage Erlang pour l'implémentation du fait qu'il supporte la correction du code en cours d'exécution ce qui est très important pour le cas des systèmes critique.

Annexe

Etudes De Cas

Études de cas

Nous présentons ici deux études de cas afin de montrer l'efficacité de l'approche proposée et de mieux comprendre l'utilisation pratique des techniques de vérification et de test utilisées.

Sommaire

1 Étude de cas N^o : 1 (Machine de distribution automatique)

1.1 Description du système et modélisation

Nous considérons une simple machine de distribution automatisée placée dans un hôtel pour fournir du café, des gâteaux et des boissons aux visiteurs de l'hôtel. Le visiteur doit utiliser sa carte magnétique — de visiteur d'hôtel — pour obtenir des services de la machine.
Une telle machine est programmée quotidiennement pour délivrer une tasse de café, des gâteaux avec une bouteille d'eau et du jus (***service 1***) au visiteur lors de la première utilisation de sa carte de visiteur. Sinon, la machine ne distribue que du café et des gâteaux (***service 2***). Pour cela, la machine teste la carte insérée pour vérifier sa validité et détermine s'il s'agit de la première utilisation de la carte d'un visiteur ou non. Ensuite, le service correspondant est fourni au visiteur.
Par souci de simplicité, nous supposons que la carte du visiteur contient un code à barres à 14 chiffres où l'addition de chiffres aux positions (1, 2, 4, 8) égale à 20. En outre, les codes de cartes des visiteurs actuels dans les hôtels sont insérés quotidiennement dans une liste dans la mémoire de la machine et, lorsqu'une carte de visiteur valide est utilisée, elle est automatiquement supprimée de cette liste. Un tel mécanisme facilite le processus pour fournir ***service 1*** ou ***service 2*** à un visiteur.
Par conséquent, deux algorithmes sont utilisés pour tester la validité d'une carte (***testing card validity***) et pour déterminer le type de service à délivrer (**determining services**) dans les deux cas.
La Figure 8.1 montre le réseau de Petri qui modélise la machine de distribution, et le tableau 8.2 présente la signification des places et des transitions.

1.2 Spécification du système et des propriétés

1.2.1 Spécification du système

La théorie de réécriture correspondante du modèle de réseau de Petri est illustrée dans les modules `PETRI_NET` et `PN_ADM`. En effet, ces théories de réécritures sont préparées suivant la nouvelle sémantique proposée pour les réseaux de Petri car le réseau de Petri présenté dans la Figure 8.1 est celui du type avec arc inhibiteur.

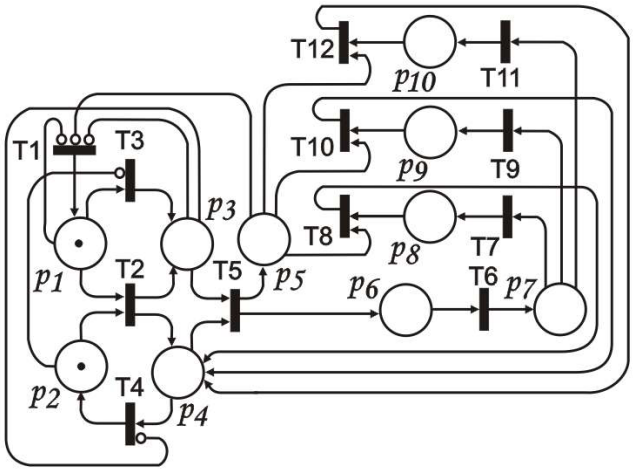

Figure 8.1: Réseau de Petri modélisant la machine de distribution

Signification des libellés des places	
P1	Un visiteur arrive et attend le démarrage de la machine
P2	Machine en état de veille
P3	Un visiteur est prêt (pour insérer sa carte)
P4	La machine est prête à l'emploi
P5	Un visiteur utilise la machine (attente de test et/ou service)
P6	La carte est insérée
P7	La carte est testée
P8	La carte invalide est rejetée
P9	Livraison du service 1 et retour de la carte au visiteur
P10	Livraison du service 2 et retour de la carte au visiteur

Signification des libellés des transitions	
T1	Ajouter un nouveau visiteur pour utiliser la machine (seulement s'il n'y a personne en attente, prêt ou utilisant la machine)
T2	Un visiteur appuie sur le bouton de démarrage de la machine
T3	Changer l'état du visiteur en état prêt (si la machine n'est pas en veille)
T4	La machine retourne à l'état de veille (s'il n'y a pas de visiteur prêt)
T5	Insérer une carte
T6	Tester la carte
T7	Rejeter une carte invalide
T8	Le visiteur prend une carte invalide et demande au responsable de la changer
T9	Acceptation de la carte et livraison du service 1
T10	Le service 1 est délivré et le visiteur prend sa carte
T11	Acceptation de la carte et livraison du service 1
T12	Le service 2 est délivré et le visiteur prend sa carte

Table 8.1: Signification des libellés des places et transitions

```
fmod PETRI_NET is
protecting INT .
sorts PlaceName Places Marking .
subsort Places < Marking .
ops P1 P2 P3 P4 P5 P6 P7 P8 P9 P10 : -> PlaceName .
op place(_,_) : PlaceName Int -> Places [ctor] .
op __ : Marking Marking -> Marking [ctor assoc comm id:null] .
op null : -> Marking .
```

```
endfm
mod PN_ADM is
protecting PETRI_NET .
rl[T1]: place(P1,0) place(P3,0) place(P5,0) => place(P1,1) place(P3,0) place(P5,0) .
rl[T2]: place(P1,1) place(P2,1) place(P3,0) place(P4,0) => place(P1,0) place(P2,0)
        place(P3,1) place(P4,1) .
rl[T3]: place(P1,1) place(P2,0) place(P3,0) => place(P1,0) place(P2,0) place(P3,1) .
rl[T4]: place(P2,0) place(P3,0) place(P4,1) => place(P2,1) place(P3,0) place(P4,0) .
rl[T5]: place(P3,1) place(P4,1) place(P5,0) place(P6,0) => place(P3,0) place(P4,0)
        place(P5,1) place(P6,1) .
rl[T6]: place(P6,1) place(P7,0) => place(P6,0) place(P7,1) .
rl[T7]: place(P7,1) place(P8,0) => place(P7,0) place(P8,1) .
rl[T8]: place(P8,1) place(P4,0) place(P5,1) => place(P8,0) place(P4,1) place(P5,0).
rl[T9]: place(P7,1) place(P9,0) => place(P7,0) place(P9,1) .
rl[T10]: place(P9,1) place(P4,0) place(P5,1) => place(P9,0) place(P4,1) place(P5,0).
rl[T11]: place(P7,1) place(P10,0) => place(P7,0) place(P10,1) .
rl[T12]: place(P10,1) place(P4,0) place(P5,1) => place(P10,0) place(P4,1) place(P5,0).
endm
```

Dans cette spécification, l'aspect statique (la signature) du système est défini dans le module fonctionnel `PETRI_NET`. Ce module a été importé dans le module système `PN_ADM`, dans lequel nous ajoutons une règle pour chaque transition afin de compléter la description de l'aspect dynamique de la machine.

1.2.2 Specification des propriétés

Dans le cas de notre système, les propriétés suivantes doivent être vérifiées :

- *Evolution du système :* Cette propriété garantit que le système est toujours actif (ou peut être activé) et qu'il ne sera pas bloqué dans une situation de blocage.
- *Servir les visiteurs :* Cette propriété garantit que lorsqu'un visiteur est présent et souhaite prendre quelque chose sur la machine, il doit être servi, soit par le service 1, le service 2, soit simplement en rejetant sa carte si elle n'est pas valide.
- *Contrôle des services :* Cette propriété garantit que chaque visiteur utilise le service 1 pour la première fois et le service 2 ensuite.

Ensuite, deux modules pour la spécification des propriétés du système sont préparés. Le premier module, `PN_ADM_PREDS`, décrit l'ensemble des propriétés supposées être vérifiées dans le modèle et qui serviront de référence dans le processus de vérification par l'outil de vérification de modèle. Cependant, dans le deuxième module, `ACS-CHECK`, le concepteur exprime les propriétés pertinentes à vérifier dans le modèle du système. Ce module contient les propriétés qui ne peuvent être exprimées que par la logique temporelle linéaire (LTL). Ces modules sont donnés comme suit :

```
mod PN_ADM_PREDS is
protecting PN_ADM .
including SATISFACTION .
subsort Marking < State .
ops New-V-Coming V-Ready V-Use-Machine : -> Prop .
ops Machine-Sleep Machine-Ready Machine-In-Use : -> Prop .
ops Reject-Card Service-1 Service-2 : -> Prop .
var A : Marking .
*** ---------- SOME IMPORTANT PROPERTIES -----------
eq place(P1,1) A |= New-V-Coming = true .
eq place(P3,1) A |= V-Ready = true .
eq place(P5,1) A |= V-Use-Machine = true .
eq place(P2,1) A |= Machine-Sleep = true .
eq place(P4,1) A |= Machine-Ready = true .
eq place(P6,1) A |= Machine-In-Use = true .
eq place(P7,1) A |= Machine-In-Use = true .
eq place(P7,1) place(P5,1) A |= Reject-Card = true .
eq place(P7,1) place(P5,1) A |= Service-1 = true .
eq place(P7,1) place(P5,1) A |= Service-2 = true .
endm

mod ADM-CHECK is
inc PN_ADM_PREDS .
inc MODEL-CHECKER .
inc LTL-SIMPLIFIER .
op initial : -> Marking .
eq initial = place(P1,1) place(P2,1) place(P3,0) place(P4,0) place(P5,0) place(P6,0)
             place(P7,0) place(P8,0) .
ops no-deadlock serving : -> Prop .
eq no-deadlock = [](New-V-Coming \/ V-Ready \/ V-Use-Machine \/ Machine-Ready \/
                 Machine-Sleep \/ Reject-Card \/ Service-1 \/ Service-2 ) .
eq serving = [](P-Use-Machine) -> <> (Reject-Card \/ Service-1 \/ Service-2)) .
endm
```

1.3 Vérification du modèle

À partir de l'état initial du réseau de Petri, le model-checker de Maude a été utilisé pour la vérification de deux propriétés.

– *Evolution du système :* Cette propriété est définie dans le module `ADM-CHECK` comme `"no-deadlock"`. Après vérification, le model-checker a confirmé l'exactitude de cette propriété comme suit :

```
Maude> reduce in ADM_CHECK : modelCheck(initial, no-deadlock) .
       rewrites: 653 in -42042743134ms cpu (31ms real) (~ rewrites/second)
       result Bool: true
```

– *Servir les visiteurs :* Cette propriété est définie comme `"serving"` dans le module `ADM-CHECK`. Le model-checker a également confirmé l'exactitude de cette propriété.

```
Maude> reduce in ADM_CHECK : modelCheck(initial, serving) .
       rewrites: 70 in -41991903084ms cpu (0ms real) (~ rewrites/second)
       result Bool: true
```

1.4 Test d'implémentation

Selon l'approche proposée, nous avons utilisé PBT pour tester la propriété "*controlling service*" mentionnée dans la section 1.2.2, car elle n'est pas exprimable sous forme de formule LTL et ne peut pas être vérifiée même par Maude "*search tool*". Cependant, il peut être traduit en plusieurs propriétés déclaratives, telles que :

```
property_first_time_client_gets_first_service() ->
  ?FORALL(Client, valid_client_card(),
          proper_first_service(sut:use_machine(Client))).

property_following_times_client_gets_second_service() ->
  ?FORALL({Client, Uses}, {valid_client_card(), nat()},
           proper_second_service(
             repeat(Uses+1, sut, use_machine, [Client]))).
```

Ces deux fonctions ont pour but de tester les deux possibilités en matière de contrôle de service : soit le client obtient le premier service lors de l'utilisation de la machine, ou bien si la carte a déjà été utilisée et le service doit donc être du second type. Ces instructions doivent être vérifiées *pour toutes* les *cartes valides des clients* que nous testons.
Dans cette première formulation, le type de test à effectuer est *test positif*, car seules les cartes client *valide* sont générées pour être utilisées en tant qu'entrée. La fonction `valid_client_card` génère des données qui, conformément à la description de la section 1.1, peut être implémenté comme suit :

```
valid_client_card() ->
  ?SUCHTHAT({P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14},
            {nat(),nat(),nat(),nat(),nat(),nat(),nat(),nat(),nat(),nat(),nat(),nat(),
             nat(),nat()}, P1+P2+P4+P8 == 20).
```

où `nat()` est un générateur de données QuickCheck qui produit des nombres naturels aléatoires. Le générateur personnalisé que nous construisons ici produit à son tour des séquences de 14 nombres naturels, mais ne renvoie que ceux qui remplissent la condition `P1+P2++P4+P8 == 20`).
Les seules fonctions supplémentaires utilisées ci-dessus sont `proper_first_service` et `proper_second_service`, qui vérifient le résultat de l'appel de la fonctionnalité testée

sur le système (`sut:use_machine(Client)`). Nous les incluons ici pour montrer comment utiliser la version QuickCheck de Erlang pour implémenter ce code de test et illustrer ses utilités universelles indépendamment de la technologie utilisée dans l'implémentation du système de manière déclarative.

```
proper_first_service(coffee)  -> true;
proper_first_service(cake)    -> true;
proper_first_service(water)   -> true;
proper_first_service(juice)   -> true;
proper_first_service(_)       -> false.

proper_second_service(coffee) -> true;
proper_second_service(cake)   -> true;
proper_second_service(_)      -> false.
```

Enfin, nous implémentons une fonction récursive pour appeler le système sous test un nombre de fois aléatoire, afin de simuler la carte utilisée n fois après sa première utilisation. En plus, *pour tout* n que nous générons, la propriété ci-dessus devrait vérifier.

2 Étude de cas N^o : 2 (Système de contrôle de qualité)

2.1 Description du système et modélisation

Notre étude de cas représente un système industriel composé de quatre robots (agents) qui ont pour rôle de contrôler la qualité d'emballage d'un produit chimique dangereux. Chaque produit est caratérisé par un code (chaîne de caractère) palindrome. Pour cela, un robot doit contrôler le code du produit. En plus, un robot doit aussi se coordonner en communiquant avec ses deux voisins de gauche et de droite pour assurer la qualité d'emballage et d'empaquetage.
Le réseau de Petri (Figure 8.2) illustre le modèle de communication proposé pour chaque robot du système avec ses voisins.

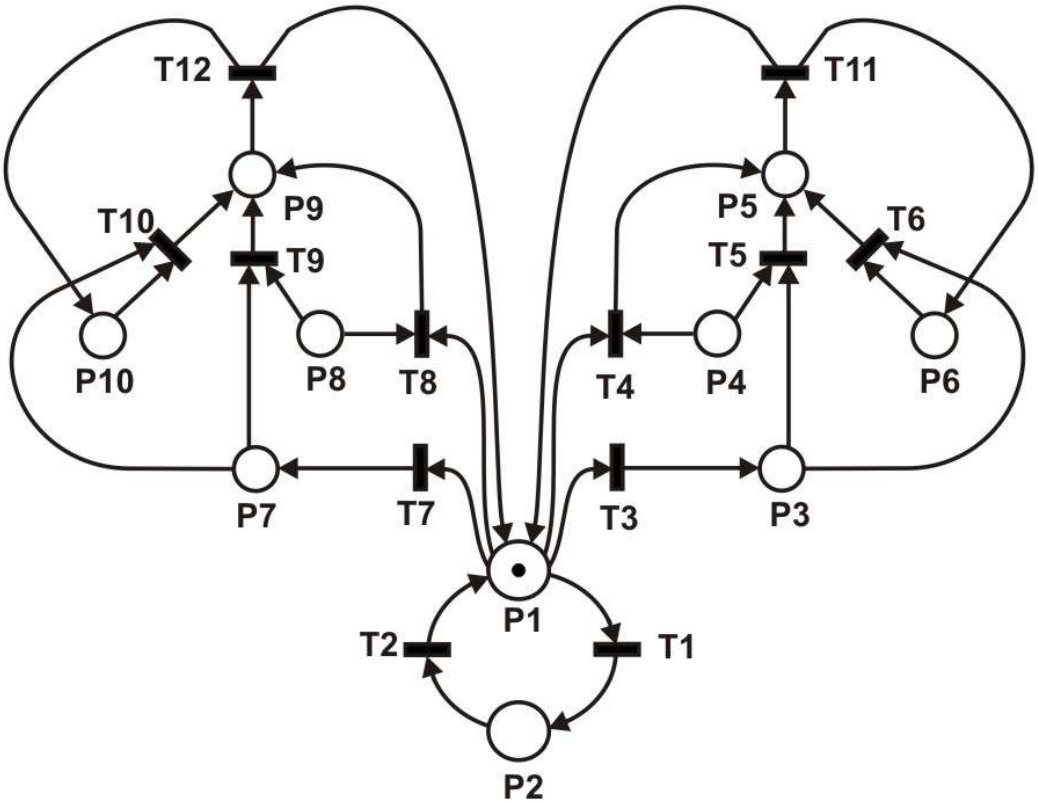

Figure 8.2: Modèle de communication entre deux robots

Les deux tableaux ci-dessous (voir Table 8.2) expliquent la signification des libellés des places et transitions du réseau de Petri

P1	l'agent X est en état Libre
P2	l'agent X est en état de vérification de code
P3	l'agent X est en état de besoin de communication avec l'agent de droite
P4	l'agent de droite est en état de besoin de communication avec l'agent X
P5	l'agent X est en état de communication avec l'agent de droite
P6	l'agent de droite est en état libre
P7	l'agent X est en état de besoin de communication avec l'agent de gauche
P8	l'agent de gauche est en état de besoin de communication avec l'agent X
P9	l'agent X est en état de communication avec l'agent de gauche
P10	l'agent de gauche est en état libre

A : Signification des Libellés des Places

T1	l'agent X passe à l'état de vérification de code
T2	l'agent X passe à l'état Libre
T3	l'agent X demande la communication avec l'agent de droite
T4	l'agent X et l'agent de droite passent à la communication
T5	l'agent X et l'agent de droite passent à la communication
T6	l'agent X et l'agent de droite passent à la communication
T7	l'agent X demande la communication avec l'agent de gauche
T8	l'agent X et l'agent de gauche passent à la communication
T9	l'agent X et l'agent de gauche passent à la communication
T10	l'agent X et l'agent de gauche passent à la communication
T11	l'agent X et l'agent de droite se liberent
T12	l'agent X et l'agent de gauche se liberent

B : Signification des Libellés des Transitions

Table 8.2: Signification des Places et Transition du Réseau de Petri

2.2 Spécification du système et des propriétés

Comme c'est déjà noté dans la sous section 3.3.2, le langage Maude définit deux niveaux de spécification. Le premier niveau concerne la spécification du système tandis que le deuxième est lié à la spécification des propriétés.

2.2.1 Spécification du Système

La théorie de réécriture associée à la spécification du modèle proposé est donnée par le module suivant :

```
mod MAS-COMMUNICATION is
pr BOOL .
inc CONFIGURATION .
sorts states action .
op AGENT : -> Cid .
ops communicate ready : -> states .
```

```
ops reqcomRR reqcomLR none : -> action .
op state :_ : states -> Attribute [gather(&)] .
op actiontodo :_ : action -> Attribute [gather(&)] .
vars M N : action .
ops Agent1 Agent2 Agent3 Agent4 : -> Oid .

*** agent 1 requests for communication ***
rl [1-req-2] : < Agent1 : AGENT | state : ready , actiontodo: none > =>
< Agent1 : AGENT | state : ready , actiontodo : reqcomLR > .
rl [1-req-4] : < Agent1 : AGENT | state : ready , actiontodo: none > =>
< Agent1 : AGENT | state : ready , actiontodo : reqcomRR > .

*** agent 1 and agent 2 are in communication ***
rl [1-com-2-req] : < Agent1 : AGENT | state : ready , actiontodo :none >
< Agent2 : AGENT | state : ready , actiontodo : reqcomRR > => < Agent1 : AGENT |
state : communicate , actiontodo : none > < Agent2 : AGENT | state : communicate ,
actiontodo : reqcomRR > .
rl [1-req-com-2-req] : < Agent1 : AGENT | state : ready , actiontodo: reqcomLR >
< Agent2 : AGENT | state : ready , actiontodo : reqcomRR > => < Agent1 : AGENT |
state : communicate , actiontodo: reqcomLR > < Agent2 : AGENT | state: communicate ,
actiontodo: reqcomRR > .
rl [1-req-com-2] : < Agent1 : AGENT | state : ready , actiontodo : reqcomLR >
< Agent2 : AGENT | state : noaction , actiontodo : none > => < Agent1 : AGENT |
state : communicate , actiontodo : reqcomLR > < Agent2 : AGENT | state :
communicate , actiontodo : none > .

*** terminate communication between agent 1 and agent 2 ***
crl [1-endcom-2] : < Agent1 : AGENT | state : communicate , actiontodo : M >
< Agent2 : AGENT | state : communicate , actiontodo : N > => < Agent1 : AGENT |
state : noaction , actiontodo : none > < Agent2 : AGENT | state : noaction ,
actiontodo : none >  if ( M == reqcomLR ) or ( N == reqcomRR ) .

*** agent 1 and agent 4 are in communication ***
rl [4-com-1-req] : < Agent4 : AGENT | state : ready , actiontodo : none >
< Agent1 : AGENT | state : ready , actiontodo : reqcomRR > => < Agent4 : AGENT |
state : communicate , actiontodo : none > < Agent1 : AGENT | state : communicate ,
actiontodo : reqcomRR > .
rl [4-req-com-1-req] : < Agent4 : AGENT | state : ready , actiontodo: reqcomLR >
< Agent1 : AGENT | state : ready , actiontodo : reqcomRR > => < Agent4 : AGENT |
state : communicate , actiontodo: reqcomLR > < Agent1 : AGENT | state: communicate ,
 actiontodo: reqcomRR > .
rl [44-com-1] : < Agent4 : AGENT | state : ready , actiontodo: reqcomLR > < Agent1 :
AGENT |state : ready , actiontodo : none> => < Agent4 : AGENT | state : communicate ,
actiontodo : reqcomLR > < Agent1 : AGENT | state : communicate , actiontodo : none > .

*** terminate communication between agent 1 and agent 4 ***
```

```
crl [4-endcom-1] : < Agent4 : AGENT | state : communicate ,actiontodo : M >
< Agent1 : AGENT | state : communicate ,actiontodo : N > => < Agent4 : AGENT |
state : noaction , actiontodo : none > < Agent1 : AGENT | state : noaction ,
actiontodo : none > if ( M == reqcomLR ) or ( N == reqcomRR ) .

*** put the specification of three other agents here ***
. . .
endm
```

2.2.2 Spécification des Propriétés

Tout d'abord, nous plaçons l'ensemble des propriétés supposées vérifiées dans un module séparé, puis nous exprimons spécification des propriétés à vérifier dans le deuxième module. Nous donnons ici les deux modules :

```
mod PROPERTY is
protecting MAS-COMMUNICATION .
including SATISFACTION .
subsort Configuration < State .
ops com1-2 com2-3 com3-4 com4-1 : -> Prop .
ops ENDcom1-2 ENDcom2-3 ENDcom3-4 ENDcom4-1 : -> Prop .
op PropReqComLR : Oid -> Prop .
op PropReqComRR : Oid -> Prop .
var M : Configuration .
var N : Oid .
vars O P : action .
*** Agent requests for communication ***
eq M < N : AGENT | state : ready , actiontodo : none > |= PropReqComLR(N) = true .
eq M < N : AGENT | state : ready , actiontodo : none > |= PropReqComRR(N) = true .
*** Agent 1 and Agent 2 in communication ***
ceq M < Agent1 : AGENT | state : ready , actiontodo : O > < Agent2 : AGENT |
state : ready , actiontodo : P > |= com1-2 = true if ( ( O == reqcomLR ) and
( P == reqcomRR or P == none) or (O == none and P == reqcomRR ) ) .
*** END communication between agent 1 and agent 2 ***
ceq M < Agent1 : AGENT | state : communicate , actiontodo : O > < Agent2 : AGENT |
state : communicate , actiontodo : P > |= ENDcom1-2 = true if ( O == reqcomLR) or
( P == reqcomRR ) .
ceq M < Agent4 : AGENT | state : communicate , actiontodo : O > < Agent1 : AGENT |
state : communicate , actiontodo : P > |= ENDcom4-1 = true if ( O == reqcomLR) or
( P == reqcomRR ) .

***  Other properties must be put here  ***
. . .
endm
```

Le deuxième module est donné comme suit :

```
mod CHECK is
inc PROPERTY .
inc MODEL-CHECKER .
inc LTL-SIMPLIFIER .
op initial : -> Configuration .
op deadlock : -> Prop .
eq communication = [] (PropReqComLR(Agent1) \/ PropReqComRR(Agent1) \/
PropReqComLR(Agent2) \/ PropReqComRR(Agent2) \/ PropReqComLR(Agent3) \/
PropReqComRR(Agent3) \/ PropReqComLR(Agent4) \/ PropReqComRR(Agent4) \/ com1-2 \/
com2-3 \/ com3-4 \/ com4-1 \/ ENDcom1-2 \/ ENDcom2-3 \/ ENDcom3-4 \/ ENDcom4-1 )) .

eq initial = < Agent1 : AGENT | state : ready , actiontodo : none > < Agent2 :
AGENT | state : ready , actiontodo : none > < Agent3 : AGENT | state : ready ,
actiontodo : none > < Agent4 : AGENT | state : ready , actiontodo : none > .
endm
```

2.3 Vérification du modèle

- **Evolution and deadlock absence** : Nous aimerions vérifier que la communication est toujours possible et que le système ne sera pas dans une situation où il ne pourra pas continuer son exécution. Pour cela, nous testons cette propriété en utilisant la commande suivante :
  `Maude> red modelCheck(initial, deadlock ) .`
  Après le test, Model-Checker nous a donné un long exemple de compteur illustrant le fait que cette propriété n'est pas vérifiée. Nous reprenons cet exemple de compteur comme suit :

```
result ModelCheckResult: counterexample ({< Agent1 : AGENT | state : ready ,
actiontodo : none > < Agent2: AGENT | state : ready , actiontodo : none > < Agent3 :
AGENT | state : ready , actiontodo : none > < Agent4 : AGENT | state: ready ,
actiontodo : none >,'1-req-com-2} {< Agent1 : AGENT | state : ready, actiontodo :
reqcomLR > < Agent2 : AGENT | state : ready, actiontodo : none > < Agent3 : AGENT |
state: ready ,actiontodo : none > < Agent4 : AGENT | state :ready , actiontodo :
none >,'2-req-com-3} {< Agent1 : AGENT | state : ready ,actiontodo : reqcomLR >
< Agent2 : AGENT | state : ready ,actiontodo : reqcomLR > < Agent3 : AGENT | state :
ready ,actiontodo : none > < Agent4 : AGENT | state : ready ,actiontodo : none >,
'3-req-com-4} {< Agent1 : AGENT | state : ready ,actiontodo : reqcomLR > < Agent2 :
AGENT | state : ready ,actiontodo : reqcomLR > < Agent3 : AGENT | state : ready ,
actiontodo : reqcomLR > < Agent4 : AGENT | state : ready ,actiontodo : none >,
'4-req-com-1}, {< Agent1 : AGENT | state : ready ,actiontodo : reqcomLR > < Agent2 :
```

```
AGENT | state : ready , actiontodo : reqcomLR > < Agent3 : AGENT | state : ready ,
actiontodo : reqcomLR > < Agent4 : AGENT | state : ready , actiontodo : reqcomLR >
< Agent5 : AGENT | state : ready ,actiontodo : reqcomLR >, deadlock})
```

- **Test de confluence** : La situation de confluence est atteinte lorsque deux robots demandent en même temps la communication avec le même robot. Nous utilisons l'outil de d'atteignabilité de Maude pour tester cette propriété.

```
Maude> search init1 =>+ C:Configuration < Agent1 : AGENT | state :ready ,
actiontodo : none > < Agent2 : AGENT | state : ready , actiontodo : reqcomRR >
< Agent4 : AGENT | state : ready , actiontodo : reqcomLR > .

search in CHECK : init1 =>+ C:Configuration < Agent1 : AGENT | state : ready ,
actiontodo : none > < Agent2 : AGENT | state : ready ,actiontodo : reqcomRR >
< Agent4 : AGENT | state : ready ,actiontodo : reqcomLR > .

Solution 1 (state 37)
states: 38 rewrites: 44
C:Configuration --> < Agent3 : AGENT | state : ready , actiontodo : none >

Solution 2 (state 96)
states: 97 rewrites: 199
C:Configuration --> < Agent3 : AGENT | state : ready , actiontodo : reqcomLR >

Solution 3 (state 109)
states: 110 rewrites: 227
C:Configuration --> < Agent3 : AGENT | state : ready, actiontodo : reqcomRR >

No more solutions.
states: 213 rewrites: 2021
```

Comme nous le voyons, cette propriété n'est pas non plus vérifiée et l'outil d'atteignabilité de Maude nous a donné trois situations possibles.
Enfin, dans une telle situation, le développeur doit reviser afin de corriger les failles dans le modèle proposé pour le système avant de passer à la réalisation.

2.4 Test d'implémentation

La formalisation et la vérification effectuées jusqu'à présent se sont concentrées sur l'aspect le plus critique d'un système de sécurité. En effet, "*Verifcation du code*" est une autre propriété intéressante, qui est pratiquement implémenté à l'aide de l'algorithme de test palindrome. Malheureusement, cette propriété n'est pas exprimable sous forme

de formule LTL et par conséquent, la technique de test basée sur cette propriété est utilisée.
La fonction C proposée pour vérifier si une chaîne de caractère (code) est palindrome est illustrée dans le Listing 8.1. Pratiquement, la préparation de cette fonction ne nécessite probablement ni du temps, ni des efforts, mais ses tests basés sur les propriétés donneront beaucoup plus de confiance dans sa réalisation que de simples tests unitaires.

```
int isPalindrome(char code[])
{ char reverse_code[] int i;
  int h = strlen(code);
    for (i = h - 1; i >= 0 ; i--)  // inversement de la chaine de caractères
    {  reverse_code[h - i - 1] = code[i];   }

    if (strcmp(code,reverse_code) == 0)
    {  return 0;  }  // chaine est palindrome
    else
    {  return -1; }  // chaine n'est pas palindrome
}
```

Listing 8.1: La fonction "Palindrome" en langage C

D'autre part, l'expression universellement quantifiée que nous utilisons en tant que propriété sera comme suit :

```
prop_palindrome() ->
       ?FORALL(P, maybe_palindrome(), sut:validate_user(P) == is_palindrome(P)).
```

Où `maybe_palindrome()` est une fonction de génération d'entrée (codes aléatoires), `sut:validate_user()` est l'implémentation de l'algorithme à tester et `is_palindrome()` est notre fonction de test.

```
maybe_palindrome() -> oneof([palindrome(), string()]).

palindrome() ->
   ?LET(Base, string(),
      ?LET(Middle, oneof([[], [char()]]),
         Base ++ Middle ++ lists:reverse(Base))).

string() ->
   list(char()).

is_palindrome(S) ->
   S == lists:reverse(S).
```

Comme mentionné précédemment, nous utilisons la fonction `maybe_palindrome()` pour générer des codes aléatoires (chaînes, parfois des palindromes, parfois simplement des listes de caractères) afin de vérifier si elles sont considérées tous comme des palindromes (et obtiennent donc un accès) par la même implémentation du système comme notre fonction `is_palindrome()` le fait. L'exécution d'une telle propriété à l'aide de QuickCheck nous assure que tel est le cas et que seul le code palindrome y aura accès :

```
> eqc:quickcheck(prop_priviledged_employee()).
.....................................................................
OK, passed 100 tests
true
```

Et d'autant plus que nous avons le temps nécessaire pour exécuter plus de tests, plus le niveau de confiance sera élevé.

```
> eqc:quickcheck(eqc:numtests(10000,palindrome: prop_check_indices())).
.....................................................................(x10)
.....................................................................(x100)
.....................................................................
OK, passed 10000 tests
true
```

Bibliographie

[1] Ammar Boucherit, Laura M Castro, Abdallah Khababa, and Osman Hasan. Towards the formal development of software based systems : Access control system as a case study. *Information Technology And Control*, 47(3) :393–405, 2018.

[2] Ammar Boucherit, Abdallah Khababa, and Laura M Castro. Automatic generating algorithm of rewriting logic specification for multi-agent system models based on petri nets. *Multiagent and Grid Systems*, 14(4) :403–418, 2018.

[3] Boucherit Ammar and Khababa Abdallah. Contribution for the formal analysis of agent based critical systems properties. *Wulfenia Journal*, 20(1) :217–230, 2013.

[4] Boucherit Ammar and Khababa Abdallah. Towards the formal specification and verification of multi-agent based systems. *IJCSI*, 2011.

[5] Boucherit Ammar, Khababa Abdallah, and Belala Faiza. Rewriting logic based approach for the formalization of critical systems based on multi-agent system. *International Journal of Computer Applications*, 13(2), 2011.

[6] Ammar Boucherit, Kamel M Barkaoui, Osman Hasan, and Abdallah Khababa. An enhanced rewriting logic based semantics for petri nets. *Journal of Logical and Algebraic Methods in Programming*, IN REVIEW AFTER REVISION.

[7] Jean-François Monin. *Understanding formal methods*. Springer Science & Business Media, 2012.

[8] José Meseguer. Rewriting logic as a semantic framework for concurrency : a progress report. In *International Conference on Concurrency Theory*, pages 331–372. Springer, 1996.

[9] José Meseguer. Membership algebra as a logical framework for equational specification. In *International Workshop on Algebraic Development Techniques*, pages 18–61. Springer, 1997.

[10] Mark-Oliver Stehr, José Meseguer, and Peter Csaba Ölveczky. Rewriting logic as a unifying framework for petri nets. In *Unifying Petri Nets*, pages 250–303. Springer, 2001.

[11] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical computer science*, 96(1) :73–155, 1992.

[12] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Formal models and semantics*, pages 243–320. Elsevier, 1990.

[13] Neal A Harman. Correctness and verification of hardware systems using maude. Technical report, Technical Report 3-2000, Department of Computer Science, University of Wales Swansea, 2000, http ://www-compsci.swan.ac.uk/reports/yr2000/CSR3-2000.pdf, 2000.

[14] Neal A Harman. Verifying a simple pipelined microprocessor using maude. In *International Workshop on Algebraic Development Techniques*, pages 128–151. Springer, 2001.

[15] Francisco Durán, Manuel Roldán, and Antonio Vallecillo. Using maude to write and execute odp information viewpoint specifications. *Computer Standards & Interfaces*, 27(6) :597–620, 2005.

[16] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, and Stavros Tripakis. Verifying ptolemy ii discrete-event models using real-time maude. In *International Conference on Formal Engineering Methods*, pages 717–736. Springer, 2009.

[17] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-npa : Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V*, pages 1–50. Springer, 2009.

[18] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, Edward A Lee, and Stavros Tripakis. Verifying hierarchical ptolemy ii discrete-event models using real-time maude. *Science of Computer Programming*, 77(12) :1235–1271, 2012.

[19] Sonia Santiago, Santiago Escobar, Catherine Meadows, and José Meseguer. A formal definition of protocol indistinguishability and its verification using maude-npa. In *International Workshop on Security and Trust Management*, pages 162–177. Springer, 2014.

[20] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about Maude — a high-performance logical framework : how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[21] Alberto Verdejo and Narciso Martí-Oliet. Two case studies of semantics execution in maude : Ccs and lotos. *Formal Methods in System Design*, 27(1-2) :113–172, 2005.

[22] Prasanna Thati, Koushik Sen, and Narciso Martí-Oliet. An executable specification of asynchronous pi-calculus semantics and may testing in maude 2.0. *Electronic Notes in Theoretical Computer Science*, 71 :261–281, 2004.

[23] Peter Borovanskỳ, Claude Kirchner, and Helene Kirchner. Controlling rewriting by rewriting. *Electronic Notes in Theoretical Computer Science*, 4 :169–189, 1996.

[24] Claude Marché. Normalized rewriting : an alternative to rewriting modulo a set of equations. *Journal of symbolic computation*, 21(3) :253–288, 1996.

[25] JA Goguen and J Tardo. Obj-0 preliminary users manual. university of california at los angeles. *Computer Science Department*, 1977.

[26] Peter Borovanskỳ, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélene Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. Elan v 3.4 user manual. *LORIA, Nancy (France),*, 561, 2000.

[27] Kokichi Futatsugi and Ataru Nakagawa. An overview of cafe specification environment-an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*, pages 170–181. IEEE, 1997.

[28] M Clave, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Marti-Oliet, José Meseguer, and José F Quesada. Towards maude 2.0. *Electronic Notes in Theoretical Computer Science*, 36 :294–315, 2000.

[29] Peter Borovanskỳ, Claude Kirchner, Hélene Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan : A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science*, 4 :35–50, 1996.

[30] Peter Borovanskỳ, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. Elan from a rewriting logic point of view. *Theoretical Computer Science*, 285(2) :155–185, 2002.

[31] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ report : The language, proof techniques, and methodologies for object-oriented algebraic specification*, volume 6. World Scientific, 1998.

[32] Joseph A Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing obj. In *Software Engineering with OBJ*, pages 3–167. Springer, 2000.

[33] Kokichi Futatsugi. Fostering proof scores in cafeobj. In *International Conference on Formal Engineering Methods*, pages 1–20. Springer, 2010.

[34] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.7. 1). *SRI International*, 2016.

[35] Theodore McCombs. Maude 2.0 primer. *Department of Computer Science, University of Illinois and Urbana-Champaign, Urbana-Champaign, Ill., USA*, 2003.

[36] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Darmstadt University of Technology, Germany, 1962.

[37] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[38] Stamos K Andreadakis and Alexander H Levis. Synthesis of distributed command and control for the outer air battle. Technical report, Massachusetts Institute of Technology, Cambridge, Laboratory for Information and Decision Systems, 1988.

[39] D Mandrioli, A Morzenti, M Pezze, P Pietro, and S Silva. A Petri net and logic approach to the specification and verification of real time systems. *Formal Methods for Real-Time Computing*, 5, 1996.

[40] Richard Zurawski and MengChu Zhou. Petri nets and industrial applications : A tutorial. *IEEE Transactions on industrial electronics*, 41(6) :567–583, 1994.

[41] Monika Heiner and Maritta Heisel. Modeling safety-critical systems with Z and Petri nets. In *SAFECOMP*, volume 99, pages 361–374. Springer, 1999.

[42] Nancy G. Leveson and Janice L. Stolzy. Safety analysis using Petri nets. *IEEE Transactions on Software Engineering*, (3) :386–397, 1987.

[43] Tilak Agerwala. Complete model for representing the coordination of asynchronous processes. Technical report, Johns Hopkins Univ., Baltimore, Md.(USA), 1974.

[44] Diego C Martinez, Maria Laura Cobo, and Guillermo Ricardo Simari. A petri net model of argumentation dynamics. In *International Conference on Scalable Uncertainty Management*, pages 237–250. Springer, 2014.

[45] Michel Hack. Petri net language. 1976.

[46] Rüdiger Valk. Self-modifying nets, a natural extension of petri nets. In *International Colloquium on Automata, Languages, and Programming*, pages 464–476. Springer, 1978.

[47] K Labadi, T Benarbia, and M Darcherif. Sur la régulation des systèmes de vélos en libre-service : Approche basée sur les réseaux de Petri. In *8th ENIM IFAC International Conference of Modeling and Simulation*, 2010.

[48] Taha Benarbia, Karim Labadi, Abdel Moumen Darcherif, and Maurice Chayet. Modelling and control of self-service public bicycle systems by using Petri nets. *International Journal of Modelling, Identification and Control*, 17(3) :173–194, 2012.

[49] Kurt Jensen. Coloured petri nets. In *Petri nets : central models and their properties*, pages 248–299. Springer, 1987.

[50] Kurt Jensen. *Coloured Petri nets : basic concepts, analysis methods and practical use*, volume 1. Springer Science & Business Media, 2013.

[51] C A. R. Hoare. An axiomatic basis for computer programming. *Communication of the ACM*, 12 :576–580, 10 1969.

[52] Yannis Bres. *Exploration implicite et explicite de l'espace d'états atteignables de circuits logiques Esterel.* PhD thesis, Université Nice Sophia Antipolis, 2002.

[53] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang.* Prentice-Hall, 2 edition, 1996.

[54] D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using functional programming within an industrial product group : Perspectives and perceptions. *ACM SIGPLAN Notices*, 45(9) :87–92, 2010.

[55] Quviq a. b. `http://www.quviq.com`, 2008.

[56] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *5th ACM SIGPLAN Workshop on Erlang (Erlang'06), Portland, Oregon, USA, September 18-20, 2006*, pages 2–10, New York, USA, 2006. ACM.

[57] Samuel Rivas, Miguel Ángel Francisco, and Víctor M. Gulías. Property Driven Development in Erlang, by Example. In *5th Workshop on Automation of Software Test (ASE '10), Cape Town, South Africa, May 1-8, 2010*, pages 75–78, New York, USA, 2010. ACM.

[58] Thomas Arts, Laura M. Castro, and John Hughes. Testing Erlang Data Types with Quviq QuickCheck. In *7th ACM SIGPLAN workshop on ERLANG (Erlang'08), Victoria, BC, Canada, September 20-28, 2008*, pages 1–8, New York, USA, 2008. ACM.

[59] Laura M. Castro and Thomas Arts. Testing Data Consistency of Data-Intensive Applications Using QuickCheck. In *10th Spanish Conference on Programming and Languages (PROLE'10), Valencia, Spain, September 8–10, 2010. Revised Selected Papers*, volume 271 of *Electronic Notes in Theoretical Computer Science*, pages 41–62, Amsterdam, The Netherlands, 2011. Elsevier Science Publishers.

[60] Laura M. Castro, Miguel A. Francisco, and Víctor M. Gulías. Testing integration of applications with QuickCheck. In *Intl. Conference on Computer Aided Systems Theory*, 2009.

[61] Nicolae Paladi and Thomas Arts. Model based testing of data constraints : Testing the business logic of a mnesia database application with Quviq QuickCheck. In *8th ACM SIGPLAN workshop on Erlang*, 2009.

[62] Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *14th ACM SIGPLAN International Conference on Functional Programming.* ACM Press, 2009.

[63] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71 :162–187, 2004.

[64] Menad Nadia. *Modélisation des Systèmes embarqués Temps-Réel : Vers une ingénierie dirigée par les méthodes formelles.* PhD thesis, Université Mohamed Boudiaf des sciences et de la technologie d'Oran, 2015.

[65] Jacques Ferber. *Les systèmes multi-agents : vers une intelligence collective.* InterEditions, 1997.

[66] Michael Wooldridge. *An introduction to multiagent systems.* John Wiley & Sons, 2009.

[67] Michael Wooldridge, Nicholas R Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and multi-agent systems*, 3(3) :285–312, 2000.

[68] Luc Steels. The artificial life roots of artificial intelligence. *Artificial life*, 1(1_2) :75–110, 1993.

[69] Brahim Chaib-Draa, Imed Jarras, and Bernard Moulin. Systèmes multi-agents : principes généraux et applications. *Edition Hermès*, 242 :1030–1044, 2001.

[70] Guy A Boy. The orchestra : A conceptual model for function allocation and scenario based engineering in multi-agent safety-critical systems. In *Proceedings of the European Conference on Cognitive Ergonomics, Finland*, pages 187–193, 2009.

[71] Andrey Glaschenko, Anton Ivaschenko, George Rzevski, and Petr Skobelev. Multi-agent real time scheduling system for taxi companies. In *8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), Budapest, Hungary*, pages 29–36, 2009.

[72] Jun Ota. Multi-agent robot systems as distributed autonomous systems. *Advanced engineering informatics*, 20(1) :59–70, 2006.

[73] Horst F Wedde, Sebastian Lehnhoff, Edmund Handschin, and Olav Krause. Real-time multi-agent support for decentralized management of electric power. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 43–51. IEEE, 2006.

[74] Volker Grimm, Eloy Revilla, Uta Berger, Florian Jeltsch, Wolf M Mooij, Steven F Railsback, Hans-Hermann Thulke, Jacob Weiner, Thorsten Wiegand, and Donald L DeAngelis. Pattern-oriented modeling of agent-based complex systems : lessons from ecology. *science*, 310(5750) :987–991, 2005.

[75] Yao-qin CHU, Xiao-an LI, and Yong PU. Cooperative control of multi-agent soccer robot system [j]. *Journal of Harbin Institute of Technology*, 7 :911–913, 2004.

[76] Holger Giese, Sven Burmester, Florian Klein, Daniela Schilling, and Matthias Tichy. Multi-agent system design for safety-critical self-optimizing mechatronic systems with uml. In *OOPSLA*, pages 21–32, 2003.

[77] Nicholas R Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4) :35–41, 2001.

[78] Adelinde M Uhrmacher and Danny Weyns. *Multi-Agent systems : Simulation and applications.* CRC press, 2009.

[79] Michał Maciejewski and Kai Nagel. Towards multi-agent simulation of the dynamic vehicle routing problem in matsim. In *International Conference on Parallel Processing and Applied Mathematics*, pages 551–560. Springer, 2011.

[80] Jing Li, Zhaohan Sheng, and Huimin Liu. Multi-agent simulation for the dominant players' behavior in supply chains. *Simulation Modelling Practice and Theory*, 18(6) :850–859, 2010.

[81] Alexia Zoumpoulaki, Nikos Avradinis, and Spyros Vosinakis. A multi-agent simulation framework for emergency evacuations incorporating personality and emotions. In *Hellenic Conference on Artificial Intelligence*, pages 423–428. Springer, 2010.

[82] Audun Botterud, Matthew R Mahalik, Thomas D Veselka, Heon-Su Ryu, and Ki-Won Sohn. Multi-agent simulation of generation expansion in electricity markets. In *Power Engineering Society General Meeting, 2007. IEEE*, pages 1–8. IEEE, 2007.

[83] Catherine Linard, Nicolas Ponçon, Didier Fontenille, and Eric F Lambin. A multi-agent simulation to assess the risk of malaria re-emergence in southern france. *Ecological Modelling*, 220(2) :160–174, 2009.

[84] Evren Bulut, Djamel Khadraoui, and Bertrand Marquet. Multi-agent based security assurance monitoring system for telecommunication infrastructures. In *Proceedings of the Fourth IASTED International Conference on Communication, Network and Information Security*, pages 90–95. ACTA Press, 2007.

[85] Fu-ren Lin, Yu-wei Sung, and Yi-pong Lo. Effects of trust mechanisms on supply-chain performance : A multi-agent simulation study. *International Journal of Electronic Commerce*, 9(4) :9–112, 2005.

[86] Adil V Timofeev, Aleksey V Syrtzev, and Anton V Kolotaev. Network analysis, adaptive control and imitation simulation for multi-agent telecommunication systems. In *Physics and Control, 2005. Proceedings. 2005 International Conference*, pages 112–115. IEEE, 2005.

[87] Jacques Ferber and Alexis Drogoul. Using reactive multi-agent systems in simulation and problem solving. *Distributed artificial intelligence : Theory and praxis*, 5 :53–80, 1992.

[88] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous agents and multi-agent systems*, 9(3) :253–283, 2004.

[89] David Midgley, Robert Marks, and Dinesh Kunchamwar. Building and assurance of agent-based models : An example and challenge to the field. *Journal of Business Research*, 60(8) :884–893, 2007.

[90] Pierre Bommel. *Définition d'un cadre méthodologique pour la conception de modèles multi-agents adaptée à la gestion des ressources renouvelables.* PhD thesis, Université Montpellier II-Sciences et Techniques du Languedoc, 2009.

[91] J Ferber and O Gutknecht. Aaladin : A meta-model fort the analysis and design of organizations in multi-agent systems. In *les actes de 3rd International Conference on multi-agent systems (ICSMAS'98)*, pages 128–135.

[92] Giovanni Caire, F Leal, P Chainho, R Evans, F Garijo, JJ Gomez-Sanz, J Pavon, P Kerney, J Stark, and P Massonet. Message : Methodology for engineering systems of software agents. *Technical Information-European Institute for Research and Strategic Studies in Telecommunications (EURESCOM)*, 2001.

[93] Lin Padgham and Michael Winikoff. Prometheus : A methodology for developing intelligent agents. In *International Workshop on Agent-Oriented Software Engineering*, pages 174–185. Springer, 2002.

[94] Massimo Cossentino. Different perspectives in designing multi-agent systems. *Proceedings of the AGES*, 2, 2002.

[95] Jean-Pierre Georgé, Marie Pierre Gleizes, and Pierre Glize. Conception de systèmes adaptatifs à fonctionnalité émergente : la théorie amas. *Revue d'intelligence artificielle*, 17(4) :591–626, 2003.

[96] Juan Pavón and Jorge Gómez-Sanz. Agent oriented software engineering with ingenias. In *International Central and Eastern European Conference on Multi-Agent Systems*, pages 394–403. Springer, 2003.

[97] Scott A DeLoach. Engineering organization-based multiagent systems. In *International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, pages 109–125. Springer, 2005.

[98] Gauthier Picard. *Méthodologie de développement de systèmes multi-agents adaptatifs et conception de logiciels à fonctionnalité émergente.* PhD thesis, Université Paul Sabatier Toulouse III, 2004.

[99] Aida Kaddoussi. *Optimisation des flux logistiques : vers une gestion avancée de la situation de crise.* PhD thesis, Ecole centrale de Lille, 2012.

[100] Michael Luck, Peter McBurney, and Chris Preist. *Agent technology : enabling next generation computing (a roadmap for agent based computing).* AgentLink, 2003.

[101] Lisa Roux. Evaluation de méthodes composées de fragments. *Rapport de master, Université Paul Sabatier, Toulouse, France*, 2012.

[102] Carole Bernon, Marie-Pierre Gleizes, and Gauthier Picard. Méthodes orientées agent et multi-agent, 2009.

[103] Steven Shapiro and Yves Lespérance. Modeling multiagent systems with casl-a feature interaction resolution application. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 244–259. Springer, 2000.

[104] Steven Shapiro, Yves Lespérance, and Hector J Levesque. The cognitive agents specification language and verification environment for multiagent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems : part 1*, pages 19–26. ACM, 2002.

[105] Steven Shapiro. *Specifying and verifying multiagent systems using the cognitive agents specification language (CASL).* University of Toronto, 2005.

[106] Steven Shapiro, Y Lespérance, and HJ Levesque. The cognitive agents specification language and verification environment. In *Specification and Verification of Multi-agent Systems*, pages 289–315. Springer, 2010.

[107] Hong Zhu. Slabs : A formal specification language for agent-based systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(05) :529–558, 2001.

[108] Hong Zhu. Developing formal specifications of mas in slabs : a case study of evolutionary multi-agent ecosystem. In *Proceedings of the 4th International Conference on Agent-Oriented Information Systems-Volume 59*, pages 20–34. CEUR-WS. org, 2002.

[109] Ji Wang, Rui Shen, and Hong Zhu. Agent oriented programming based on slabs. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 127–132. IEEE, 2005.

[110] Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel. Designing agent behavior with the extensible agent behavior specification language xabsl. In *Robot Soccer World Cup*, pages 114–124. Springer, 2003.

[111] Martin Loetzsch, Max Risler, and Matthias Jüngel. Xabsl-a pragmatic approach to behavior engineering. In *IROS*, pages 5124–5129, 2006.

[112] Max Risler and Oskar von Stryk. Formal behavior specification of multi-robot systems using hierarchical state machines in xabsl. In *AAMAS08-workshop on formal models and methods for multi-robot systems, Estoril, Portugal*. Citeseer, 2008.

[113] Mihai Barbuceanu and Mark S Fox. Cool : A language for describing coordination in multi agent systems. In *ICMAS*, pages 17–24. Citeseer, 1995.

[114] Anand S Rao. Agentspeak (l) : Bdi agents speak out in a logical computable language. In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 42–55. Springer, 1996.

[115] Fabio Y Okuyama, Rafael H Bordini, and Antônio Carlos da Rocha Costa. Elms : An environment description language for multi-agent simulation. In *International Workshop on Environments for Multi-Agent Systems*, pages 91–108. Springer, 2004.

[116] Joachim Fischer, Eckhardt Holz, Martin von Löwis, and Andreas Prinz. Sdl-2000 : A language with a formal semantics. In *Rigorous Object-Oriented Methods*, 2000.

[117] Federico Bergenti and Agostino Poggi. Exploiting uml in the design of multi-agent systems. In *International Workshop on Engineering Societies in the Agents World*, pages 106–113. Springer, 2000.

[118] James J Odell, H Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. In *International Workshop on Agent-Oriented Software Engineering*, pages 121–140. Springer, 2000.

[119] Christoph Oechslein, Franziska Klügl, Rainer Herrler, and Frank Puppe. Uml for behavior-oriented multi-agent simulations. In *International Workshop of Central and Eastern Europe on Multi-Agent Systems*, pages 217–226. Springer, 2001.

[120] H Van Dyke Parunak and James J Odell. Representing social structures in uml. In *International workshop on agent-oriented software engineering*, pages 1–16. Springer, 2001.

[121] Viviane Torres da Silva, Ricardo Choren, and Carlos JP De Lucena. A uml based approach for modeling and implementing multi-agent systems. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 914–921. IEEE Computer Society, 2004.

[122] Darshan S Dillon, Tharam S Dillon, and Elizabeth Chang. Using uml 2.1 to model multi-agent systems. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 1–8. Springer, 2008.

[123] Radovan Červenka, Ivan Trenčanskỳ, Monique Calisti, and Dominic Greenwood. Aml : Agent modeling language toward industry-grade agent-based modeling. In *International Workshop on Agent-Oriented Software Engineering*, pages 31–46. Springer, 2004.

[124] Ivan Trencansky and Radovan Cervenka. Agent modeling language (aml) : A comprehensive approach to modeling mas. *Informatica*, 29(4), 2005.

[125] Agostino Poggi, Giovanni Rimassa, Paola Turci, James Odell, Haralambos Mouratidis, and G Manson. Modeling deployment and mobility issues in multiagent systems using auml. In *International Workshop on Agent-Oriented Software Engineering*, pages 69–84. Springer, 2003.

[126] Sonia Bergamaschi, Gionata Gelati, Francesco Guerra, and Maurizio Vincini. Experiencing auml for the wink multi-agent system. In *WOA*, pages 148–154, 2003.

[127] Luca Cernuzzi and Franco Zambonelli. Experiencing auml in the gaia methodology. In *ICEIS (3)*, pages 283–288. Citeseer, 2004.

[128] Lawrence Cabac and Daniel Moldt. Formal semantics for auml agent interaction protocol diagrams. In *International Workshop on Agent-Oriented Software Engineering*, pages 47–61. Springer, 2004.

[129] Marc-Philippe Huget, James Odell, and Bernhard Bauer. The auml approach. In *Methodologies and software engineering for agent systems*, pages 237–257. Springer, 2004.

[130] Bernhard Bauer, Jörg P Müller, and James Odell. Agent uml : A formalism for specifying multiagent software systems. *International journal of software engineering and knowledge engineering*, 11(03) :207–230, 2001.

[131] Lars Ehrler and Stephen Cranefield. Executing agent uml diagrams. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 906–913. IEEE Computer Society, 2004.

[132] M-P Huget. Agent uml notation for multiagent system design. *IEEE Internet Computing*, 8(4) :63–71, 2004.

[133] Marc-Philippe Huget and James Odell. Representing agent interaction protocols with agent uml. In *International Workshop on Agent-Oriented Software Engineering*, pages 16–30. Springer, 2004.

[134] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347. Springer, 1999.

[135] Sophie Dupuy and Lydie Du Bousquet. A multi-formalism approach for the validation of uml models. *Formal Aspects of Computing*, 12(4) :228–230, 2000.

[136] Jean-Michel Bruel, Johan Lilius, Ana Moreira, and Robert B France. Defining precise semantics for uml. In *European Conference on Object-Oriented Programming*, pages 113–122. Springer, 2000.

[137] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigorous semantics of uml supporting its multiview approach. In *International Conference on Fundamental Approaches to Software Engineering*, pages 171–186. Springer, 2001.

[138] Jing Liu and Jin Song. Linking uml with integrated formal techniques. In *Unified Modeling Language : Systems Analysis, Design and Development Issues*, pages 211–224. IGI Global, 2001.

[139] Marcel Kyas, Harald Fecher, Frank S De Boer, Joost Jacob, Jozef Hooman, Mark Van Der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing uml models and ocl constraints in pvs. *Electronic Notes in Theoretical Computer Science*, 115 :39–47, 2005.

[140] OMG Ocl. 2.0 specification. *Final Adopted Specification ptc/03-10*, 14, 2005.

[141] J.-T Ma, S.-Y Fu, and J.-R Liu. A-adl : An architecture description language for multi-agent systems. 11 :1382–1389, 2000.

[142] Marie-Pierre Gervais and Florin Muscutariu. Towards an adl for designing agent-based systems. In *International Workshop on Agent-Oriented Software Engineering*, pages 263–277. Springer, 2001.

[143] Stéphane Faulkner and Manuel Kolp. Towards an agent architectural description language for information systems. In *ICEIS (3)*, pages 59–66, 2003.

[144] Zhenhua Yu, Yuanli Cai, Ruifeng Wang, and Jiuqiang Han. π-net adl : An architecture description language for multi-agent systems. In *International Conference on Intelligent Computing*, pages 218–227. Springer, 2005.

[145] Haralambos Mouratidis, Manuel Kolp, Stephane Faulkner, and Paolo Giorgini. A secure architectural description language for agent systems. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 578–585. ACM, 2005.

[146] Stéphane Faulkner, Manuel Kolp, Yves Wautelet, and Youssef Achbany. A formal description language for multi-agent architectures. In *Agent-Oriented Information Systems IV*, pages 143–163. Springer, 2008.

[147] N Darragi, EM El-Koursi, and S Collart-Dutilleul. Architecture description language for cyber physical systems analysis : a railway control system case study. *Computers in Railways XIV : Railway Engineering Design and Optimization*, 135 :227–237, 2014.

[148] Tom Holvoet. Agents and petri nets. *Petri Net Newsletter*, 49 :3–8, 1995.

[149] Robert John Allan. *Survey of agent based modelling and simulation tools.* Science & Technology Facilities Council, 2010.

[150] William J Clancey, Maarten Sierhuis, Chin Seah, Chris Buckley, Fisher Reynolds, Tim Hall, and Mike Scott. Multi-agent simulation to implementation : a practical engineering methodology for designing space flight operations. In *International Workshop on Engineering Societies in the Agents World*, pages 108–123. Springer, 2007.

[151] Simon Adameit, Tobias Betz, Lawrence Cabac, Florian Hars, Marcin Hewelt, Michael Köhler-Bußmeier, Daniel Moldt, Dimitri Popov, José Quenum, Axel Theilmann, et al. Modelling distributed network security in a petri net-and agent-based approach. In *German Conference on Multiagent System Technologies*, pages 209–220. Springer, 2010.

[152] KÃ Michael, Roman Langer, Rolf von LÃ1/4de, Daniel Moldt, RÃ Heiko, Rüdiger Valk, et al. Socionic multi-agent systems based on reflexive petri nets and theories of social self-organisation. *Journal of Artificial Societies and Social Simulation*, 10(1) :1–3, 2007.

[153] Michael Köhler, Marcel Martens, and Heiko Rölke. Modelling social behaviour with petri net based multi-agent systems. In *Proceedings of the Workshop MASHO*, volume 3, 2003.

[154] Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling the structure and behaviour of petri net agents. In *International Conference on Application and Theory of Petri Nets*, pages 224–241. Springer, 2001.

[155] Lawrence Cabac, Daniel Moldt, and Heiko Rölke. A proposal for structuring petri net-based agent interaction protocols. In *International Conference on Application and Theory of Petri Nets*, pages 102–120. Springer, 2003.

[156] Djamel Benmerzoug, Fabrice Kordon, and Mahmoud Boufaida. A petri-net based formalisation of interaction protocols applied to business process integration. In *Advances in Enterprise Engineering I*, pages 78–92. Springer, 2008.

[157] Mariusz Nowostawski, Martin Purvis, and Stephen Cranefield. A layered approach for modelling agent conversations. In *the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, pages 163–170, 2001.

[158] Yehia Thabet Kotb, Steven S Beauchemin, and John L Barron. Petri net-based cooperation in multi-agent systems. In *Computer and Robot Vision, 2007. CRV'07. Fourth Canadian Conference on*, pages 123–130. IEEE, 2007.

[159] Fu-Shiung Hsieh. Developing cooperation mechanism for multi-agent systems with petri nets. *Engineering Applications of Artificial Intelligence*, 22(4-5) :616–627, 2009.

[160] Weihua Sheng and Qingyan Yang. Peer-to-peer multi-robot coordination algorithms : petri net based analysis and design. In *Advanced Intelligent Mechatronics. Proceedings, 2005 IEEE/ASME International Conference on*, pages 1407–1412. IEEE, 2005.

[161] C Degano and A Pellegrino. Multi-agent coordination and collaboration for control and optimization strategies in an intermodal container terminal. In *Engineering Management Conference, 2002. IEMC'02. 2002 IEEE International*, volume 2, pages 590–595. IEEE, 2002.

[162] Kolja Lehmann and Daniel Moldt. Modelling and analysis of agent protocols with petri nets. In *German Conference on Multiagent System Technologies*, pages 85–98. Springer, 2004.

[163] Quan Bai, Minjie Zhang, and Haijun Zhang. A colored petri net based strategy for multi-agent scheduling. In *Rational, Robust, and Secure Negotiation Mechanisms in Multi-Agent Systems, 2005*, pages 3–10. IEEE, 2005.

[164] Matej Perše, Matej Kristan, Janez Perš, Gašper Mušič, Goran Vučkovič, and Stanislav Kovačič. Analysis of multi-agent activity using petri nets. *Pattern Recognition*, 43(4) :1491–1501, 2010.

[165] Rajib Kr Chatterjee, Anirban Sarkar, and Swapan Bhattacharya. Modeling and analysis of agent oriented system : Petri net based approach. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'11)*, pages 17–23, 2011.

[166] Said Brahimi, Ramdane Maamri, and Zaidi Sahnoun. Dynamic verification of hierarchical multi-agent plans. *Multiagent and Grid Systems*, 13(2) :113–142, 2017.

[167] Belkacem Athamena and Zina Houhamdi. A petri net based agent behavioral testing. In *American Journal of Applied Sciences*, volume 12, pages 1876–1883, 2011.

[168] Michael Köhler, Daniel Moldt, and Heiko Rölke. Modelling mobility and mobile agents using nets within nets. In *International Conference on Application and Theory of Petri Nets*, pages 121–139. Springer, 2003.

[169] Walid Chainbi. Multi-agent systems : a petri net with objects based approach. In *Intelligent Agent Technology, 2004.(IAT 2004). Proceedings. IEEE/WIC/ACM International Conference on*, pages 429–432. IEEE, 2004.

[170] Kathrin Hoffmann, Hartmut Ehrig, and Till Mossakowski. High-level nets with nets and rules as tokens. In *International Conference on Application and Theory of Petri Nets*, pages 268–288. Springer, 2005.

[171] Lily Chang, Xudong He, J Lian, and S Shatz. Applying a nested petri net modeling paradigm to coordination of sensor networks with mobile agents. In *Proc. of Workshop on Petri Nets and Distributed Systems, Xian, China*, pages 132–145, 2008.

[172] Borhen Marzougui, Khaled Hassine, and Kamel Barkaoui. A new formalism for modeling a multi agent systems : Agent petri nets. *Journal of Software Engineering and Applications*, 3(12) :1118, 2010.

[173] Rajib Kumar Chatterjee, Neha Neha, and Anirban Sarkar. Behavioral modeling of multi agent system : high level petri net based approach. *International Journal of Agent Technologies and Systems (IJATS)*, 7(1) :55–78, 2015.

[174] Faiza Belala and A Boucherit. A contribution to the formal checking of multi-agents systems. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 9–16. IEEE, 2006.

[175] Rafael H Bordini, Louise A Dennis, Berndt Farwer, and Michael Fisher. Automated verification of multi-agent programs. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 69–78. IEEE Computer Society, 2008.

[176] Lăcrămioara Aştefănoaei, Mehdi Dastani, John-Jules Meyer, and Frank S de Boer. A verification framework for normative multi-agent systems. In *Pacific Rim International Conference on Multi-Agents*, pages 54–65. Springer, 2008.

[177] Lacramioara Astefanoaei, Mehdi Dastani, John-Jules Ch Meyer, and Frank S de Boer. On the semantics and verification of normative multi-agent systems. *J. UCS*, 15(13) :2629–2652, 2009.

[178] Farid Mokhati, Brahim Sahraoui, Soufiane Bouzaher, and Mohamed Tahar Kimour. A tool for specifying and validating agents' interaction protocols : From agent uml to maude. *Object Technology*, 9(3), 2010.

[179] Farid Mokhati and Yahia Menassel. Towards formalising use case maps in maude strategy language : application to multi–agent systems. *International Journal of Computer Applications in Technology*, 47(2-3) :138–151, 2013.

[180] M Birna Van Riemsdijk, Frank S De Boer, Mehdi Dastani, and John-Jules Ch Meyer. Prototyping 3apl in the maude term rewriting language. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 95–114. Springer, 2006.

[181] M Birna van Riemsdijk, L Aştefănoaei, and Frank S de Boer. Using the maude term rewriting language for agent development with formal foundations. In *Specification and Verification of Multi-agent Systems*, pages 255–287. Springer, 2010.

[182] Mohamed Amin Laouadi, Farid Mokhati, and Hassina Seridi-Bouchelaghem. A formal framework for organization-centered multi-agent system specification : A rewriting logic based approach. *Multiagent and Grid Systems*, 13(4) :395–419, 2017.

[183] Noura Boudiaf, Farid Mokhati, Mourad Badri, and Linda Badri. Specifying dima multi-agents models using maude. In *Pacific Rim International Workshop on Multi-Agents*, pages 29–42. Springer, 2004.

[184] Noura Boudiaf, Farid Mokhati, and Mourad Badri. Supporting formal verification of dima multi-agents models : towards a framework based on maude model checking. *International Journal of Software Engineering and Knowledge Engineering*, 18(07) :853–875, 2008.

[185] Muaz A Niazi, Amir Hussain, and Mario Kolberg. Verification & validation of agent based simulations using the vomas (virtual overlay multi-agent system) approach. *arXiv preprint arXiv :1708.02361*, 2017.

[186] Panagiotis Kouvaros and Alessio Lomuscio. Automatic verification of parameterised multi-agent systems. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 861–868. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[187] Mehdi Dastani, Koen V Hindriks, and John-Jules Meyer. *Specification and verification of multi-agent systems*. Springer Science & Business Media, 2010.

[188] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. Verification of protocol conformance and agent interoperability. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 265–283. Springer, 2005.

[189] Amal El Fallah-Seghrouchni, Irene Degirmenciyan-Cartault, and Frédéric Marc. Modelling, control and validation of multi-agent plans in dynamic context. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 44–51. IEEE Computer Society, 2004.

[190] SM Manson. Validation and verification of multi-agent models for ecosystem management. *Complexity and ecosystem management : The theory and practice of multi-agent approaches*, pages 63–74, 2003.

[191] Pablo Gruer, Vincent Hilaire, and Abder Koukam. Towards verification of multi-agent systems. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 393–394. IEEE, 2000.

[192] Alireza Souri and Nima Jafari Navimipour. Behavioral modeling and formal verification of a resource discovery approach in grid computing. *Expert Systems with Applications*, 41(8) :3831–3849, 2014.

[193] Wing Lok Yeung. Behavioral modeling and verification of multi-agent systems for manufacturing control. *Expert Systems with applications*, 38(11) :13555–13562, 2011.

[194] Tibor Bosse, Catholijn M Jonker, Lourens Van der Meij, Alexei Sharpanskykh, and Jan Treur. Specification and verification of dynamics in agent models. *International Journal of Cooperative Information Systems*, 18(01) :167–193, 2009.

[195] Tibor Bosse, Dung N Lam, and K Suzanne Barber. Automated analysis and verification of agent behavior. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1317–1319. ACM, 2006.

[196] Amal El Fallah-Seghrouchni, Serge Haddad, and Hamza Mazouzi. Protocol engineering for multi-agent interaction. In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, pages 89–101. Springer, 1999.

[197] Hamza Mazouzi, Amal El Fallah Seghrouchni, and Serge Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems : part 2*, pages 517–526. ACM, 2002.

[198] Marco Alberti, Davide Daolio, Paolo Torroni, Marco Gavanelli, Evelina Lamma, and Paola Mello. Specification and verification of agent interaction protocols in a logic-based system. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 72–78. ACM, 2004.

[199] Michael Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous agents and multi-agent systems*, 3(1) :9–31, 2000.

[200] X Vila, A Schuster, and Adolfo Riera. Security for a multi-agent system based on jade. *computers & security*, 26(5) :391–400, 2007.

[201] Clare Dixon, M-CF Gago, Michael Fisher, and Wiebe Van Der Hoek. Using temporal logics of knowledge in the formal verification of security protocols. In *Temporal Representation and Reasoning, 2004. TIME 2004. Proceedings. 11th International Symposium on*, pages 148–151. IEEE, 2004.

[202] Petr Novák, Milan Rollo, Jiří Hodík, and Tomáš Vlček. Communication security in multi-agent systems. In *International Central and Eastern European Conference on Multi-Agent Systems*, pages 454–463. Springer, 2003.

[203] Michael Wooldridge, Michael Fisher, Marc-Philippe Huget, and Simon Parsons. Model checking multi-agent systems with mable. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems : part 2*, pages 952–959. ACM, 2002.

[204] Massimo Benerecetti and Alessandro Cimatti. Symbolic model checking for multi-agent systems. *Proc. MoChart*, 2 :1–8, 2002.

[205] Rafael H Bordini, Michael Fisher, Carmen Pardavila, Willem Visser, and Michael Wooldridge. Model checking multi-agent programs with casp. In *International Conference on Computer Aided Verification*, pages 110–113. Springer, 2003.

[206] Rafael H Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking agentspeak. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 409–416. ACM, 2003.

[207] Peter Gammie and Ron Van Der Meyden. Mck : Model checking the logic of knowledge. In *International Conference on Computer Aided Verification*, pages 479–483. Springer, 2004.

[208] Alessio Lomuscio and Franco Raimondi. Mcmas : A model checker for multi-agent systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 450–454. Springer, 2006.

[209] Alessio Lomuscio, Charles Pecheur, and Franco Raimondi. Automatic verification of knowledge and time with nusmv. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 1384–1389. IJCAI/AAAI Press, 2007.

[210] Magdalena Kacprzak, Wojciech Nabiałek, Artur Niewiadomski, Wojciech Penczek, Agata Półrola, Maciej Szreter, Bożena Woźna, and Andrzej Zbrzezny. Verics 2007-a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1-4) :313–328, 2008.

[211] Supriya D'Souza, Abhishek Rao, Amit Sharma, and Sanjay Singh. Modeling and verification of a multi-agent argumentation system using nusmv. *arXiv preprint arXiv :1209.4330*, 2012.

[212] Eric Ramat. Introduction à la modélisation et à la simulation à événements discrets. *Modélisation et simulation multi-agents : Applications pour les sciences de l'homme et de la société*, pages 50–73, 2006.

[213] Gauthier Quesnel. Approche formelle et opérationnelle de la multi-modélisation et de la simulation des systèmes complexes. *PHD trésis Laboratoire d'Informatique du Littoral (LIL). Calais-France*, 2006.

[214] Jianli Xu and Juha Kuusela. Modeling execution architecture of software system using colored petri nets. In *Proceedings of the 1st international workshop on Software and performance*, pages 70–75. ACM, 1998.

[215] Jianli Xu and Juha Kuusela. Analyzing the execution architecture of mobile phone software with colored petri nets. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(2) :133–143, 1998.

[216] Hassan Reza and Xudong He. *A framework for specifying software architecture based on petri net pattern*. PhD thesis, PhD Dissertation, Department of Computer Science, North Dakota State University, 2002.

[217] Robert G Pettit IV and Hassan Gomaa. Modeling behavioral patterns of concurrent software architectures using petri nets. In *4th Working IEEE / IFIP Conference on Software Architecture (WICSA 2004)*, pages 57–68. IEEE, 2004.

[218] Sima Emadi and Fereidoon Shams. A comparison of petri net based approaches used for specifying the executable model of software architecture. In *IMECS*, pages 1104–1109, 2007.

[219] João M Fernandes. Combining petri nets and uml for model-based software engineering. CEUR Workshop Proceedings, 2012.

[220] Sima Emadi and Fereidoon Shams. Modeling of component diagrams using petri nets. *Indian Journal of Science and Technology*, 3(12) :1151–1161, 2010.

[221] Sima Emadi and Fereidoon Shams. Transformation of usecase and sequence diagrams to petri nets. In *Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on*, volume 4, pages 399–403. IEEE, 2009.

[222] Nasreddine Aoumeur. Stepwise rigorous development of distributed agile information systems : from uml-diagrams to component-based petri nets. *Enterprise Information Systems*, 2(2) :121–156, 2008.

[223] Sima Emadi and Fereidoon Shams. From uml component diagram to an executable model based on petri nets. In *Information Technology, 2008. ITSim 2008. International Symposium on*, volume 4, pages 1–8. IEEE, 2008.

[224] Javier Campos and José Merseguer. On the integration of uml and petri nets in software development. In *International Conference on Application and Theory of Petri Nets*, pages 19–36. Springer, 2006.

[225] Zhaoxia Hu and Sol M Shatz. Mapping uml diagrams to a petri net notation for system simulation. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2004)*, pages 213–219, 2004.

[226] Kimiyuki Fukuzawa and Motoshi Saeki. Evaluating software architectures by coloured petri nets. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 263–270. ACM, 2002.

[227] Luciano Baresi and Mauro Pezzè. Improving uml with petri nets. *Electronic Notes in Theoretical Computer Science*, 44(4) :1–13, 2001.

[228] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. A system dependability modeling framework using aadl and gspns. In *Architecting Dependable Systems IV*, pages 14–38. Springer, 2007.

[229] A-E Rugina, Karama Kanoun, and Mohamed Kaâniche. The adapt tool : From aadl architectural models to stochastic petri nets through model transformation. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 85–90. IEEE, 2008.

[230] Xavier Renault, Fabrice Kordon, and Jérôme Hugues. From aadl architectural models to petri nets : Checking model viability. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC'09. IEEE International Symposium on*, pages 313–320. IEEE, 2009.

[231] Wenxin Wu and Motoshi Saeki. Specifying software architectures based on coloured petri nets. *IEICE TRANSACTIONS on Information and Systems*, 83(4) :701–712, 2000.

[232] Julia Padberg. Safety properties in petri net modules. *Journal of Integrated Design and Process Science*, 8(4) :65–78, 2004.

[233] Dianxiang Xu, Priti Borse, Ken Grigsby, and Kendall E Nygard. A petri net based software architecture for uav simulation. In *Software Engineering Research and Practice*, pages 227–234, 2004.

[234] Sima Emadi and Fereidoon Shams. A new executable model for software architecture based on petri net. *Indian Journal of Science and Technology*, 2(9) :15–25, 2009.

[235] Zhenhua Yu and Yuanli Cai. Object-oriented petri nets based architecture description language for multi-agent systems. *IJCSNS*, 6(1) :123–131, 2006.

[236] Zhenhua Yu and Zhiwu Li. Architecture description language based on object-oriented petri nets for multi-agent systems. In *Networking, Sensing and Control, 2005. Proceedings. 2005 IEEE*, pages 256–260. IEEE, 2005.

[237] Roder Koci, Zdenek Mazal, Frantisek Zboril, and Vladimir Janousek. Modeling deliberative agents using object oriented petri nets. In *Intelligent Systems Design and Applications, 2007. ISDA 2007. Seventh International Conference on*, pages 15–20. IEEE, 2007.

[238] Zdenek Mazal, Radek Kocí, Vladimír Janoušek, and František Zboril. Pnagent : a framework for modelling bdi agents using object oriented petri nets. In *Intelligent Systems Design and Applications, 2008. ISDA'08. Eighth International Conference on*, volume 2, pages 420–425. IEEE, 2008.

[239] Dianxiang Xu, Richard Volz, Thomas Ioerger, and John Yen. Modeling and verifying multi-agent behaviors using predicate/transition nets. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 193–200. ACM, 2002.

[240] Irina A Lomazova. Nested petri nets : Multi-level and recursive systems. *Fundamenta Informaticae*, 47(3-4) :283–293, 2001.

[241] Zhang Qiqian, Zhou Guomin, and Si Jin. A robot architecture validating method with agent oriented hierarchical petri net. In *Proceedings of the ICAL 2008, IEEE International Conference on Automation and Logistics*, pages 2481–2485. IEEE, 2008.

[242] X. Tao, Y. Miao, Y. Zhang, and Z. Shen. Collaborative medical diagnosis through fuzzy petri net based agent argumentation. In *Proceedings of the IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1197–1204. IEEE, 2014.

[243] Jon Whittle, João Araújo, Ambrosio Toval, and Jose Luis Fernández Alemán. Rigorously automating transformations of uml behavior models. *Dynamic Behaviour in UML Models : Semantic Questions in conjunction with UML*, 2000.

[244] Alexander Knapp. Generating rewrite theories from uml collaborations. In *Cafe : An Industrial-Strength Algebraic Formal Method*, pages 97–120. Elsevier, 2000.

[245] Nasreddine Aoumeur and Gunter Saake. Integrating and rapid-prototyping uml structural and behavioural diagrams using rewriting logic. In *CAiSE*, pages 296–310. Springer, 2002.

[246] Farid Mokhati and Mourad Badri. Generating maude specifications from uml use case diagrams. *Journal of Object Technology*, 8(2) :319–136, 2009.

[247] Kyungmin Bae, Peter Csaba Ölveczky, José Meseguer, and Abdullah Al-Nayeem. The synchaadl2maude tool. In *International Conference on Fundamental Approaches to Software Engineering*, pages 59–62. Springer, 2012.

[248] Kyungmin Bae, Peter Csaba Ölveczky, Abdullah Al-Nayeem, and José Meseguer. Synchronous aadl and its formal analysis in real-time maude. In *Formal Methods and Software Engineering*, pages 651–667. Springer, 2011.

[249] Peter Csaba Ölveczky, Artur Boronat, and José Meseguer. Formal semantics and analysis of behavioral aadl models in real-time maude. In *Formal Techniques for Distributed Systems*, pages 47–62. Springer, 2010.

[250] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a formal semantics for the aadl behavior annex. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 1166–1171. IEEE, 2009.

[251] Chadlia Jerad, Kamel Barkaoui, and Amel Grissa-Touzi. On the use of real-time maude for architecture description and verification : A case study. In *BCS Int. Acad. Conf.*, pages 305–317, 2008.

[252] Christiano Braga and Alexandre Sztajnberg. Towards a rewriting semantics for a software architecture description language. *Electronic Notes in Theoretical Computer Science*, 95 :149–168, 2004.

[253] Sahar Smaali, Aicha Choutri, and Faiza Belala. K semantics for dynamic software architectures. In *Proceedings of the International Arab Conference on Information Technology (ACIT '2013)*, 2013.

[254] Chafia Bouanaka and Faiza Belala. Towards a mobile architecture description language. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 743–748. IEEE, 2008.

[255] Alexandre Rademaker, Christiano Braga, and Alexandre Sztajnberg. A rewriting semantics for a software architecture description language. *Electronic Notes in Theoretical Computer Science*, 130 :345–377, 2005.

[256] Wolfgang Reisig. *A primer in Petri net design*. Springer Science & Business Media, 2012.

[257] George Fink and Matt Bishop. Property-based testing : a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4) :74–80, jul 1997.

[258] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, pages 154–163. IEEE, 1994.

[259] Laura M Castro. Advanced management of data integrity : property-based testing for business rules. *Journal of Intelligent Information Systems*, 44(3) :355–380, 2015.

[260] Wilhelm Hasselbring. Software architecture : Past, present, future. In *The Essence of Software Engineering*, pages 169–184. Springer, 2018.

[261] Najd Altoyan and Dewayne E Perry. Towards a well-formed software architecture analysis. In *Proceedings of the 11th European Conference on Software Architecture : Companion Proceedings*, pages 173–179. ACM, 2017.

[262] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 10 years of software architecture knowledge management : Practice and future. *Journal of Systems and Software*, 116 :191–205, 2016.

[263] Chun Jian Wang, Hong Jun Fan, and Shuang Pan. Research on mapping uml to petri-net in system modeling. In *MATEC Web of Conferences*, volume 44. EDP Sciences, 2016.

[264] Wojciech Szmuc and Tomasz Szmuc. Modeling uml object event handling with petri nets. In *Mixed Design of Integrated Circuits and Systems, 2016 MIXDES-23rd International Conference*, pages 454–457. IEEE, 2016.

[265] Xavier Renault, Fabrice Kordon, and Jerome Hugues. Adapting models to model checkers, a case study : Analysing aadl using time or colored petri nets. In *Rapid System Prototyping, 2009. RSP'09. IEEE/IFIP International Symposium on*, pages 26–33. IEEE, 2009.

[266] Hassan Reza and Emanuel S Grant. Toward extending aadl-osate toolset with color petri nets (cpns). In *Information Technology : New Generations, 2009. ITNG'09. Sixth International Conference on*, pages 1085–1088. IEEE, 2009.

[267] Hassan Reza and Amrita Chatterjee. Mapping aadl to petri net tool-sets using pnml framework. *Journal of Software Engineering and Applications*, 7(11) :920, 2014.

[268] David Garlan, Bradley R Schmerl, and Shang-Wen Cheng. Software architecture-based self-adaptation. *Autonomic computing and networking*, 1 :31–55, 2009.

[269] Chi Zhang, Yunyun Ma, Xiaohua Wang, and Ruixue Wang. Software architecture modeling and reliability evaluation based on petri net. In *Dependable Systems and Their Applications (DSA), 2017 International Conference on*, pages 51–56. IEEE, 2017.

[270] Sima Emadi. Modeling of component diagrams using petri nets. *International Research Journal Of Biochemical Technology*, 2(2) :1–7, 2016.

[271] Meuse NO Junior, Paulo Maciel, Ricardo Lima, Angelo Ribeiro, Cesar Oliveira, Adilson Arcoverde, Raimundo Barreto, Eduardo Tavares, and Leornado Amorin. A retargetable environment for power-aware code evaluation : An approach based on coloured petri net. *Lecture notes in computer science*, 3728 :49, 2005.

[272] Chadlia Jerad and Kamel Barkaoui. On the use of rewriting logic for verification of distributed software architecture description based lfp. In *null*, pages 202–208. IEEE, 2005.

[273] Farid Mokhati, Patrice Gagnon, and Mourad Badri. Verifying uml diagrams with model checking : A rewriting logic based approach. In *Quality Software, 2007. QSIC'07. Seventh International Conference on*, pages 356–362. IEEE, 2007.

[274] Patrice Gagnon, Farid Mokhati, and Mourad Badri. Applying model checking to concurrent uml models. *Journal of Object Technology*, 7(1) :59–84, 2008.

[275] Xudong He, Junhua Ding, and Yi Deng. Model checking software architecture specifications in sam. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 271–274. ACM, 2002.

[276] Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Software architecture modeling & analysis : A rigorous approach. In *Formal Methods for Software Architectures*, pages 44–51. Springer, 2003.

[277] Dimitra Giannakopoulou. *Model checking for concurrent software architectures*. PhD thesis, University of London, 1999.

[278] Pengcheng Zhang, Henry Muccini, and Bixin Li. A classification and comparison of model checking software architecture techniques. *Journal of Systems and Software*, 83(5) :723–744, 2010.

[279] Antonio Bucchiarone, Henry Muccini, Patrizio Pelliccione, and Pierluigi Pierini. Model-checking plus testing : from software architecture analysis to code testing. *Applying Formal Methods : Testing, Performance, and M/E-Commerce*, pages 351–365, 2004.

[280] H Muccini. Software architecture for testing, coordination and views model checking. 2002.

[281] Henry Muccini. Software architecture-based testing and model-checking. 2005.

[282] Lin Gui, Jun Sun, Yang Liu, Yuan Jie Si, Jin Song Dong, and Xin Yu Wang. Combining model checking and testing with an application to reliability prediction and distribution. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 101–111. ACM, 2013.

[283] Elthon Oliveira, Hyggo Almeida, and Leandro Silva. Formal modelling and verification of a component model using coloured petri nets and model checking. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1427–1431. ACM, 2007.

[284] Shiladitya Pujari and Sripati Mukhopadhyay. Petri net : A tool for modeling and analyze multi-agent oriented systems. *International Journal of Intelligent Systems and Applications*, 4(10) :103, 2012.

[285] Robert Lukomski and Kazimierz Wilkosz. Modeling of multi-agent system for power system topology verification with use of petri nets. In *IEEE Conferences*, pages 1–6, 2010.

[286] XM Yu, Jun Zeng, HX Guo, and Dan Liu. Distributed wind-pv system based on multi-agent and petri nets. *Control Theory & Applications*, 25(2) :353–356, 2008.

[287] Jose R Celaya, Alan A Desrochers, and Robert J Graves. Modeling and analysis of multi-agent systems using petri nets. In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pages 1439–1444. IEEE, 2007.

[288] Amal El Fallah-Seghrouchni, Jorge J Gomez-Sanz, and Munindar P Singh. Formal methods in agent-oriented software engineering. In *International Workshop on Agent-Oriented Software Engineering*, pages 213–228. Springer, 2009.

[289] Zina Houhamdi. Multi-agent system testing : A survey. *International Journal of Advanced Computer*, 2011.

[290] Wojciech Jamroga and Wojciech Penczek. Specification and verification of multi-agent systems. In *Lectures on Logic and Computation*, pages 210–263. Springer, 2012.

[291] Michael Winikoff and Lin Padgham. Agent oriented software engineering. *Multiagent systems*, page 2, 2013.

[292] Mohamed Nezar Abourraja, Mustapha Oudani, Mohamed Yassine Samiri, Dalila Boudebous, Abdelaziz El Fazziki, Mehdi Najib, Abdelhadi Bouain, and Naoufal Rouky. A multi-agent based simulation model for rail–rail transshipment : An engineering approach for gantry crane scheduling. *IEEE Access*, 5 :13142–13156, 2017.

[293] Charles M Macal and Michael J North. Tutorial on agent-based modelling and simulation. *Journal of simulation*, 4(3) :151–162, 2010.

[294] Artur Niewiadomski, Wojciech Penczek, and Maciej Szreter. Verics 2004 : A model checker for real time and multi-agent systems. In *Humboldt University*. Citeseer, 2004.

[295] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. Mcmas : A model checker for the verification of multi-agent systems. In *International conference on computer aided verification*, pages 682–688. Springer, 2009.

[296] Ryan Kirwan and Alice Miller. Model checking multi-agent systems. In *Automated Reasoning Workshop 2010 Bridging the Gap between Theory and Practice ARW 2010*, page 18, 2010.

[297] Peter Csaba Ölveczky. *Real-Time Maude 2.3 Manual*, 2007. `http://www.ifi.uio.no/RealTimeMaude/`.

[298] José Meseguer and Peter CsabaOlveczky. Real-time maude : A tool for simulating and analyzing real-time and hybrid systems. *Electronic Notes in Theoretical Computer Science*, 36 :361–382, 2000.

[299] Shiyong Wang, Jiafu Wan, Daqiang Zhang, Di Li, and Chunhua Zhang. Towards smart factory for industry 4.0 : a self-organized multi-agent system with big data based feedback and coordination. *Computer Networks*, 101 :158–168, 2016.

[300] Victor Lesser, Charles L Ortiz Jr, and Milind Tambe. *Distributed sensor networks : A multiagent perspective*, volume 9. Springer Science & Business Media, 2012.

[301] Thillainathan Logenthiran, Dipti Srinivasan, and Ashwin M Khambadkone. Multi-agent system for energy resource scheduling of integrated microgrids in a distributed system. *Electric Power Systems Research*, 81(1) :138–148, 2011.

[302] Mihalis Giannakis and Michalis Louis. A multi-agent based framework for supply chain risk management. *Journal of Purchasing and Supply Management*, 17(1) :23–31, 2011.

[303] Arend Ligtenberg, Ron JA van Lammeren, Arnold K Bregt, and Adrie JM Beulens. Validation of an agent-based model for spatial planning : A role-playing approach. *Computers, Environment and Urban Systems*, 34(5) :424–434, 2010.

[304] K Suzanne Barber, Karen Fullam, and Joonoo Kim. Challenges for trust, fraud and deception research in multi-agent systems. In *Workshop on Deception, Fraud and Trust in Agent Societies*, pages 8–14. Springer, 2002.

[305] Sarvapali D Ramchurn, Dong Huynh, and Nicholas R Jennings. Trust in multi-agent systems. *The Knowledge Engineering Review*, 19(1) :1–25, 2004.

[306] Alex Rogers, Sarvapali D Ramchurn, and Nicholas R Jennings. Delivering the smart grid : Challenges for autonomous agents and multi-agent systems research. In *AAAI*, 2012.

[307] Can Can Zhao, Xiao Dong Zhang, Shao Juan Lei, and Jun Jiang Qiu. Agent-based modeling and simulation on multi-stage supply chain operation. In *Advanced Materials Research*, volume 291, pages 3216–3220. Trans Tech Publ, 2011.

[308] Kunihiko Hiraishi. A petri-net-based model for the mathematical analysis of multi-agent systems. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 84(11) :2829–2837, 2001.

[309] R Pais, JP Barros, and L Gomes. From petri net models to c implementation of digital controllers. In *2010 IEEE International Symposium on Industrial Electronics*, 2010.

[310] EA Golenkov, AS Sokolov, GV Tarasov, and DI Kharitonov. Experimental version of parallel programs translator from petri nets to c++. In *International Conference on Parallel Computing Technologies*, pages 226–231. Springer, 2001.

[311] Nils Hagge and Bernardo Wagner. Java code patterns for petri net based behavioral models. In *Industrial Informatics, 2005. INDIN'05. 2005 3rd IEEE International Conference on*, pages 450–455. IEEE, 2005.

[312] Roberto Willrich, Pierre De Saqui-Sannes, Patrick Sénac, and Michel Diaz. Multimedia authoring with hierarchical timed stream petri nets and java. *Multimedia Tools and Applications*, 16(1-2) :7–27, 2002.

[313] Luís Gomes, Anikó Costa, João Paulo Barros, and Paulo Lima. From petri net models to vhdl implementation of digital controllers. In *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*, pages 94–99. IEEE, 2007.

[314] Oscar R Ribeiro and Joao M Fernandes. Translating synchronous petri nets into promela for verifying behavioural properties. In *Industrial Embedded Systems, 2007. SIES'07. International Symposium on*, pages 266–273. IEEE, 2007.

[315] Noura Boudiaf, Allaoua Chaoui, and Hichem Bakha. A rewriting logic based tool for ecatnets'analysis : Edition and simulation steps description. *Editorial Board*, 6(2) :16, 2005.

[316] Elhillali Kerkouche and Allaoua Chaou. A graphical tool support to process and simulate ecatnets models based on meta-modelling and graph grammars. *INFOCOMP*, 8(4) :37–44, 2009.

[317] Noura Boudiaf and Abdelhamid Djebbar. Towards an automatic translation of colored petri nets to maude language. *International Journal of Computer Science & Engineering*, 3(1), 2009.

[318] Wolfgang Reisig. *Petri nets : an introduction*, volume 4. Springer Science & Business Media, 2012.

[319] Gerald C Gannod and Sunil Gupta. An automated tool for analyzing petri nets using spin. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 404–407. IEEE, 2001.

[320] Nicholas J Dingle, William J Knottenbelt, and Tamas Suto. Pipe2 : a tool for the performance evaluation of generalised stochastic petri nets. *ACM SIGMETRICS Performance Evaluation Review*, 36(4) :34–39, 2009.

[321] Dianxiang Xu. A tool for automated test code generation from high-level petri nets. *Applications and Theory of Petri Nets*, pages 308–317, 2011.

[322] Maria Paola Cabasino, Alessandro Giua, Stéphane Lafortune, and Carla Seatzu. A new approach for diagnosability analysis of petri nets using verifier nets. *IEEE Transactions on Automatic Control*, 57(12) :3104–3117, 2012.

[323] Kamel Barkaoui, Hanifa Boucheneb, and Awatef Hicheur. Modelling and analysis of time-constrained flexible workflows with time recursive ecatnets. In *International Workshop on Web Services and Formal Methods*, pages 19–36. Springer, 2008.

[324] Alexander Schulz. Model checking of reconfigurable Petri nets. *arXiv preprint arXiv :1409.8404*, 2014.

[325] Xudong He, Reng Zeng, Su Liu, Zhuo Sun, and Kyungmin Bae. A term rewriting approach to analyze high level petri nets. In *Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on*, pages 109–112. IEEE, 2016.

[326] Julia Padberg and Alexander Schulz. Model checking reconfigurable petri nets with maude. In *International Conference on Graph Transformation*, pages 54–70. Springer, 2016.

[327] Chadlia Jerad, Kamel Barkaoui, and Amel Grissa Touzi. Hierarchical verification in maude of lfp software architectures. In *European Conference on Software Architecture*, pages 156–170. Springer, 2007.

[328] Manuel Clavel, Francisco Durán, Steven Eker, P Lincoln, N Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.3), 2007. *URL : http ://maude. cs. uiuc. edu/maude2-manual*, 2007.

[329] T Benarbia, K Labadi, A Omari, and JP Barbot. Balancing dynamic bike-sharing systems : A petri nets with variable arc weights based approach. In *Control, Decision and Information Technologies (CoDIT), 2013 International Conference on*, pages 112–117. IEEE, 2013.